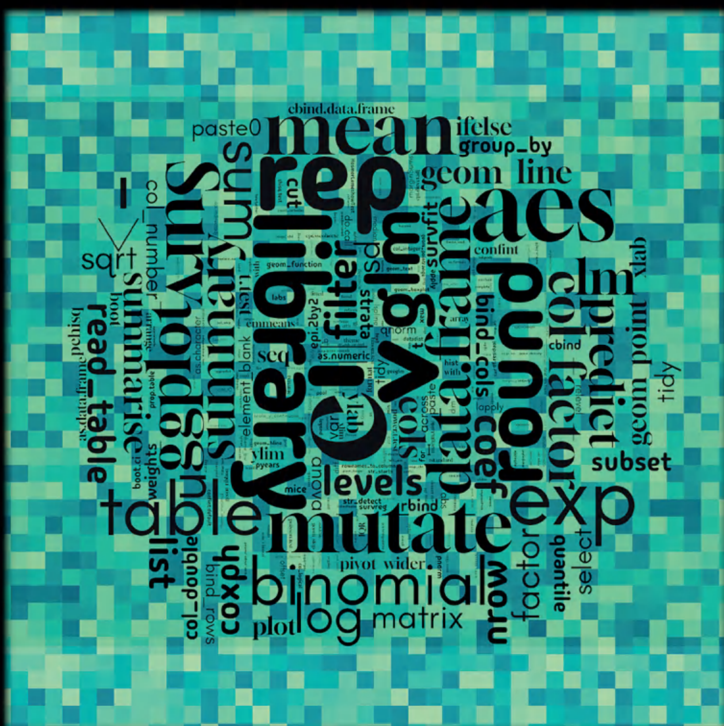
R Companion to Epidemiology: Study Design and Data Analysis



Ajith R

A Chapman & Hall Book

CRC Press
Taylor & Francis Group

R
Companion
to
Epidemiology: Study Design and Data Analysis

R Companion to Epidemiology: Study Design and Data Analysis is a companion volume to the classic textbook by Mark Woodward, *Epidemiology: Study Design and Data Analysis, Third Edition*. It aims to equip the reader with sufficient knowledge to use R for practising epidemiology. Towards this aim, it reworks the examples in the textbook, presenting the code followed by an explanation and its result.

Key Features:

- Almost all of the numerical examples in the textbook are reworked in R
- R code is introduced in small portions and explained thoroughly
- Complexity of introduced code is increased only gradually
- More than 300 commands spanning more than 40 libraries are introduced

The book is intended primarily to be used as a supplement to the textbook by undergraduate and graduate students in the fields of epidemiology and statistics. It will also serve practitioners and researchers in epidemiology who want to learn R for use in their work.

Dr Ajith R worked as a primary care physician for 21 years after completing graduation. He has a postgraduate diploma in clinical pathology and has completed India Epidemic Intelligence Service Training.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

R Companion to Epidemiology: Study Design and Data Analysis

Ajith R

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

First edition published 2025
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton, FL 33431

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

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

ISBN: 978-1-032-96465-2 (hbk)
ISBN: 978-1-032-96464-5 (pbk)
ISBN: 978-1-003-58956-3 (ebk)

DOI: 10.1201/9781003589563

Typeset in Latin Modern font
by KnowledgeWorks Global Ltd.

Publisher's note: This book has been prepared from camera-ready copy provided by the authors.

Operating System: Debian GNU/Linux trixie/sid
Kernel: Linux 6.11.4-amd64
Architecture: x86-64
R version 4.4.2 (2024-10-31)

R package : version used
----
base : 4.4.2
boot : 1.3-31
broom : 1.0.7
car : 3.1-3
coin : 1.4-3
DescTools : 0.99.49
dplyr : 1.1.4
effsize : 0.8.1
emmeans : 1.10.5
Epi : 2.56
epiDisplay : 3.5.0.2
epiR : 2.0.76
flextable : 0.9.2
geepack : 1.3.12
ggeffects : 1.7.2
ggplot2 : 3.5.1
graphics : 4.4.2
Greg : 2.0.2
gridExtra : 2.3
Hmisc : 5.2-0
lmtest : 0.9-40
lspline : 1.0-0
lubridate : 1.9.2
MASS : 7.3-61
MatchIt : 4.5.5
metafor : 4.6-0
mice : 3.16.0
popEpi : 0.4.12
pROC : 1.18.5
purrr : 1.0.2
readr : 2.1.5
riskRegression : 2023.12.21
rms : 6.8-2
ROCit : 2.1.2
sandwich : 3.1-1
sjmisc : 2.8.10
splines : 4.4.2
stats : 4.4.2
stringr : 1.5.0
survival : 3.7-0
SurvRegCensCov : 1.7
tibble : 3.2.1
tidyr : 1.3.1
tinytable : 0.5.0
utils : 4.4.2
VIM : 6.2.2

To my parents,

for they are the reason why I am what I am

Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

List of Figures

List of Tables

Foreword

This book is a wonderful companion to the third edition of my textbook, *Epidemiology: Study Design and Data Analysis*. It is comprehensive and thoughtfully laid out. Most importantly to me, it echoes the underlying principles of my work, which were to educate and motivate the reader by leading them systematically through ideas and methodology with explanations and examples. The author follows my book in sequence, providing R code and explaining how he derived it, for not only the modelling examples I provided but also the descriptive statistics and the figures. Quite rightly, he acknowledges that there are often other ways to obtain the same results, which may prompt the reader to instigate their own investigation into R.

In 1989, the World Wide Web was launched by Sir Tim Berners-Lee, personal computers were still rare, undergraduate courses in statistics in the U.K. had only recently introduced computer labs for practical instruction, and I started writing what became the first edition of my textbook. Due to my own experience working on international aid projects in Africa and Asia, and teaching visitors from those countries to the Department of Applied Statistics at Reading University, U.K. where I was based in the 1990s, I wanted to make the book useful to students and researchers in low- and middle-income countries where computers were relatively slow to come into routine use. Therefore, it felt essential to provide much detail on arithmetic methods that did not rely on the use of statistical software (apps were not even a concept then). Thus, the book was in two parts; by and large, the "design" part of the book, including formulae, was the first half and the "analysis" part was the second. The passage of time has shown that readers appreciate this progression, and the way concepts are introduced with examples.

Nevertheless, I had to choose which statistical software package to use in the modelling chapters of the first edition. At Reading, as well as our own software and routines, we mainly used Minitab to teach undergraduates and GENSTAT and GLIM for postgrads and for research. The former was too simplistic for my purposes, although I did use it to produce the figures. The latter two packages were British, and the local mantra was that US packages gave a lot of output for little effort, whilst the U.K. ones gave little output for a lot of effort – and thus were superior! SPSS was regarded as the number one criminal as far as leaving the user to decide whatever result (p-value, usually) best suited their hypothesis. So that was out for my book. SAS was thought to be (a bit) better and had the huge advantages of having a broad array of procedures and being widely used by Big Pharma; little by little it was usurping GLIM and GENSTAT at Reading. I liked it myself and hence I chose SAS.

SAS remained the only package used in the book's second edition. However, I discovered Stata when asked, with a few days' notice, to give a short course which involved computer labs using that package, in Antwerp. It turned out that, like Minitab, it was very easy to learn and handled the usual basic statistical procedures in a logical fashion with informative (but limited!) output. Later, whilst working at Johns Hopkins University in the 2010s, I discovered that it was also a superb tool for research, with a plethora of commands and user-supplied add-ons. It was also more accessible than SAS, although SAS was still the choice of many professionals. Hence, the third edition included both SAS and Stata output and online code. But that was ten years ago, and undoubtedly, R is now the main statistical package of choice for the type of reader that my book targets. As a not-for-profit undertaking, with strong input from leading statisticians, it is surely going to be around for many years, and I would recommend its use.

I am flattered that Dr Ajith R found my book so useful that he took on the mammoth task of working out how the vast majority of material in the book could be redone with R, and then writing such a masterful text around the code he produced. I would argue that the book you have in your hand, or on your screen, is an excellent primer for R even if you don't use it in conjunction with my book. I look forward to trying out his code and learning much more about R. Many thanks to him, but also to Rob Calver of Chapman & Hall/CRC Press for his support of both this book and my own.

Mark Woodward

Sydney, Australia

Preface

I bought *Epidemiology: Study Design and Data Analysis, Third Edition* while I was undergoing epidemiology training in 2014-16. I was impressed by the breadth of the topics covered and by the fact that the topics were explained thoroughly without burdening the reader with mathematical details. However, I couldn't use any software I knew to work the examples myself, and SAS and STATA were too costly to afford (Later, I learned that SAS is also available for free for independent learners). That is when I learned about R.

R is free and open source to mean not only that we needn't pay anything to use it, but also to mean that you are free to modify the software if you are capable. This means that many more people can use it, unlike paid proprietary software, and that many more people contribute to bring cutting edge methods to R. Even for common problems, there are multiple choices of solutions to choose from.

I learned R by myself using free online resources which are innumerable. It was not very easy, but I enjoyed it. How could I know that the method I was employing was correct? *Epidemiology: Study Design and Data Analysis* had made that easy by providing the data sets it used. I could use the data, rework the examples and confirm from the book if they were correct. When the results were not as in the book, I would read more to find the reasons and correct it if needed. I wrote down the code I learned and made some notes.

As I slowly progressed through the chapters, I grew confident. I thought, "Why not share the code with whoever wants to use R for practising epidemiology?", which resulted in this book. I hope this book will be as useful to you too as it has been to me.

Most things in R can be done in multiple ways. Most often, I have shown only one of these many ways to solve any given problem. I do not claim it is the most efficient way to approach that problem. It works. If there is a difference from the result given in the textbook, I have tried to explain the reasons. As you progress in your R journey, you may find better/easier solutions; do share them with me.

I was introduced to regression analysis by Dr. Melissa Rolfes, who was an EIS officer at CDC at that time. I thank her for showing me this new path. I would not have learned R if not for the thousands of people who maintain R, contribute R packages and provide free resources to learn R. I am indebted to them all.

I am grateful to Professor Mark Woodward, who was very encouraging when I approached him with the book. I thank Rob Calver, Senior Publisher – Mathematics, Statistics, and Physics – Chapman & Hall/CRC Press, for making this book a reality. I am indebted to the (unknown) early reviewers who reviewed the first draft and gave valuable suggestions to improve the book. I am grateful to Sherry Thomas, CRC Press Senior Editorial Assistant, and Shashi Kumar, Production Controller, KnowledgeWorks Global Ltd, for their support while preparing the manuscript. I acknowledge with gratitude the assistance provided by my daughter, Manjari, while checking the results against that in the textbook. I am grateful to the free service provided by https://www.wordclouds.com/ that was used to generate the word cloud used for the cover art.

Dr Ajith R

Kerala, India

Preliminaries

This book is a companion to *Epidemiology: Study Design and Data Analysis*, written with the goal to enable readers to practice the concepts discussed in that book using R. This book doesn't teach epidemiology. Neither is this book an R tutorial in the usual sense. This book teaches you to use R as a tool to practice epidemiology. Towards this goal, this book reworks the examples and concepts discussed in *Epidemiology: Study Design and Data Analysis* (which I will refer to as the textbook) using R. The R code is provided along with an explanation of it. I am sure that that will help develop a pretty decent level of R skill; just that it is not the primary aim.

Requirements

Thus, you need to have a copy of *Epidemiology: Study Design and Data Analysis*, without which this book will be of no use to you. You also need access to a computer with R installed on it. You may install the latest version of R from https://cran.r-project.org, following the instructions for your computer's operating system. It is also recommended to install RStudio from https://posit.co/download/rstudio-desktop/.

Organisation of the book

This book uses the same chapter headings as in the textbook. Under each chapter we discuss an example, figure, table or section of the textbook. The beginning of each such part is marked with a horizontal line that ends with the name of that part and the starting page number from the textbook.

In each part, R code to tackle the problem at hand is introduced in small portions. An explanation follows each portion. Once the code is introduced completely, its result is shown. If there is any difference from the textbook, it is highlighted.

At the end of the chapters, there is a Recap section that lists out the commands and concepts introduced in the chapter.

How to use this book

I suggest you to read a section of the textbook (after a quick overview reading of the entire chapter if you are like me), repeating it as many times as needed to understand it completely. Then switch to this book to the section, example or figure discussed in that section and read. Once you read it, type (TYPE!) the code into the R console (of RStudio) of your computer and execute it by pressing the ENTER key. There will be errors, typically typing mistakes. Try to understand what the error message is telling; compare the code you typed with that given in this book, correct and rerun. R is case-sensitive. Many a times the error is a case change.

There are no sections that you may skip. You need to go in sequence as almost all sections depend on what was discussed in earlier sections. Before leaving a chapter, confirm that the concepts and commands listed in "recap" are not unfamiliar to you.

Typographic convention

There is prose and code in this book. The code is given in monospaced font as below

```
mock_code <- seq(1,10)
mock_code
```

The numerical results of the commands are shown as

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

In the prose, we will be referring to the code. We will use text typeset like `seq` to indicate portions of the code that are to be written exactly. The portions typeset like *mock_code* indicate the portions of the code that may well have been another word of our choice. There are a few words typeset like **concepts**. These are words used with a distinctive meaning in R, the meaning of which should be clear from the paragraph where such words are seen.

Groups of R commands are available as add-on modules called packages or libraries. We will be using many packages in the course of this book. Occasionally more than one package adopts the same name for one of their commands. While using the command name suffices most of the time, when there is conflict across packages for the same command name, the right command is specified by prefixing the package name separating it from the command name using two colons `::`. At the end of each chapter, I provide a recap of the commands used in that chapter. There and in the command index, I refer to commands as `packagename::commandname`, though in the body of the chapter I would have used only `commandname`. This is only to inform you of the package to which the command I discussed belongs. That extra bit of information will help you to search information regarding a specific command and when you encounter a conflict in names.

Let us start!

1

Fundamental issues

The first chapter of our textbook introduces us to epidemiology. There aren't any examples in this chapter that demonstrate calculations relevant to epidemiology. So, here we will learn some basic concepts related to R.

R is a software tool to interact with data to make sense of it. It is thus essential for R to have a system to represent data and a way to interact with it. We interact with R by issuing commands at the R prompt displayed in the R console that we are presented with when we start R either directly or through RStudio. R can also accept commands non-interactively through a **script** file. There also is the medium of an **R markdown** file through which we can issue R commands. Typically, we will be using the command prompt to interact with the data, save the useful commands in a script or markdown file to generate final reports at a later stage. Throughout this book we will use the console. In the final chapter, we will see the relevance of R markdown files.

R organises data into different data objects based on what type of data it is. While there are many data objects, the majority of the time we are concerned only with **dataframes** and **vectors**. When we import data from external files of various formats, we usually end up with dataframes, which are collections of vectors of same length. Vectors can exist by themselves too. There are different types of vectors. When we issue commands, some commands accept a dataframe and some require a specific type of vector. Many commands accept different varieties of data objects. Mismatch between what the command expects and what we provide will result in errors. Some commands which manipulate a specific type of vector will accept a different type, doing the conversion automatically. This may confuse those already familiar with other data analysis programs. We start with vectors.

1.1 Vectors

A vector is a collection of similar type (**mode**) of data items. R recognises different types – numeric, character and logical being the most important. What R can do with data depends on its mode. For example, you can do arithmetic operations on data of numeric type but not on character type data. R does allow explicit conversion between different modes where possible. R also converts data to a different mode without explicit instructions. This **implicit type conversion** is useful most of the time but may be the source of unexpected results.

Let us use table 1.1 as an example. It displays the number of cases and controls in different tobacco consumption categories for either sex. The percentages that are calculated from the raw numbers are shown in parentheses. We can tell R the number of male cases in each category as

DOI: 10.1201/9781003589563-1

```
c(2,33,250,196,136,32)
```

The `c()` is the command that tells R that we are typing a vector. The elements of the vector are separated by commas. The parentheses are mandatory. What does R do in response to our command?

```
[1]   2  33 250 196 136  32
```

R simply prints the vector on the screen. It appends "[1]" in front of the values, removes the command name "c", the parentheses and the commas. The "[1]" is the number of the element with which the printed line started. You will appreciate the value of this piece of information when the vector is long enough to be printed in more than one line.

1.1.1 Assignment

There isn't much use in typing some data and seeing it displayed immediately. We need to store the data (for at least the duration of our interaction with R). We do this by assigning the vector to a **variable**. In simple terms, we can think of variable as a name for the data. So, the name that we select should help us remember what the data is.

```
c(2,33,250,196,136,32) -> case_m
```

Here, *case_m* is the name we selected for the variable. R allows almost all sensible combinations as names. However, it is recommended to use only the alphabets, digits, underscore and period. R is case-sensitive. We generally use different cases, underscore or period to separate the different parts of the name we choose. If we use space inside a variable's name, R, understandably, considers it as different words. We are allowed to use space inside a variable name if we quote the name using back ticks. But, stick to underscores, period or case changes. The `->` is the **assignment operator** which tells R to assign the name we specified to the vector we chose. R also allows `=` and `<-` as assignment operators. If we use `<-`, the variable name should be on its left side and the vector on its right. In this text we will be using both `->` and `<-` for assignment. We will be using `=` as well, but not for assigning values to variable names.

What does our assignment command print on screen? Nothing! No news is good news. We asked R to assign our vector to the variable. R did just that. There was no error; hence nothing to print on screen.

What good is our assignment command? After the assignment command is executed, the entire vector is available if we type *case_m*. Let us check.

```
case_m
```

```
[1]   2  33 250 196 136  32
```

There are four more columns of data in table 1.1. Let us enter them too.

```
c(27,55,293,190,71,13) -> control_m
c(19,7,19,9,6,0) -> case_f
c(32,12,10,6,0,0) -> control_f
```

What about the row labels? We can enter those too as a vector.

```
c("Never-smokers", "1-4", "5-14", "15-24", "25-49",
  "50 or mor") -> tobacco
```

In contrast to the vectors we have entered already, *tobacco* is a character vector. The individual elements of the vector are enclosed within quotes to indicate so. If you omit the quotes, R will try to look for a variable by the name indicated by the word that was not quoted, and this will result in an error.

We may change the value of a variable by issuing a new assignment command with the updated values. Thus, if we notice the missing "e" in *tobacco*, we can correct it now by issuing

```
c("Never-smokers", "1-4", "5-14", "15-24", "25-49",
  "50 or more") -> tobacco
```

Variables exist for only a session. If you want to save them for a future session, you should save your workspace information when prompted at the time of closing the current R session. Most often, it is not needed to save the workspace. Saving the codes in a text file, either script or a markdown, would be preferable. Executing them to rerun all the code that we saved takes only one mouse click. We will see more of markdown files in the last chapter.

Though not required most often, we may need to remove R objects already loaded from memory if they are huge and affecting performance. To do this, we use the command `rm` with the object to remove as its argument. For example, to remove *tobacco*, we would issue

```
rm(tobacco)
```

In the console, we can recall a previous command using arrow keys. Press the up arrow key until you find the command we issued to create *tobacco* and execute it again to bring back the variable (We need it in the next step.)

1.2 Dataframes

Now, we have all the relevant raw data from table 1.1. We are not going to type the calculated percentages or the totals. But, can't we have all these vectors joined together as a table? R holds tabular data in what are called **dataframes**. They are essentially vectors of equal length held together. The vectors correspond to the columns of the tables.

```
cbind.data.frame(tobacco, case_m, control_m, case_f,
                 control_f) -> table_1_1
```

The command we use to join together existing vectors as a dataframe is `cbind.data.frame`. The vectors that we want it to join together are provided to it as **parameters** inside parentheses. The resulting dataframe is assigned to the variable *table_1_1* and hence is not printed on screen. If we type *table_1_1* and press enter, we will see the data frame printed.

```
        tobacco case_m control_m case_f control_f
1 Never-smokers      2        27     19        32
2           1-4     33        55      7        12
3          5-14    250       293     19        10
4         15-24    196       190      9         6
5         25-49    136        71      6         0
6    50 or more     32        13      0         0
```

The table printed has the variable names as the column heading and each row is serially numbered. Note that we can pretty the table when we want to print it which we will learn later.

1.3 Functions

The R commands are functions, and we use both the words interchangeably. The functions we use may accept one, more or no values. We call those values arguments or parameters. There is a difference between the words argument and parameter. Parameter is the name used in the code of the function, and argument is the value we pass when we call the function. However, I use the words interchangeably. How does R assign values to the different parameters? In the definition of the command, all parameters have names. When we call a command, we may give the name of the parameter, an equal to sign and then the value we want to assign to that parameter and separate each of these name-value pairs by commas. We may choose not to provide the name of the parameter but only a comma-separated list of values. In such case, R assigns the values by position. That is, the first value is taken as that of the first parameter in the command definition, the second as that of the second and so on. Many a time, commands have a mandatory first parameter and many optional parameters. We will pass the mandatory argument without a name as the first argument and the optional parameters whose values we want to change from the default are supplied as name-value pairs. We won't use `=` as an assignment operator as it is also used to mark name-value pairs when we call functions. If it is used in both ways, there is the possibility of confusion of the two roles.

Commands are collected together into **packages**. Some packages come with the base installation of R. Many are add-ons. Packages are also called libraries. Packages number in thousands and new ones get added with time. Add-on packages are added with the command `install.packages`. While installation is a one-time process, we need to load it to memory whenever we want to use it. This is accomplished by the command `library`. We will visit these commands later in this book when we have to use an add-on package. A good starting point to know about the common packages is the CRAN task view.

1.4 Importing

What if we have large tables saved as separate files formatted as comma-separated values, tab separated values, excel files etc.? Most often we will be importing data from separate files rather than typing it ourselves. R is capable of importing data stored in different formats. As an example, let us first save the data of table 1.3 as a comma-separated file. Open a spreadsheet program and type the data of table 1.3 in it and save it as a csv file. Use only one row of headers when you type, avoid the figures in parentheses and the totals. Give it the name *test.csv* and save it in your current working folder. You can find your current working folder by executing `getwd()` at your R command prompt.

```
table_1.3 <- read.csv("test.csv")
```

The read.csv command reads the csv file specified by its first parameter. You may specify the full path of the file. Here, just the file name was sufficient as we saved the file in our working directory. There are other commands for other formats – `read.csv2`, `read.delim`, `read.delim2` etc. You can read about any command by typing `?` followed by the command name (`?read.csv` for example) from the R command prompt. There are many add-on packages with functions to import specific formats. The package `haven` which has functions to import(and export) data from SPSS, Stata and SAS, the `readxl` package to import excel formats are examples.

The `read.csv` command accepts many more parameters in addition to the file name. For example, the parameter named `dec` denotes the character that specify the decimal marker. All these parameters have certain **default values** specified. That is the reason why we don't specify them when we call the command. If we want to change the value of those parameters to something other than the default, we will include them in our command call. For example, if we want to tell R that the character that is used as the decimal marker in our csv file is `,` instead of the usual `.`, we should change our command to `table <- read.csv("test.csv", dec = ",")`. For all the commands that we will be using later, we will discuss only those parameters that are relevant for our problem at hand. If you need more details, use the `?` command to bring up the documentation.

1.5 Other data objects

1.5.1 Matrices and arrays

We discussed vectors and data frames. But, those are not the only data structures in R. Rectangular data can be represented as **matrices**. **Arrays** can be used to store data with more than two dimensions. Matrix is an array with two dimensions. Arrays and matrices are similar to vectors in that the data contained in them are all of the same mode. Table 1.3 can be represented as a matrix. Calculating totals and percentages would be easier than when represented as a data frame. Even then, data frame is more flexible as it allows different type of data to be together. We will discuss matrices when we see them later.

1.5.2 Lists

Another type of data structure is a list. List can have elements of different modes. Also, the length of the elements need not be the same. The data frame that we saw earlier is a list, but with the restriction that the elements should be of the same length.

1.5.3 Dates and factors

The basic vectors can be converted to derived data types. By basic vectors, I mean those vectors types that can be made directly using the `c()` command. The derived vectors are made using commands that accept a basic vector and modify them. Easiest to comprehend would be dates. We write dates in different ways and yet can make sense of them easily. It is not so easy for a computer. We have to tell it the logic we followed to write a date. When we type dates, we should specify them as character vectors in a consistent way. We should import them into R as character vectors. After inspecting the imported data to confirm that the text has been imported correctly, we may change it to one of the data classes R uses to store date. A relevant package to manage dates in R is `lubridate`.

When we have labels of categories as our data, it is possible that they are typed in as codes instead of the actual labels. Indeed, some R functions also require such **factors**. However, whenever possible, use the actual labels and import the data as character vectors. After importing, it can be converted to factors using the command `factor`. The package `forcats` is a package to manipulate factors.

1.6 Rounding

R calculates results to many decimal places. While we needn't care about this while interacting with data, we typically restrict the decimal places when we present results. We use the function `round` to round numbers to a specified number of decimal places. We also have the function `signif` that allows us to round to a specified number of significant digits. Here, we round 1 divided by 30 to two decimal places and two significant digits.

```
1/30
round(1/30,2)
signif(1/30,2)
```

```
[1] 0.03333333
[1] 0.03
[1] 0.033
```

Both the functions accept the number to be rounded and the number of digits to round to. In the chapters that follow, I use rounding only rarely. That is only because I want to show the result given by R without any modifications. You should use rounding when you present your results. We may also set the number of significant digits to print globally using the command `options`. For example, `options(digits = 5)`, will set it to 5 significant digits.

Now that we have an idea of the basic aspects of R, we can proceed to see how R is useful for epidemiology.

1.7 Recap

Let us recap what we learned in this chapter.

1.7.1 Concepts introduced in this chapter

- script
- R markdown
- vectors
- mode
- implicit type conversion
- variable
- assignment
- operator
- parameters
- dataframes
- default values
- packages
- matrices
- arrays
- list
- factor

1.7.2 Commands introduced in this chapter

- base::c
- base::<-
- base::=
- base::->
- base::cbind.data.frame
- base::getwd
- utils::read.csv
- utils::?
- base::round
- base::signif

2

Basic analytical procedures

The second chapter of the textbook introduces us to basic descriptive and analytical techniques. We will be using the standard packages that are installed and loaded by default along with R. We will introduce add-on packages `dplyr`, `readr` and `ggplot2` which are part of `tidyyverse`, `tinytable` and `DescTools`. The command names that we use are intuitive. For graphs, these are names of the graph itself or remind of it. For common statistical tests, these have two parts – an initial part which reminds us of the test, followed by ".test". The arguments that the command expects are varied, with a majority accepting vectors.

2.1 Tables and charts

First, we will look at the chart made using the data in table 2.1.

2.1.1 Bar chart

Figure 2.1 (page 28)

```
c(I = 592,II = 2254,IIIn = 1017,IIIm = 3150,IV = 1253,V = 415) -> table_1_1
```

We have made a vector by name *table_1_1* and assigned to it the data in the column "Number". In contrast to how we created vectors in our previous chapter where only the bare numbers were supplied, we give a name to each value (Note that the names are not quoted). These names will be useful for plotting labels.

```
barplot(table_1_1)
```

To plot the graph, the command `barplot` is called with our vector as its argument. The bar chart drawn by R isn't exactly like in our parent book. In particular, it lacks the axis labels. We can provide more arguments to the command to add the labels.

```
barplot(table_1_1,
        xlab = "Social class",
        ylab = "Frequency")
```

DOI: 10.1201/9781003589563-2

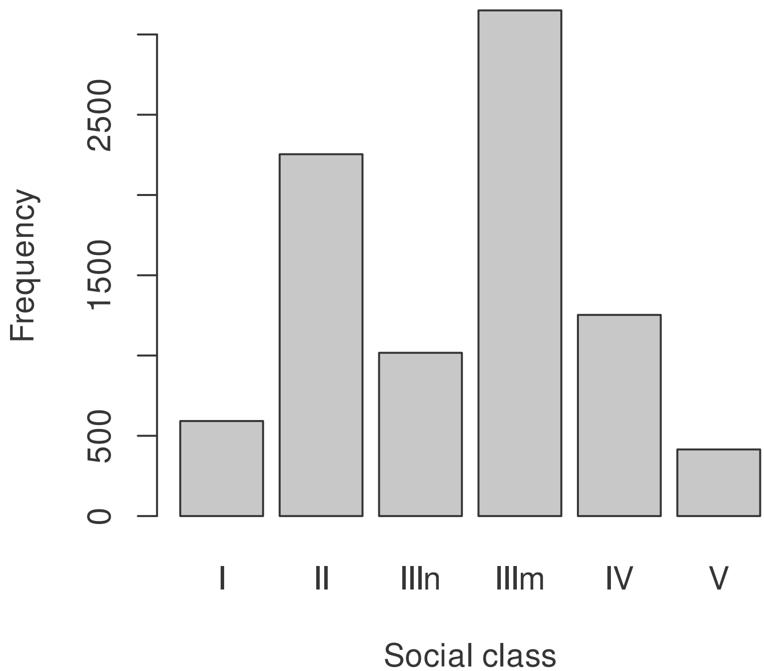


FIGURE 2.1
Replication of figure 2.1

Similarly, we can make the pie chart given in the chapter with the command `pie(table_1_1)`. Now, let us try to chart figure 2.3.

Figure 2.3 (page 29)

```
matrix(c(100,492,382,1872,183,834,668,2482,279,974,109,306),
       ncol = 6) -> table_1_2
rownames(table_1_2) <- c("CHD Yes", "CHD No")
colnames(table_1_2) <- c( "I","II","IIIn","IIIm","IV","V")
```

Here we use the command `matrix` to create a matrix from the unnamed vector holding our data. Matrix is similar to vectors in that all the values stored in it are of the same mode. In contrast to vectors, matrices are (conceptually) rectangular, i.e., the data in them can be thought of as arranged in a rectangular grid. We need to tell the number of rows and columns our matrix contains. This is accomplished by specifying the `ncol` parameter in our example. We may also specify `nrow` parameter. By default, `matrix` command fills the matrix top down column wise. The parameter `byrow` can be specified to ask the matrix to be filled row wise. We have used the command `rownames` with the matrix we created as its argument and assigned to it a character vector to indicate the row headings. Similarly, the command `colnames` sets the column headings. We could have set the row and column names inside the `matrix` command itself by using the `dimnames` parameter which expect a **list** of length two. A **list** is another kind of data object in R. It differs from vectors and matrices in that its components need not be of the same type or length. We will learn more about lists later.

Now, to make the graph.

```
prop.table(table_1_2, 2)*100 -> percent_1_2
percent_1_2
```

```
              I      II    IIIn   IIIm     IV      V
CHD Yes 16.892 16.948 17.994 21.206 22.267 26.265
CHD No  83.108 83.052 82.006 78.794 77.733 73.735
```

We use the command `prop.table`, which returns the proportion of each cell of its first argument, an array (A matrix is an array with two dimensions). The second argument `margin` determines the denominator for calculating proportions – 2 indicating that we want column proportions. If we want row proportions we should specify 1 instead. If we want proportions to be calculated relative to total of all cells, we can leave out the second argument. We multiply the proportions returned by the function with 100, to obtain percentages. Note that when we ask R to multiply with 100, it multiplies each element of the vector with 100. This is an example of **vector arithmetic**.

```
barplot(percent_1_2, horiz = TRUE, xlab = "Percent", ylab = "Social class")
```

When we call `barplot` with the matrix carrying the percentage values as its argument, we get a bar diagram with one bar for each column. The values corresponding to each cell of the columns are stacked one above the other; hence it is a stacked bar chart. Because we calculated column percentages and passed it to `barchart`, each of the stacked bars will be of the same height, i.e., 100%. Specifying the `horiz` argument as `TRUE` makes the graph horizontal.

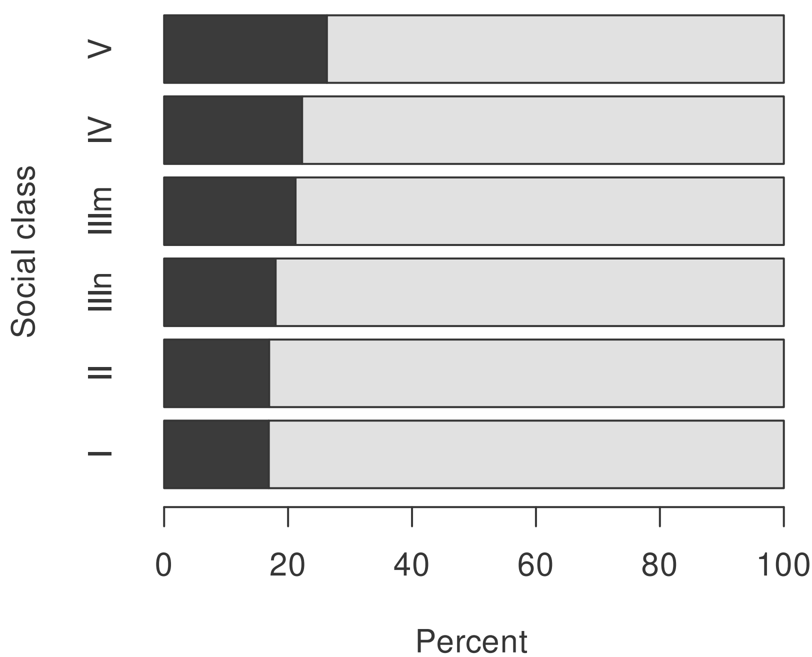


FIGURE 2.2
Replication of figure 2.3

2.2 Inferential techniques for qualitative variables

2.2.1 Chi-square test

Section 2.5.1 (page 33)

Section 2.5 discusses inferential techniques for categorical values. The first test discussed is chi square. We will first prepare the matrix that will be provided as the argument for `chisq.test`, the R command to do chi square test.

```
matrix(c(2241,1400,103,352,424,2599,1551,0,9,2),
       nrow = 2,
       byrow = TRUE) -> tbl_2_8
chisq.test(tbl_2_8)
```

The result displayed by the `chisq.test` shows the value of chi squared, the degrees of freedom as well as the p value of the test.

```
    Pearson's Chi-squared test

data:  tbl_2_8
X-squared = 868, df = 4, p-value <2e-16
```

In reality, the test provides more information than what is shown by default. A common paradigm in R is to store the results of tests in an object to inspect it closely. For example, we can execute `chisq.test(tbl_2_8) -> chi.tbl_2_8` to store our result in *chi.tbl_2_8*. The result returned by `chisq.test` is a list. We can find the structure of an object by passing it as an argument to the command `str`. When we execute `str(chi.tbl_2_8)`, we can see that it contains 9 components, the names of which are self explanatory. If we need to see any one component of the list, we can use the `$` operator. For example, if we want to see the expected values calculated by the command, we can execute `chi.tbl_2_8$expected`. This will display a matrix on the screen which will reveal that the expected values calculated by the command are the same as in our parent text except for rounding error.

```
chisq.test(tbl_2_8) -> chi.tbl_2_8
chi.tbl_2_8$expected
```

```
       [,1]   [,2]  [,3]   [,4]   [,5]
[1,] 2520.1 1536.5 53.63 187.96 221.81
[2,] 2319.9 1414.5 49.37 173.04 204.19
```

The command `chisq.test` accepts other arguments too, one of which is `correct` which needs to be specified as `TRUE` if you want continuity corrections to be applied. You can read more about the command by executing `?chisq.test`.

2.2.2 Proportions – One Sample

Section 2.5.2 (page 36)

We can use `binom.test` command to calculate a proportion and its confidence interval.

```
binom.test(1562, 4161)
```

The `binom.test` is designed to test whether the proportion calculated from the numbers provided is different from 0.5. Along with that, it also provides a confidence interval of the proportion calculated. If we need to use only the confidence interval, we can specify `binom.test(1562, 4161)$conf.int`. Note that the method for calculating the confidence interval is different from the approximate method used in our parent text and may differ from it. We can specify our choice of confidence interval through the parameter `conf.level` as a number between 0 and 1.

```

    Exact binomial test

data:  1562 and 4161
number of successes = 1562, number of trials = 4161, p-value <2e-16
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.36065 0.39031
sample estimates:
probability of success
               0.37539
```

After rounding, the confidence interval we get is not different from that given in the textbook.

To test if the proportion is different from, say 0.39, we need to specify the `p` argument as

```
binom.test(1562, 4161, p = 0.39)
```

The p value we obtain is similar to that in our parent text, though the method used here is different.

```

    Exact binomial test

data:  1562 and 4161
number of successes = 1562, number of trials = 4161, p-value = 0.054
alternative hypothesis: true probability of success is not equal to 0.39
95 percent confidence interval:
 0.36065 0.39031
sample estimates:
probability of success
               0.37539
```

To test if the proportion is less than 0.39, i.e. for a one sided test, we need to specify the `alternative` argument as

```
binom.test(1562, 4161, p = 0.39, alternative = "less")
```

We get the same p value as in our parent text.

```
    Exact binomial test

data:  1562 and 4161
number of successes = 1562, number of trials = 4161, p-value = 0.027
alternative hypothesis: true probability of success is less than 0.39
95 percent confidence interval:
 0.00000 0.38792
sample estimates:
probability of success
               0.37539
```

2.2.3 Proportions – Two Sample

Section 2.5.3 (page 40)

Two sample proportions are handled by `prop.test`. First, we will enter the data.

```
matrix(c(2279,1562, 2241,2599),
       nrow = 2,
       dimnames = list(c("Smoker","Nonsmoker"),
                        c("Male","Female"))) -> tbl_2_9
prop.test(tbl_2_9, correct = FALSE)
```

Note that the order in which we specify the data is different from the textbook. While the p value and Chi squared value would be the same if we specify the data in the same order as given in the textbook, the difference between proportions would have a different numerical sign. The test result includes a confidence interval for the difference between the proportions as well as the p value of a significance test. The significance test used employs chi squared rather than the normal approximation. Hence the value of the statistic is expected to be the square of the textbook value, though the p value is similar. As with `binom.test`, we can store result in a variable and inspect it. Similar to `binom.test` we may specify the `conf.level` we require as well as the `alternative` that we want tested. We have specified `correct = FALSE` to say that we don't want to apply continuity correction.

```
    2-sample test for equality of proportions without continuity correction

data:  tbl_2_9
X-squared = 146, df = 1, p-value <2e-16
alternative hypothesis: two.sided
```

```
95 percent confidence interval:
 0.10810 0.14952
sample estimates:
 prop 1  prop 2
0.50420 0.37539
```

The test statistic we get is 145.726, the square of 12.07 given in the textbook.

2.3 Descriptive techniques for quantitative variables

Example 2.3 (page 43)

Section 2.6 deals with descriptive techniques for quantitative variables. Example 2.3 requires a variable to be declared

```
c(15,3,9,3,14, 20,7,8,11) -> xmpl_2_3
```

We calculate the minimum using `min(xmpl_2_3)` and maximum using `max(xmpl_2_3)`. We can obtain the quartiles using the command `quantile(xmpl_2_3)`. By default, the `quantile` command outputs the five-number summary – minimum, maximum and median in addition to the first and third quartiles. The `quantile` command provides any quantile value for the probabilities we specify. For this we need to provide a numeric vector with values between zero and one to the `probs` argument. If there are any unknown values in our data vector, we need to specify `na.rm = TRUE`, to ignore those values in calculations.

```
min(xmpl_2_3)
max(xmpl_2_3)
quantile(xmpl_2_3)
```

```
[1] 3
[1] 20
  0%  25%  50%  75% 100%
   3    7    9   14   20
```

Section 2.6.1 (page 43)

Before we find out the five-number summary of the cholesterol data given in table 2.10, we should import it into R.

While base R provides us with a lot of functions, the abilities of R are extended by different packages. Some of the commonly used packages are available as a meta package named `tidyverse`. We will use that package from now on. Before using the package, we need to install the package. The command to do that is `install.packages("tidyverse")`. If your internet connection is okay, the package will be downloaded and installed. Though the package is installed, it is not loaded for use. To use the package during your R session, you need to issue the command `library(tidyverse)`. Note that installation is a one-time step, while library(tidyverse) needs to be issued during each R session. Also, note that the string used for installation is enclosed in quotes, while in the library command, it is not. Once loaded, all the commands in the package are available including read_table that we use to import the data given in table 2.10.

The command that we use to import the data would be

```
library(tidyverse)
read_table("./K11828 supplements/Datasets/Table 2.10.DAT",
           col_names=c("cholesterol","diastolic_bp","systolic_bp",
                       "alcohol","cigs","co","cotinine", "chd"),
           col_types = cols( cholesterol = col_number(),
                             diastolic_bp = col_number(),
                             systolic_bp =col_number(),
                             alcohol = col_number(),
                             cigs = col_number(),
                             co = col_number(),
                             cotinine = col_number(),
                             chd = col_factor())) -> tbl_2_10
```

The command read_table is used to import textual data where the columns are separated by space. It requires a string which specifies the path to the file we want to import. If the path to the data file you downloaded from the textbook's website is different from what I have used, you need to change the argument accordingly. As our data doesn't have a header row in our data file, we specify the argument col_names as a vector of strings based on the description of the data file given in the textbook. The argument col_types is called with a cols() definition. It tells how R should treat each column of data. col_number tells that the data in the column for which it is specified should be treated as a number. For us, all columns are numbers except the column chd which is a **factor**. Factor stands for those values which are actually codes for something else. For us, the value 1 in the column *chd* stands for "yes" and the value 2 for "no". Though read_table can guess correctly the type of data in our data file, it is better to provide the col_types argument to avoid surprises. The tidyverse uses a type of dataframe called **tibble** and read_table is no exception. However, there aren't any particular difference that we will have to keep in mind. I will be using the word tibble only in situations where the differences from dataframe is significant to note.

Now that we have our data in the variable *tbl_2_10*, finding the five-number summary for the cholesterol data is easy

```
quantile(tbl_2_10$cholesterol, na.rm = TRUE)
```

```
   0%   25%   50%   75%  100%
4.350 5.750 6.270 6.775 7.860
```

If we want just the range, we can use range which returns a numeric vector containing the minimum and maximum. We can then find their difference.

```
range(tbl_2_10$cholesterol) -> range_chol
range_chol[2] - range_chol[1]
```

The result given by `range` is a vector with two elements. To find their difference, we subtract the first element from the second. The square brackets that follow the name of the vector and the number given inside it is the way we refer to an element of a vector and is called **subsetting**.

```
[1] 3.51
```

To find the interquartile range, we can use the result of `quantile` or the function `IQR`

```
quantile(tbl_2_10$cholesterol, na.rm = TRUE) -> five_chol
five_chol[4] - five_chol[2]
IQR(tbl_2_10$cholesterol, na.rm = TRUE)
```

```
  75%
1.025
[1] 1.025
```

The semi interquartile range is easily calculated by `IQR(tbl_2_10$cholesterol)/2`. Quartile symmetry can be calculated using the values given by `quantile`. As an example: `(five_chol[2] - five_chol[1])`.

I hope that you have noticed that R uses the usual symbols to do basic arithmetic operations `-`, `+`, `/` and `*`.

Section 2.6.3 (page 48)

We can calculate the mean of the cholesterol data using

```
mean(tbl_2_10$cholesterol)
```

```
[1] 6.2866
```

The variance is calculated using

```
var(tbl_2_10$cholesterol)
```

```
[1] 0.57299
```

The standard deviation is calculated using

```
sd(tbl_2_10$cholesterol)
```

```
[1] 0.75696
```

All the three functions require the argument `na.rm=TRUE` if any values of the provided numeric vector is not available.

Section 2.6.4 (page 50)

While there is no ready available function to calculate coefficient of variation, we can calculate it from the mean and standard deviation provided by the previous functions.

```
sd(tbl_2_10$cholesterol) * 100 / mean(tbl_2_10$cholesterol)
```

```
[1] 12.041
```

Similarly, we can calculate the standard error by

```
sd(tbl_2_10$cholesterol) / sqrt(length(tbl_2_10$cholesterol))
```

```
[1] 0.10705
```

This formula doesn't take care of `NA` values. A better alternative would be

```
sd(tbl_2_10$cholesterol, na.rm = TRUE) /
  sqrt(sum(is.na(tbl_2_10$cholesterol) == FALSE))
```

The `na.rm` option to the command `sd` asks the command to remove `NA` values before calculating sd. The command `length` doesn't have such an option. So, we use the command `is.na` which evaluates each element of its argument and reports `TRUE` if it is `NA` and `FALSE` otherwise. We then select only those which are equal to `FALSE`, i.e. those that are not NA. We use the equality comparison operator `==` for this purpose. Notice that we check against the value `FALSE`, without quotes as we are asking if the value is the **boolean** (**logical**) value `FALSE`, not if it is the string *"FALSE"*, in which case it would have been provided inside quotes. We `sum` the elements that satisfy the result. R implicitly converts the TRUE values to 1 and FALSE to 0. The net result is that we get the length of the supplied vector where the value is not `NA`. We use `sqrt` to get the square root of this number.

We can actually avoid the explicit comparison. What `is.na` returns is a logical vector. Type and see the result of `is.na(tbl_2_10$cholesterol)`. It is a vector composed of the logical values `TRUE` and `FALSE`. The command `sum` when given a logical vector will implicitly convert all `TRUE`s to 1 and add. But, we want to sum those that are not `NA`s; `is.na` reports `TRUE` only if the value is `NA`. We can invert (i.e. turn `TRUE` to `FALSE` and vice-versa) the logical values by using the negation operator `!`. Thus our command should be

```
sd(tbl_2_10$cholesterol, na.rm = TRUE) /
  sqrt(sum(!is.na(tbl_2_10$cholesterol)))
```

Figure 2.5 (page 45)

Section 2.6.1 also deals with graphical summary of quantitative variables. We can construct a box plot of the cholesterol data with

```
boxplot(tbl_2_10$cholesterol,
        horizontal = TRUE,
        xlab="Serum total cholesterol (mmol/l)")
```

The `horizontal` argument was provided to change the default orientation of the graph. The boxplot of alcohol shown in figure 2.6, can be made similarly.

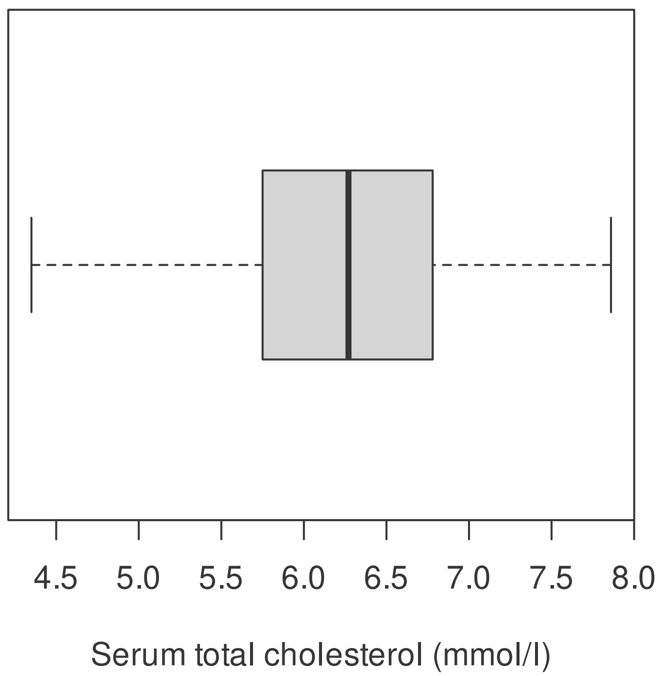


FIGURE 2.3
Replication of figure 2.5

Figure 2.7 (page 47)

To produce figure 2.7, where two boxplots are produced for the two categories of *chd*, we use

```
boxplot(cholesterol ~ chd,
        data = tbl_2_10,
        horizontal = TRUE,
```

```
        xlab = "Serum total cholesterol (mmol/l)",
        ylab = NA,
        names = c("No CHD","CHD"))
```

The first argument is called a **formula**. It informs R that we want *cholesterol* values to be used for constructing the boxplot, but split by *chd* which appears after the tilde `~`. The `data` argument tells R that the variables mentioned in the formula are part of the dataframe *tbl_2_10*. The argument `horizontal = TRUE` makes the boxplots horizontal, `xlab` specifies the x-axis label, `ylab = NA` prevents y-axis label and `names` determine the labels against each box.

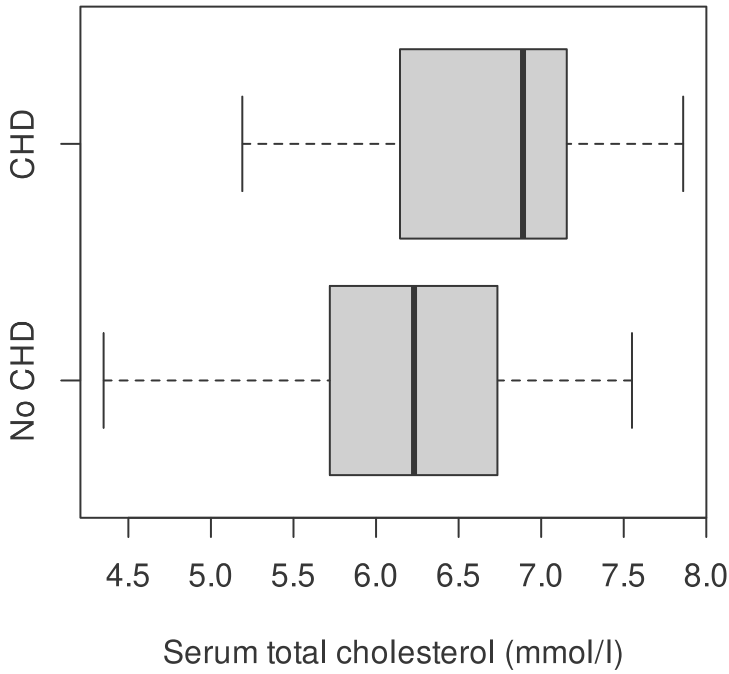


FIGURE 2.4
Replication of figure 2.7

Table 2.12 (page 52)

How can we prepare a table like 2.12 of the textbook?

```
summarise(tbl_2_10, Mean =  mean(alcohol))
```

The command `summarise` is provided by `dplyr`, part of `tidyverse`. It accepts a data frame and a function. It will summarise the column that we provide as the argument to the function using the function given as the second argument. Here, we want the function `mean` to summarise the column *alcohol* of the dataframe *tbl_2_10* and assign it the name *Mean*.

```
# A tibble: 1 x 1
```

```
   Mean
  <dbl>
1  26.8
```

We can request multiple summaries of the same column or of different columns.

```
summarise(tbl_2_10,
          Mean.alcohol =  mean(alcohol),
          SD.alcohol = sd(alcohol),
          Mean.Cholesterol = mean(cholesterol))
```

```
# A tibble: 1 x 3
  Mean.alcohol SD.alcohol Mean.Cholesterol
         <dbl>      <dbl>            <dbl>
1         26.8       27.8             6.29
```

Notice that we have different names for the summary columns. To get the same summary for all the columns requires some copy pasting or ...

```
summarise(tbl_2_10, across(!chd, ~  mean(.x, na.rm = TRUE)))
```

Here, instead of providing a list of comma-separated name function pairs, we provide another function `across`. The first argument to `across` is a way of specifying the required columns that is called **tidy select**. We may specify *col_1* `:` *col_3* to mean all columns from that named *col_1* till the one named *col_3*, `c(` *col_1, col_3* `)` to mean *col_1* and *col_3*, `!`*col_1* to mean all columns except *col_1* or `everything()`. There are many more helper functions like `starts_with()`. In our example, we select all columns except *chd*. After the first argument, we provide the function that needs to be applied. If we are not providing any arguments to the function, we need only provide its name. Here, we want to pass `na.rm = TRUE` to `mean`. So, we provide a one sided function, with only the right-hand side. After the tilde, the function name is provided and within the parentheses that follow, the arguments for the function. The first argument is a special one `.x`, which stands for the column that will be passed to the function.

```
# A tibble: 1 x 7
  cholesterol diastolic_bp systolic_bp alcohol  cigs    co cotinine
        <dbl>        <dbl>       <dbl>   <dbl> <dbl> <dbl>    <dbl>
1        6.29         84.6        131.    26.8  8.74  12.8     139.
```

The `across` function can handle multiple functions too. However, what it returns is a dataframe with a single row and many many columns. So, to get to table 2.12, we follow a different path.

```
bind_rows(summarise(tbl_2_10, across(!chd, ~ mean(.x, na.rm = TRUE))),
          summarise(tbl_2_10, across(!chd, ~ sd(.x, na.rm = TRUE))),
          reframe(tbl_2_10, across(!chd,
```

```
                                    ~ quantile(.x,
                                            probs = c(0.5,0.25,0.75,0,1),
                                            na.rm = TRUE)))) ->
  tbl_2_10_summary
```

The command **reframe** is very similar to **summarise**. The difference is that **summarise** handles summaries that are a single value (like those provided by **mean**, **sd**...) while **reframe** handles summaries that return multiple values (like those provided by **quantile**). The **bind_rows** joins together multiple rows of data to make one dataframe. Thus we get one row for each call of **summarise** and as many rows as the value returned by **reframe**. We save this summary table for future use. We will come back to table 2.12 a little while later.

```
# A tibble: 7 x 7
  cholesterol diastolic_bp systolic_bp alcohol  cigs    co cotinine
        <dbl>        <dbl>       <dbl>   <dbl> <dbl> <dbl>    <dbl>
1       6.29          84.6       131.     26.8  8.74  12.8     139.
2       0.757         10.5        14.3    27.8 12.5   14.9     177.
3       6.27          82.5       132.     19.1  0      6         7
4       5.75          76.2       123       6.45 0      3         0
5       6.78          92         139      36.5 20     16       284.
6       4.35          65         104       0    0      1         0
7       7.86         109         183     120.  40     57       554
```

Figure 2.8 (page 53)

Now, we will turn our attention to figure 2.8. We need to change our data as described in the textbook to make this graph. A function named **mutate** of the **dplyr** package will help us accomplish this.

```
mutate(tbl_2_10,
       talcohol = (alcohol - median(alcohol))/ IQR(alcohol)) -> tbl_2_10t
```

Similar to **summarise**, the first argument is the dataframe that we want to manipulate. Next, we specify the name of the new column that we want and how its value should be calculated. In our example, we want a new column named *talcohol* (short for transformed alcohol) defined as the value of each observation minus the median for the column, which is then divided by the inter quartile range. We store this transformed table with the name *tbl_2_10_t*. We may repeat the same technique to add more name calculation pairs to get the transformed values for all columns. Instead, we will use **across**.

```
mutate(tbl_2_10, across(!chd, rescale)) -> tbl_2_10t
```

Here, we are asking to mutate all columns except *chd* using the function *rescale*. But, what is *rescale*? It is a custom function that we define as given below.

```
rescale <- function(x) {(x-median(x))/IQR(x)}
```

In contrast to the earlier instance where we defined a function, here we are assigning the name *rescale* to our function because we want to be able to call it from else where. Note that we could have used any other name. The keyword `function` tells R that we are defining a function. The x inside the `()` is the variable name that will represent inside the function body, the argument we pass to the function. You may note the similarity between our first `mutate` example and the code inside the function body. Note that though I have given the function definition after calling it inside `across`, we should define the function before calling it. Otherwise, R will complain that the object *rescale* is not found.

The actual command to make the graph is

```
boxplot(select(tbl_2_10t,
               c(diastolic_bp,systolic_bp,cigs,co,cotinine)),
        horizontal = TRUE,
        ann = FALSE,
        names = c("dBP", "sBP", "Cigs", "CO", "Cotinine"))
```

The first argument to `boxplot` is `select`, another function provided by `dplyr`. From the data frame that is supplied to `select`, it returns a new dataframe selecting only those columns we specify. It supports tidy select mentioned earlier. Here, we are giving a list of desired columns. The command `boxplot` will create a boxplot for each of the columns in the dataframe. We use `ann = FALSE` to prevent annotation of the axis. Note that though we have shortened the labels given as `names`, `boxplot` does not print them properly. Later, we will learn better ways to create boxplots.

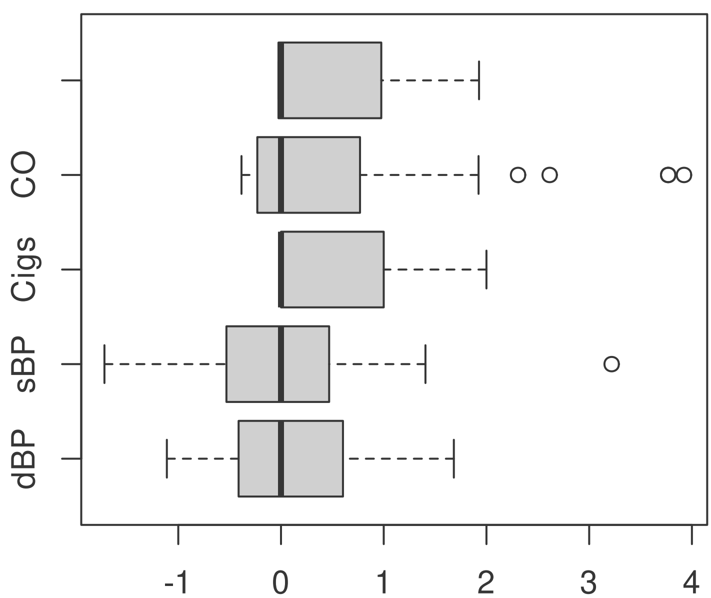


FIGURE 2.5
Replication of figure 2.8

Table 2.12 (page 52)

Now let us turn our attention back to table 2.12. The *tbl_2_10_summary* that we made till now doesn't have all the rows of table 2.12. While the data to calculate the remaining rows are there, it is not arranged properly. The command `mutate` can produce new columns from existing columns, not new rows. We need a way to "rotate" the table we prepared.

```
rownames(tbl_2_10_summary) <- c("mean", "sd", "median",
                                "Q1", "Q3", "min", "max")
as.data.frame(t(tbl_2_10_summary)) -> tbl_2_10_summary
```

The rows of the summary table that we generated don't have names, but numbers as indicated at the beginning of each row when the data frame is printed. While it is of no consequence if we were using the table as such, we need row names so that they become column names when we transpose the table. First of all check the current row names by executing `rownames(tbl_2_10_summary)`. You can confirm that the row names are just a sequence of numbers. Now execute `rownames(tbl_2_10_summary) <- c("mean", "sd", "median", "Q1", "Q3", "min", "max")`. This will assign the names in the character vector on the right-hand side to the row names of *tbl_2_10_summary*. After executing this command, check the row names again. You will see that the row names are the strings we supplied. Confirm that the names conform to the data in the row. If needed, change the order of row names and reassign them. Now, we will use the command `t(tbl_2_10_summary) -> tbl_2_10_summary` to transpose the dataframe, i.e. to change the columns to rows and rows to columns. R will issue a warning that you can safely ignore. Note that by assigning the transposed data frame to the name of the original dataframe, we overwrite and loose the original dataframe.

However, what `t` returns is not a dataframe. It is a matrix. We need to convert it back to a data frame. The command we use to do that is `as.data.frame`. We can actually combine the previous step with this as `as.data.frame(t(tbl_2_10_summary)) -> tbl_2_10_summary`. Now we are in a position to manipulate the table to calculate the missing data

```
nrow(tbl_2_10) -> num
mutate(tbl_2_10_summary,
       serror =  sd / sqrt(num),
       range = max -min,
       iqr = Q3-Q1,
       cv = sd * 100/ mean) -> tbl_2_10_summary
select(tbl_2_10_summary,
       c(mean,median,sd,serror,Q1,
         Q3,iqr,min,max,range,cv)) -> tbl_2_10_summary

as.data.frame(t(tbl_2_10_summary)) -> tbl_2_10_summary
```

```
       cholesterol diastolic_bp systolic_bp  alcohol    cigs      co cotinine
mean       6.28660      84.5600    131.2600  26.7640  8.7400 12.7800   139.46
median     6.27000      82.5000    131.5000  19.1000  0.0000  6.0000     7.00
```

```
sd          0.75696      10.4533     14.2812  27.7533  12.4619  14.9466   177.27
serror      0.10705       1.4783      2.0197   3.9249   1.7624   2.1138    25.07
Q1          5.75000      76.2500    123.0000   6.4500   0.0000   3.0000     0.00
Q3          6.77500      92.0000    139.0000  36.5000  20.0000  16.0000   283.75
iqr         1.02500      15.7500     16.0000  30.0500  20.0000  13.0000   283.75
min         4.35000      65.0000    104.0000   0.0000   0.0000   1.0000     0.00
max         7.86000     109.0000    183.0000 119.6000  40.0000  57.0000   554.00
range       3.51000      44.0000     79.0000 119.6000  40.0000  56.0000   554.00
cv         12.04087      12.3620     10.8801 103.6963 142.5844 116.9527   127.11
```

Notice that the quartiles and IQR for alcohol in our table doesn't match the values in our textbook (6.45 is our Q1 against the 6.2 in the textbook: Q3 is 36.5 against 38.3; IQR is 30.05 against 32.1). The reason for this discrepancy is the way quantiles are calculated. We can change the way quantiles are calculated by specifying the `type` option for the `quantile` command. As the textbook doesn't talk about how to choose from the various ways to calculate quantiles, it is up to you to read about them. We haven't calculated skewness in our table. This can be achieved by defining a custom function and using it while building the summary table. You will also find ready made functions in some packages. We use the `select` command to rearrange the columns in the data frame to align with the one in the textbook.

We will try to pretty the table now. For that we use a package named `tinytable`. Remember to install it and load it before using.

```
library(tinytable)
bind_cols(c("Mean","Median","Std deviation","Std error","Q1",
            "Q2","IQR","Minimum","Maximum","Range","CV"),
          tbl_2_10_summary) |>
  setNames(c("Summary statistic",
           "{Serum total\\\\cholesterol\\\\(mmol/l)}",
           "{Diastolic\\\\blood pressure\\\\(mmHg)}",
           "{Systolic\\\\blood pressure\\\\(mmHg)}",
           "{Alcohol\\\\(g/day)}",
           "{Cigarettes\\\\(no./day)}",
           "{Carbon\\\\monoxide\\\\(ppm)}",
           "{Cotinine\\\\(ng/ml)}")) |>
  tt(caption = "Replication of table 2.12",
     notes = "Note: IQR = inter quartile range;
              CV = coefficient of variation.
              Sample size, n = 50.",
      width = 8)  |>
  format_tt(j = c(3,6,7), digits = 2) |>
  format_tt(j = c(2,4,5,8), digits = 3) |>
  format_tt(i = 11,j = 2:8, sprintf = "%3.0f%%", escape = TRUE) |>
  style_tt(j = 2:8,
           align = "d") |>
  style_tt(i = 0,
           align = "c",
           alignv = "b") |>
```

```
theme_tt("placement",
         latex_float = "H")
```

We use what are called **pipes** in our code. The pipe that we use is `|>`. Earlier, R did not support pipes natively. A pipe was provided by `magrittr` package. However, now that R supports pipe natively, we use that. The pipe passes the result of the left-hand side expression as the first argument of right-hand side. Though not all functions accept its argument from a pipe, for those functions which do accept its argument from a pipe it is a very convenient way to make incremental changes as we do in the above example.

`tinytable` is used to pretty tables for the purpose of presentation. The package provides many functions to pretty tables while allowing great flexibility in usage. The function `tt` creates a tinytable from its argument, a dataframe. Here, we piped the result of `bind_cols` to `setNames` to change names for printing. The `bind_cols` was used to bind together a column of row labels with the dataframe *tbl_2_10_summary* that we want to pretty. `setNames` returns the dataframe after assigning the new names, which we pass to `tt`. We enclose the new names in `{}` and add `\\\\` at places where we want line breaks to let know LaTeX how we want them to be printed. We specify the arguments `caption` which specifies the table caption, `notes` which specifies the footnote of the table and `width` which specifies the print width of the table. While the output of `tt` itself would be pretty clean, we want fine control over the appearance.

So, we pipe the result of `tt` to `format_tt` thrice, one after the other. This function is used to format many aspects of the table contents. We can specify which rows and columns the formatting should apply to by using the `i` and `j` arguments. We use different decimal places for different columns – two digits for the third, sixth and seventh column and three digit for the remaining columns except the first. Then we ask to not show decimal places for the last row of numbers and to add a percentage sign after the numbers. We specify this as the argument `sprintf`, which asks tinytable to format the specified contents using the `sprintf` command. Read its document to see the format string specification. We set the argument `escape` as `TRUE` in the last call to escape (prefixes) the characters special for LaTeX (which is the program that makes the pdf version of this document) appropriately.

The result is piped to `style_tt` to align the contents of the second to the eighth columns by the decimal point. This is achieved by specifying the argument `align` as `"d"`. We use `style_tt` again to align the table header row at the bottom centre. The `theme_tt` is also used here to set a LaTeX parameter.

There are many more functions in the `tinytable` package that allows us to fine tune the appearance of the table. We will need them when we want to prepare tables for publications. Do read about them from the tiny table manual. While I may use `tt` commands to pretty the dataframes that are shown in this book, I will not be showing those commands. Similarly, the default print command prints dataframe columns vertically, as expected. A better way to see the columns of a dataframe is arranging them horizontally. The function `glimpse` (package `pillars`, re-exported in `dplyr`) achieves this. I may use `glimpse` to show contents of a dataframe though I may not show the command.

TABLE 2.1
Replication of table 2.12

Summary statistic	Serum total cholesterol (mmol/l)	Diastolic blood pressure (mmHg)	Systolic blood pressure (mmHg)	Alcohol (g/day)	Cigarettes (no./day)	Carbon monoxide (ppm)	Cotinine (ng/ml)
Mean	6.287	84.6	131.26	26.76	8.7	12.8	139.5
Median	6.27	82.5	131.5	19.1	0	6	7
Std deviation	0.757	10.5	14.28	27.75	12.5	14.9	177.3
Std error	0.107	1.5	2.02	3.92	1.8	2.1	25.1
Q1	5.75	76.2	123	6.45	0	3	0
Q2	6.775	92	139	36.5	20	16	283.8
IQR	1.025	15.8	16	30.05	20	13	283.8
Minimum	4.35	65	104	0	0	1	0
Maximum	7.86	109	183	119.6	40	57	554
Range	3.51	44	79	119.6	40	56	554
CV	12%	12%	11%	104%	143%	117%	127%

Note: IQR = inter quartile range; CV = coefficient of variation. Sample size, n = 50.

2.3.1 Grouped Frequency Distribution Table

Table 2.13 (page 53)

To make the grouped frequency table 2.13, we use

```
table(
  cut(tbl_2_10$cholesterol,
      breaks = seq(from = 4, to = 8, by = 0.5))) -> tbl_2_13
```

The `cut` command accepts a numeric vector and returns a factor based on the options we provide. The first argument that we provide is the numeric vector. The next argument `breaks` can be a single number, in which case the numeric vector is broken up into that many groups. Here we call a command `seq` to provide a vector of cut off points. It provides a sequence of numbers starting from `from` until `to` in steps of `by`. Thus `seq(from = 4, to = 8, by = 0.5)` returns the sequence 4,4.5,5,5.5,6,6.5,7,7.5,8. The command `cut` determines the interval of the sequence in which each value of the numeric vector falls and labels it as such. The command `table` provides an aggregate of these labels. If we want our own labels, we can supply a character vector as the `label` argument to `cut`. Here, we are not changing the default labels.

```
cbind(Frequency = tbl_2_13,
      Percentage = prop.table(tbl_2_13) * 100) |>
  as.data.frame() |>
  rownames_to_column(var = "Cholesterol") |>
  mutate(Cumulative = cumsum(Percentage)) -> tbl_2_13
```

If we inspect `tbl_2_13`, we will see that it includes the frequencies only. To calculate the proportions, we can use `prop.table(tbl_2_13)`. To convert it to percentages, we can multiply the proportions with 100. We bind both sets of values together using `cbind` and convert them to a data frame using `as.data.frame`. The command `rownames_to_column` adds the row name of the supplied dataframe as a new column with the value of `var` as its heading. In our case, it adds the labels that `cut` creates.

Finally, we add the cumulative column using `mutate`. You must have inferred that `cumsum` provides cumulative sum of the numerical vector passed to it.

TABLE 2.2
Replication of table 2.13

Cholesterol (mmol/l)	Frequency	Percentage	Cumulative Percentage
(4,4.5]	1	2%	2%
(4.5,5]	2	4%	6%
(5,5.5]	4	8%	14%
(5.5,6]	11	22%	36%
(6,6.5]	11	22%	58%
(6.5,7]	11	22%	80%
(7,7.5]	7	14%	94%
(7.5,8]	3	6%	100%
Total	50	100%	

2.3.2 Histogram

Figure 2.9 (page 54)

Making a histogram from the frequency table is possible with `barplot(tbl_2_13$frequency)`. However, it will not be a true histogram as the bars would be separated from each other. To get a true histogram, we pass the ungrouped data to `hist`.

```
hist(tbl_2_10$cholesterol,
     xlab = "Serum total cholesterol (mmol/l)",
     ylab = "Frequency",
     main = NA)
```

Notice how the bars of the real histogram touch each other in contrast to that of the bar diagram (not shown). The arguments `xlab`, `ylab` and `main` serves the same functions as in `boxplot`. Here we prevent a title from being drawn.

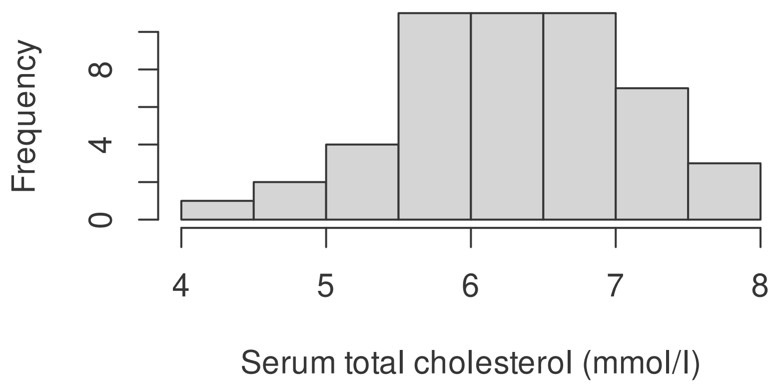


FIGURE 2.6
Replication of figure 2.9

2.3.3 Frequency Density

Table 2.14 (page 54)

We can make the frequency distribution table 2.14 using

```
c(0,10,20,30,50,70,120) -> brks
table(
  cut(tbl_2_10$alcohol,
      breaks = brks,
      include.lowest = TRUE)) -> tbl_2_14
```

Notice that we provided `include.lowest=TRUE` option while calling `cut`. Otherwise, the zeroes which are the lowest values would be excluded from the tabulation.

```
diff(brks) -> class_size
prop.table(tbl_2_14) -> rel_freq
cbind(Frequency = tbl_2_14,
      Proportion = rel_freq,
      Density = rel_freq/ class_size) |>
  as.data.frame() |>
  rownames_to_column(var = "Alcohol") -> tbl_2_14
```

Calculating the proportion is achieved the same way as in our earlier example. To calculate the frequency density we need to supply the class size. We calculate the class size using `diff` to which we supply the *brks* vector. The `diff` calculates the difference between the consecutive elements of its argument, thereby giving us the class size.

TABLE 2.3
Replication of table 2.14

Alcohol (g/day)	Frequency	Relative frequency	Frequency density
[0,10]	16	0.32	0.032
(10,20]	9	0.18	0.018
(20,30]	10	0.2	0.02
(30,50]	5	0.1	0.005
(50,70]	6	0.12	0.006
(70,120]	4	0.08	0.0016
Total	50	1	

Figure 2.10 (page 55)

As we have grouped data to get the table, we won't get a histogram using that data. Instead, we pass the ungrouped data to `hist` to make the histogram of figure 2.10. We can provide the `breaks` that we want `hist` to use instead of the breaks that `hist` calculates itself. The other argument serve the same purpose as in the graphs we have seen earlier.

```
hist(tbl_2_10$alcohol,
     breaks =   c(0,10,20,30,50,70,120),
     xlab = "Alcohol (g/day)",
     ylab = "Frequency",
     main = NA)
```

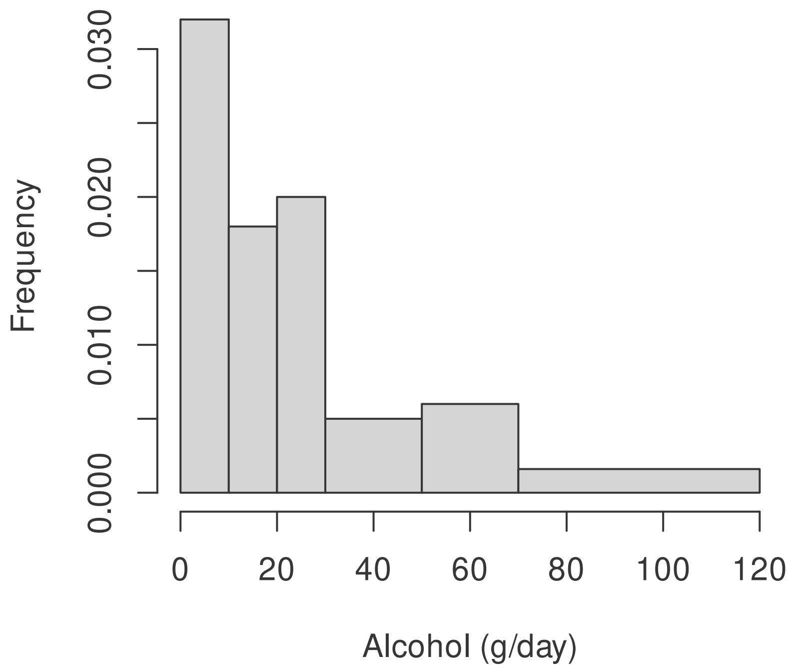


FIGURE 2.7
Replication of figure 2.10

2.3.4 Kernel Density Plot

Figure 2.11 (page 56)

The kernel density plot of figure 2.11 is made with the command

```
density(tbl_2_10$cholesterol,kernel = "epanechnikov", bw = 0.5) |>
  plot(xlab = "Serum total cholesterol (mmol/l)",
       ylab = "Density")
```

The result of `density` is piped to the `plot` command. The command `density` accepts the numeric vector, the density of which needs to be calculated, the `kernel` to be used as well as the bandwidth `bw`.

Figure 2.12 (page 57)

2.3.5 Ogive

Constructing an ogive is achieved by the command given below

```
ecdf(tbl_2_10$cholesterol) |>
    plot(xlab = "Serum total cholesterol (mmol/l)",
```

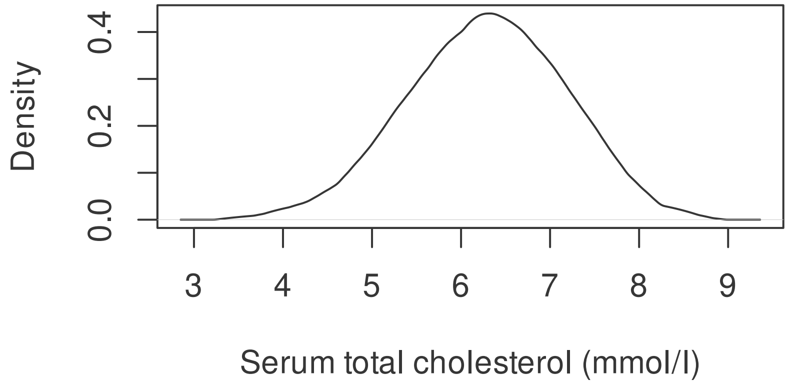


FIGURE 2.8
Replication of figure 2.11

```
        ylab = "Cumulative percentage",
        main = NA)
```

Here too we use the `plot` command. However the argument we provide to `plot` is different from what we used for kernel density plot. Here we use `ecdf` which provides the empirical cumulative distribution of the numeric vector that is supplied to it, in our case *cholesterol* of *tbl_2_10*.

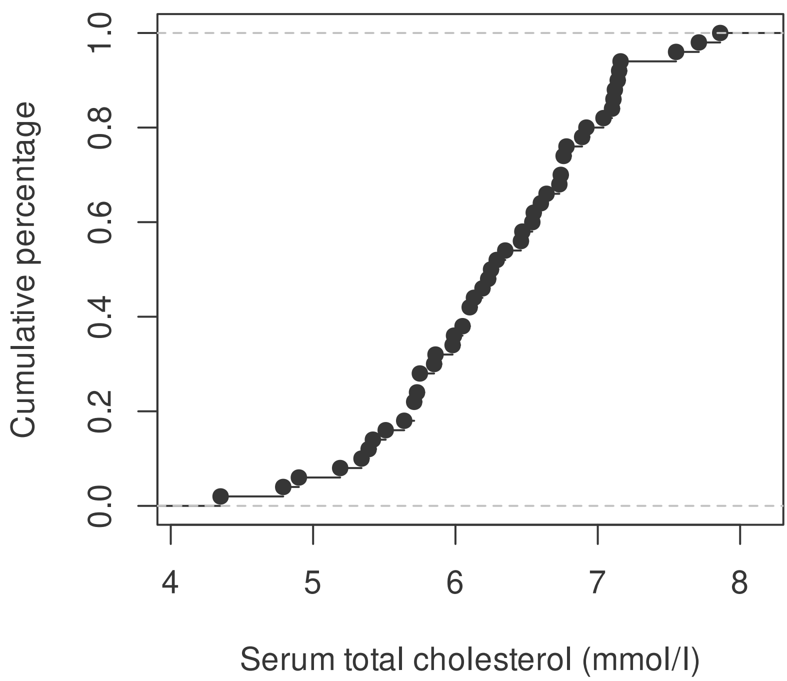


FIGURE 2.9
Replication of figure 2.12

2.4 Inferences about means

2.4.1 Normal Plots

Figure 2.14 (page 58)

We can build normal plots using

```
qqnorm(tbl_2_10$cholesterol,
       main = NA,
       xlab = "Normal scores",
       ylab = "Serum total cholesterol (mmol/l)")
qqline(tbl_2_10$cholesterol)
```

The command `qqnorm` produces a normal quantile quantile plot of the data provided. The `qqline` adds the ideal straight line. The argument supplied to `qqnorm` and `qqline` is the numeric vector holding the data which is to be checked for normality. R also provides, `qqplot` if you want to compare your data against other distributions.

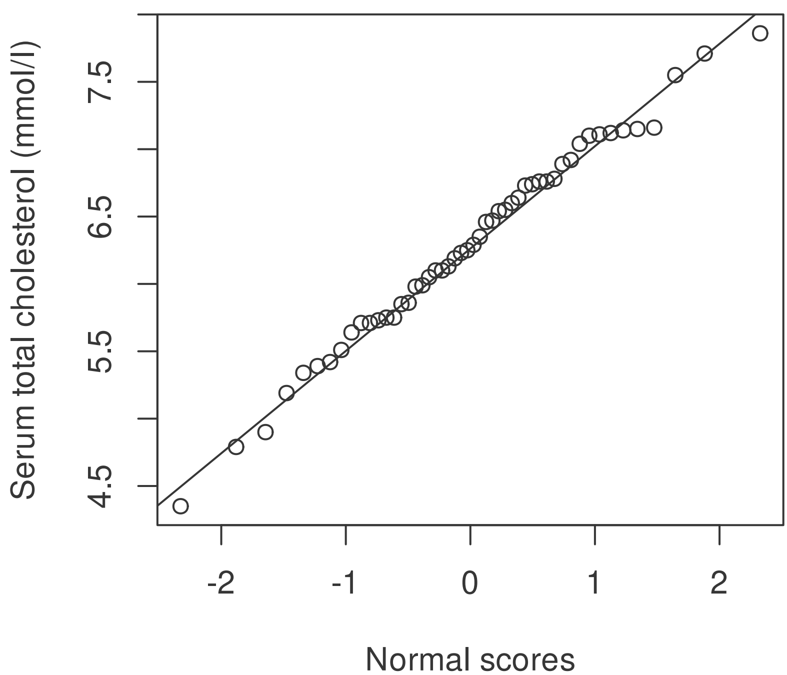


FIGURE 2.10
Replication of figure 2.14

2.4.2 Inference for a single mean

Section 2.7.2 (page 60)

To find the confidence interval for a population mean, we use

```
t.test(tbl_2_10$cholesterol)
```

The default print of `t.test` shows the mean and its 95% confidence interval. We may provide the `conf.level` argument if we desire a different confidence interval. As usual, the command by default doesn't show every value it calculates. For example, if we want to know the calculated standard error, we should store the result of the test in a variable, say *chol.t*, and inspect the component named `stderror`.

```

    One Sample t-test

data:  tbl_2_10$cholesterol
t = 58.7, df = 49, p-value <2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 6.0715 6.5017
sample estimates:
mean of x
   6.2866
```

Example 2.9 (page 61)

Example 2.9 shows the technique of assessing whether the population mean is different from a pre-specified value. We can accomplish this in R using

```
t.test(tbl_2_10$cholesterol, mu = 6)
```

Here we supplied the argument `mu`, the pre-specified value. The default print includes the test statistic calculated and p value among others. We may also specify `alternative` to indicate whether we want a one sided or a two sided hypothesis test, the default being a two sided test.

```

    One Sample t-test

data:  tbl_2_10$cholesterol
t = 2.68, df = 49, p-value = 0.01
alternative hypothesis: true mean is not equal to 6
95 percent confidence interval:
 6.0715 6.5017
sample estimates:
mean of x
   6.2866
```

2.4.3 Two sample t test

Example 2.10 (page 63)

To perform a two sample t-test, we use the same function.

```
t.test(cholesterol ~ chd, data = tbl_2_10)
```

Here, we provide a formula - *cholesterol ~ chd* as the first argument. It means that cholesterol should be split into groups using the value of *chd* and the groups should be used for the two sample t-test. The argument `data` specifies the dataframe in which to search for the variables used in the formula.

The result we get is different from the textbook. Why is that so? The reason is that the default value of the argument `var.equal` is `FALSE`. If we provide `var.equal = TRUE` to the `t.test` function, we will get the same answer as in the textbook. But, before that we need to verify if there is any evidence to say that the variances are not equal. We achieve that by

```
var.test(cholesterol ~ chd, data = tbl_2_10)
```

```
    F test to compare two variances

data:  cholesterol by chd
F = 0.78, num df = 38, denom df = 10, p-value = 0.55
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.23891 1.87734
sample estimates:
ratio of variances
           0.77987
```

Like the `t.test`, `var.test` accepts a formula. The F ratio of variances that the test returns is the reciprocal of that in the textbook. This results because of the difference in treating which *chd* group as the numerator and which as the denominator. The p value is the same after allowing for rounding errors. The difference in which group is considered first is also visible in the result of `t.test`, where the t statistic calculated has a negative sign, though this doesn't alter the p value. We can provide `conf.level = 0.99`, if we want a 99% confidence interval for the difference in means.

```
    Two Sample t-test

data:  cholesterol by chd
t = -2.17, df = 48, p-value = 0.035
alternative hypothesis: true difference in means between
      group 2 and group 1 is not equal to 0
95 percent confidence interval:
 -1.041486 -0.039493
sample estimates:
mean in group 2 mean in group 1
         6.1677          6.7082
```

2.4.4 Paired t test

Example 2.11 (page 65)

We need to store the data in table 2.15 to learn to do the paired t test. If the first visit is stored in a variable named *first* and the second visits in a variable named *second*, we can perform the paired t-test using

```
first <- c(3.795,6.225,5.210,7.040,7.550,7.715,6.555,5.360,5.285,
           6.230,6.475,5.680,5.490,9.865,4.625,7.480,4.970,6.710,
           4.765,6.695,4.025,5.510,5.495,5.435,5.350,5.905,6.895,
           4.350,5.950,5.855,5.410,5.220,4.700,4.215,5.395,7.475,
           4.925,7.115,7.020,5.365,3.665,6.130,4.895,7.000)

second <- c(3.250,6.935,4.750,5.080,8.685,7.775,6.005,4.940,5.620,
            5.870,6.620,5.635,5.080,9.465,4.120,6.955,5.100,7.480,
            4.530,6.160,4.160,6.010,5.010,5.975,4.705,5.465,6.925,
            4.260,5.325,5.505,5.280,5.175,4.815,3.610,5.705,6.580,
            5.190,6.150,6.395,5.805,3.710,5.160,5.145,7.425)
```

```
t.test(second, first, paired = TRUE)
```

Here, we provide the arguments as two vectors. We may use a formula too if it is appropriate to how we store the data. The essential argument is `paired` which should be `TRUE`. There is no problem in reversing the order of the two vectors – just that the sign of the difference and its confidence interval will change.

```

    Paired t-test

data:  second and first
t = -2.02, df = 43, p-value = 0.05
alternative hypothesis: true mean difference is not equal to 0
95 percent confidence interval:
 -0.3398596 -0.0001404
sample estimates:
mean difference
          -0.17
```

2.5 Inferential techniques for non-normal data

Section 2.8.1 (page 66)

How to transform data in R should be evident by now. We use `mutate`. Thus, to obtain square root transformation of the alcohol data, we can use `mutate(tbl_2_10, talcohol`

= sqrt(alcohol)) and mutate(tbl_2_10, tco = log(co)). We need to store them into variables to enable us to use them later.

Selecting some records based on a criteria is called **subsetting**. Many functions support a subset argument. But the boxplot command uses the subset argument only when we supply a formula. We also have subset command, which can be used like subset(tbl_2_10, cigs > 0, select = c(cigs)) The first argument we supply is the object that should be subsetted. Next, we specify the criteria for subsetting. Because we supplied a dataframe as the first argument, subset will understand that the *cigs* mentioned in the criteria is a column of *tbl_2_10*. The last option select specifies that we want only the *cigs* column from the subsetted data frame. We may store the subsetted data frame as an object or we can use it directly as an argument. For example, to build figure 2.17 we may use

```
subset(tbl_2_10, cigs > 0, select = c(cigs)) |>
  boxplot(horizontal = TRUE )
```

If we need to subset one vector, we needn't use subset. We may subset the vector directly tbl_2_10$cigs[tbl_2_10$cigs > 0]. So, we can shorten our command to make the previous boxplot to

```
boxplot(tbl_2_10$cigs[tbl_2_10$cigs > 0],
        horizontal = TRUE)
```

When we want to refer to one vector of a dataframe, prefixing name of the data frame every time we require the vector will become tiring. We can address this in two ways. The first is to create a new object, say tbl_2_10$cigs -> cig and then use *cig* wherever we need *tbl_2_10$cigs*. Thus our boxplot call will become

```
tbl_2_10$cigs -> cig
boxplot(cig[cig > 0], horizontal = TRUE)
```

The second option is to attach the dataframe using attach(tbl_2_10). This makes available the data frame in the search path of R. After executing this command, we may refer to the component vectors directly without prefixing the data frame's name. If we take this route, our boxplot call will become

```
attach(tbl_2_10)
boxplot(cigs[cigs > 0],
        horizontal = TRUE,
        xlab = "Cigarettes (no.day)",
        ylab = NA)
```

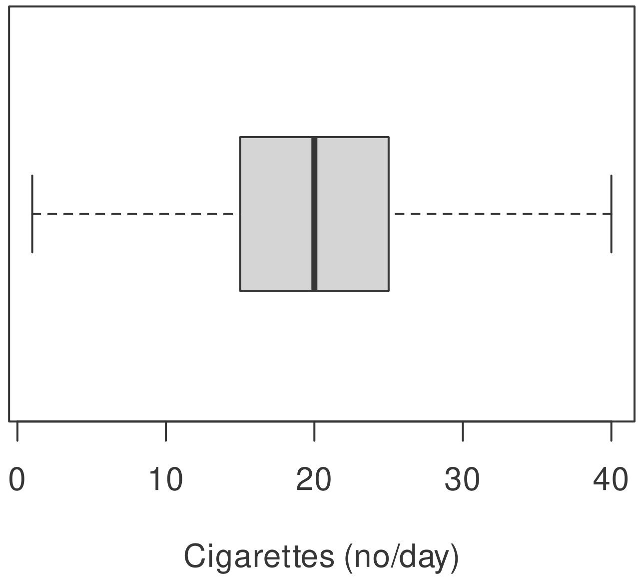


FIGURE 2.11
Replication of figure 2.17

2.5.1 Wilcoxon test

Example 2.12 (page 71)

The command to perform Wilcoxon test is `wilcox.test`. For the cholesterol data, we can use

```
wilcox.test(cholesterol ~ chd, data = tbl_2_10)
```

The command `wilcox.test` is similar in usage to `t.test`. We can use it to perform signed rank sum test, the non-parametric equivalent of one-sample t-test as well as paired test by specifying the similarly named arguments.

```
    Wilcoxon rank sum test with continuity correction

data:  cholesterol by chd
W = 130, p-value = 0.048
alternative hypothesis: true location shift is not equal to 0
```

2.6 Measuring agreement

2.6.1 Bland – Altman plot

Figure 2.18 (page 73)

Before we learn to build Bland – Altman plots, we need to import and shape the data.

```
read_table("./K11828 supplements/Datasets/Example 2.13.dat",
           col_names=c("pt", "vc"),
           col_types = cols(pt = col_number(),
                            vc = col_number())) -> xmpl_2_10
mutate(xmpl_2_10,
       differ =  pt - vc,
       avg =  (pt + vc)/2,
       dlog = log(pt) - log(vc),
       alog = (log(pt) + log(vc))/2) -> xmpl_2_10
```

While base R provides a lot of commands to draw graphs, there are many packages that improve upon the graphing capabilities of R. The packages `grid`, `lattice` and `ggplot2` are the better known ones. Here will learn the basics of `ggplot2`. The package `ggplot2` is part of `tidyverse`. So, if you have already installed `tidyverse`, there is no need to install `ggplot2` separately. While the ggplot commands are a bit verbose compared to base R, there is greater clarity and more flexibility in them. The graphs are built step by step. The command

```
ggplot(xmpl_2_10)
```

draws nothing, but sets the stage for the subsequent commands. We supply *xmpl_2_10* as the data argument to `ggplot`. This data is available to the subsequent commands. Subsequent commands can be entered if a line is terminated with `+`. `ggplot` sees each data element as being mapped to a geometric aspect of the graph. Thus, at each step, we add a **geom** function that suits our requirement. We provide an **aesthetic** mapping as an argument to the geom, specifying which data variable should map to each of the aesthetics. Seeing this in action will help you understand.

```
ggplot(xmpl_2_10) +
  geom_point(aes(x=avg, y = differ))
```

We used `geom_point` as our geom because we want points in our graph. Where do we want the dots? We want a dot at each location specified by the *avg* and *diff* vectors of our data frame. We specify this by saying `aes(x = avg, y = differ)`.

The black dots really crowd the graph. Is there a way to reduce that crowded appearance?

```
ggplot(xmpl_2_10) +
  geom_point(aes(x=avg, y = differ), alpha = 0.5)
```

Notice that `alpha` is not specified inside `aes`. We don't want `alpha` to map to any data. We want all the dots to have a translucent look. Hence, we specify it as an argument to `geom_point`. We may also change the colour and shape of the point in a similar fashion.

The axis labels don't look good. How can we change them?

```
ggplot(xmpl_2_10) +
  geom_point(aes(x=avg, y = differ), alpha = 0.5) +
  labs(x = "Mean",
       y = "Difference (PT - Clause)")
```

The `labs` accept many more arguments, including `title`, `caption` as well as the labels in legends for other aesthetics specified (like colour) .

I don't like that grey background. How can I remove them? ggplot's answer to this requirement is `theme`. There are different themes. You may choose to use any of them or build one from the default theme by tweaking any element to your satisfaction.

```
ggplot(xmpl_2_10) +
  geom_point(aes(x = avg, y = differ), alpha = 0.5) +
  labs(x = "Mean",
       y = "Difference (PT - Clause)") +
  theme(panel.grid.major.y = element_blank(),
        panel.grid.minor.y = element_blank(),
        panel.grid.major.x = element_blank(),
        panel.grid.minor.x = element_blank(),
        panel.background = element_blank(),
        panel.border = element_rect(fill= NA))
```

Here we chose to modify some elements of the default theme. The other option we have is to use one of the complete themes like `theme_bw`, `theme_minimal` etc.

The final prettying that we will do is drawing of the horizontal line at $y = 0$.

```
ggplot(xmpl_2_10) +
  geom_point(aes(x = avg, y = differ),alpha = 0.5) +
  labs(x = "Mean",
       y = "Difference (PT - Clause)") +
  theme(panel.grid.major.y = element_blank(),
        panel.grid.minor.y = element_blank(),
        panel.grid.major.x = element_blank(),
        panel.grid.minor.x = element_blank(),
        panel.background = element_blank(),
        panel.border = element_rect(fill= NA)) +
  geom_hline(yintercept = 0, colour = "green")
```

We add another geom, `geom_hline` which draws a horizontal line at the specified y intercept. The graph is drawn in the order we specify. Thus the horizontal line is drawn over the dots. It is to demonstrate this point that we used `colour` for the line. In the next example, we draw it before the dots.

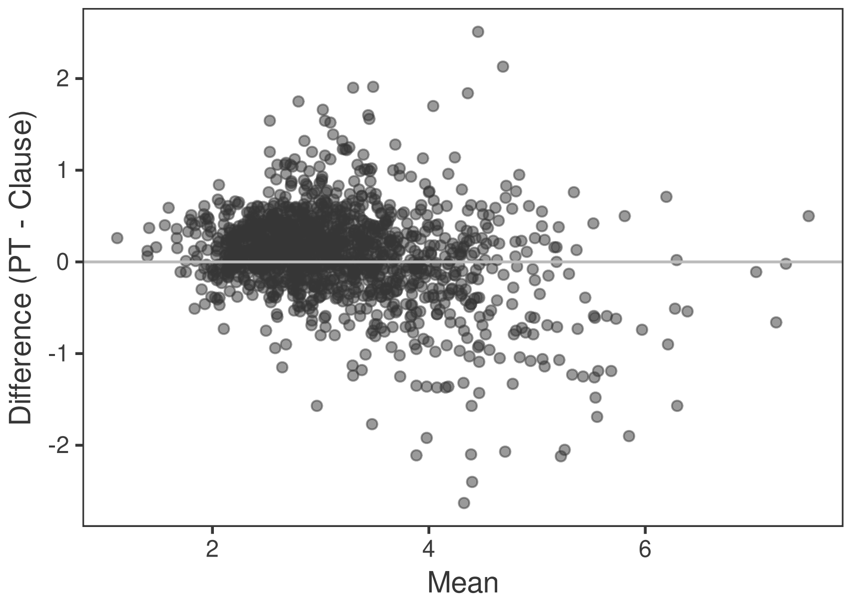


FIGURE 2.12
Replication of figure 2.18

Figure 2.19 (page 73)

Producing the Bland - Altman plot for log transformed fibrinogen data is as easy as changing the mapping of the `x` and `y` arguments to *alog* and *dlog*.

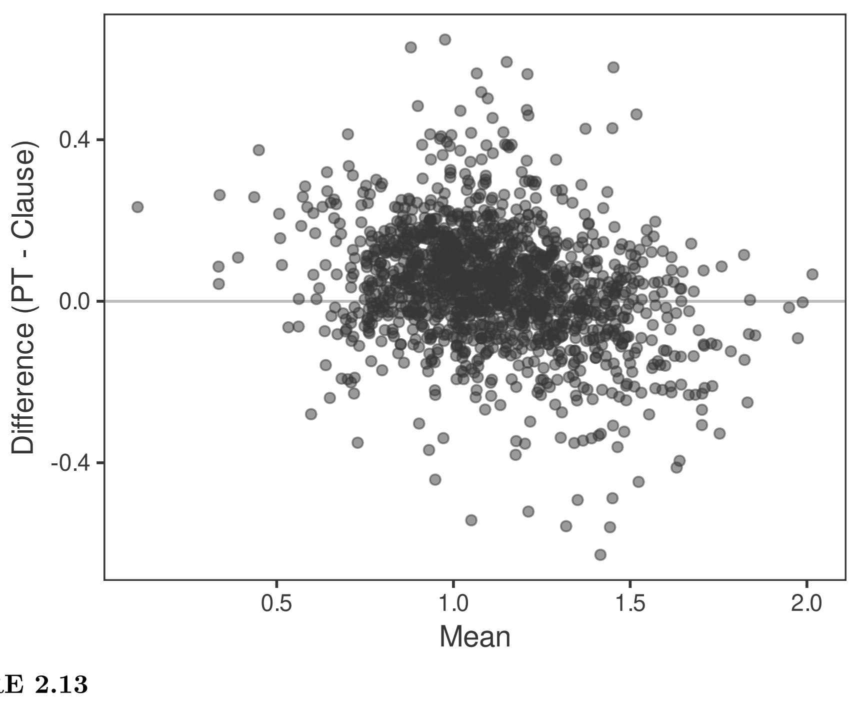


FIGURE 2.13
Replication of figure 2.19

2.6.2 Cohen's kappa

Example 2.14 (page 76)

Functions for measures of inter-rater agreement is not available in base R. Many packages provide suitable functions. Here we will use the package `DescTools`. Remember to install it once and load it whenever required. The function that we use to calculate kappa is `CohenKappa`. It requires a square matrix, which we will prepare in the next step from table 2.19.

```
matrix(c( 22,80,73,6,9,4,11,471,241,31,60,12,3,61,379,20,29,15,0,
          159,326,197,263,152,0,60,92,43,266,64,0,10,26,11,41,97),
       ncol =6,
       byrow = TRUE) -> tbl_2_19
library(DescTools)
CohenKappa(tbl_2_19)
```

By default, the function prints the unweighted kappa without a confidence interval. If we want a confidence interval, we should specify the `conf.level` argument.

```
[1] 0.30202
```

Example 2.15 (page 77)

2.6.3 Weighted kappa

To obtain weighted kappa, we should specify the `weight` argument. You have the choice to specify `Equal-Spacing` or `Fleiss-Cohen`. Or, we can specify a custom weight matrix, having the same dimension as the data we supplied. Let us first build the weight matrix.

```
sapply(1:6, function(x) {1 - abs((1:6)-x)/5}) -> wc
wc
```

The function `sapply` repeatedly passes each of the value of the first argument to the second argument, which should be a function and collects the result returned by the function in a list. The list is simplified to an array if the argument `simplify` is `TRUE`, the default value. The first argument we supply to `sapply` is a vector with the numbers 1 to 6. This vector is built by the colon`:` operator from its arguments, the numbers that precede and follow it. The second argument is an anonymous function. It accepts a number, subtracts it from each value of another vector with the values 1 to 6, takes the absolute value, divides it by five and finds the difference from one. Thus it will return a vector of length 6 corresponding to one row of weights calculated according to formula 2.33. We call this function repeatedly using `sapply`, each time with one value from 1 to 6. As we didn't change the default value of `simplify` argument, these vectors are joined together as a matrix, which is stored in the object *wc*.

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  1.0  0.8  0.6  0.4  0.2  0.0
```

```
[2,]  0.8  1.0  0.8  0.6  0.4  0.2
[3,]  0.6  0.8  1.0  0.8  0.6  0.4
[4,]  0.4  0.6  0.8  1.0  0.8  0.6
[5,]  0.2  0.4  0.6  0.8  1.0  0.8
[6,]  0.0  0.2  0.4  0.6  0.8  1.0
```

Let us call the `CohenKappa` function using these weights.

```
CohenKappa(tbl_2_19, weights = wc)
```

The same result is obtained if we use `weight="Equal-Spacing"`.

If we need the confidence interval of the weighted kappa result, we need to specify a value to the `conf.level` argument.

```
CohenKappa(tbl_2_19, weights = wc, conf.level = 0.95)
```

The confidence interval and the point estimates are the same as in our text within the tolerance of rounding error.

```
  kappa  lwr.ci  upr.ci
0.42833 0.40740 0.44927
```

2.7 Assessing diagnostic tests

2.7.1 Sensitivity and specificity

Example 2.16 (page 80)

To calculate sensitivity and other measures used to assess diagnostic tests, we will use `Conf` from `DescTools`. First we need to input the data in table 2.22.

```
matrix(c(84,10,43,92),
       nrow =2,
       dimnames = list(Dipstick = c("Positive","Negative"),
                       Culture = c("Positive","Negative"))) -> tbl_2_22
Conf(tbl_2_22, pos = "Positive")
```

The command `Conf` returns multiple calculated values including sensitivity, specificity and the predictive values. If you need just the sensitivity you can use `Sens` and for specificity alone, you can use `Spec`.

```
Confusion Matrix and Statistics

          Culture
Dipstick   Positive Negative
  Positive       84       43
  Negative       10       92

               Total n : 229
              Accuracy : 0.769
                95% CI : (0.710, 0.818)
    No Information Rate : 0.590
    P-Value [Acc > NIR] : 8.94e-09

                 Kappa : 0.546
 Mcnemar's Test P-Value : 1.1e-05

           Sensitivity : 0.894
           Specificity : 0.681
        Pos Pred Value : 0.661
        Neg Pred Value : 0.902
            Prevalence : 0.410
        Detection Rate : 0.555
   Detection Prevalence : 0.367
      Balanced Accuracy : 0.788
        F-val Accuracy : 0.760
     Matthews Cor.-Coef : 0.569

       'Positive' Class : Positive
```

2.7.2 ROC plot

Figure 2.20 (page 84)

There are many packages to plot ROC curves. However, most are aimed for real world analysis. Thus they are poor for the aggregated data presented in table 2.24. It is much easier to build the plot directly. But, first, we need the data.

```
5621 -> tot_smk
3274 -> tot_nsmk
c(5621,5460,5331,5200,5057,4932,4818,3499,1984,874,0) -> smk
c(0,817,1403,1914,2360,2696,2972,3266,3273,3273,3274) -> nsmk

smk / tot_smk -> sens
nsmk / tot_nsmk -> spec
c(1, 5:10, seq(20,50, by = 10)) -> colbl
cbind.data.frame(smk, nsmk, sens, spec) -> tbl_2_24
```

We will use ggplot to build figure 2.20

```
ggplot(tbl_2_24, aes(x= colbl)) +
  geom_line(aes( y = sens), linetype = 3) +
  geom_line(aes( y = spec), linetype = 2) +
  geom_line(aes( y = sens + spec), linetype = 1) +
  labs(x= "CO cut-point (ppm)", y = NULL) +
  annotate("text",
           x = c(40,20,40),
           y = c(.3,1.1, 1.25),
           label = c("Sensitivity", "Specificity", "Sum"))
```

Notice that we supplied the aesthetic `x` to `ggplot` as it is common to most of the geoms used. We used the same geom multiple times because we wanted multiple lines to be created, each with a different `linetype` argument that decides the dash pattern. Also, note that we supplied result of an operator `+`, as the value of one of the arguments. The `labs(x= "CO cut-point (ppm)", y = NULL)` determines the axis labels, `NULL` removing the label. We supplied vectors of length three for annotation layer because we want three labels. The x and y locations of the labels used for annotation were determined by trial and error.

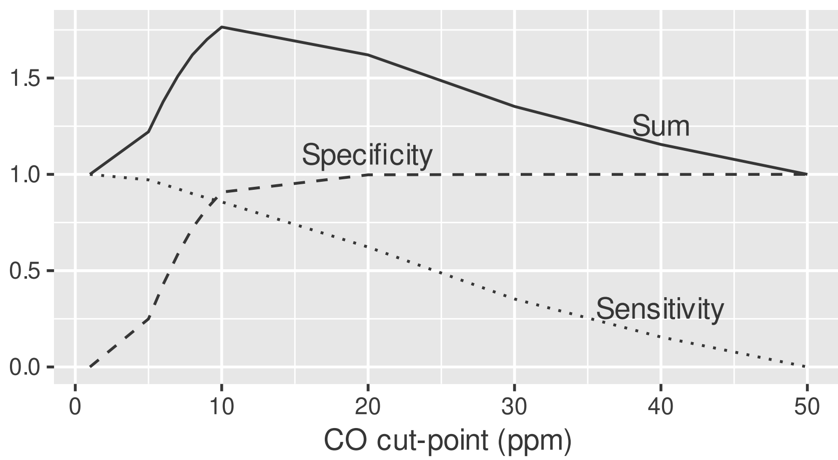


FIGURE 2.14
Replication of figure 2.20

While we won't make ROC curve for thiocyanate, we will make the ROC curve for CO in a similar manner. I hope you can understand the code without any explanation.

Figure 2.21 (page 84)

```
ggplot(tbl_2_24) +
  geom_line(aes( x = 1- spec, y = sens)) +
  labs(x= "One minus specificity", y = "Sensitivity")
```

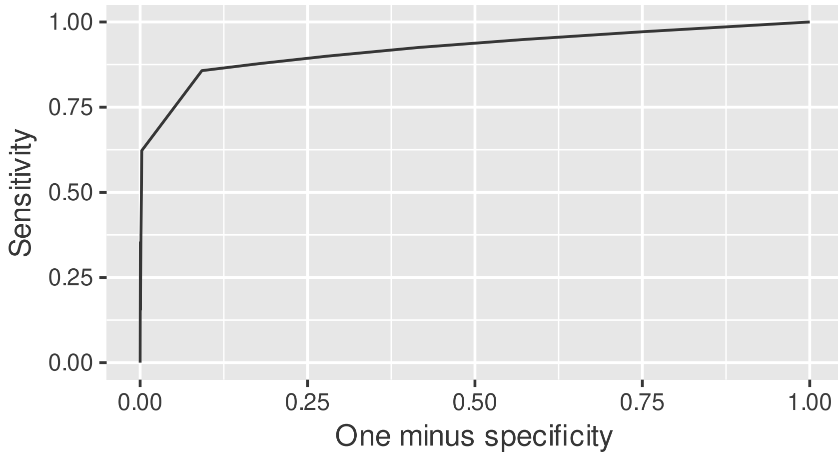


FIGURE 2.15
Replication of figure 2.21

When we analyse ungrouped data, it is better to use packages like `plotROC`, `pROC`, `PRROC`, `ROCit` etc. to plot ROC curves.

2.8 Recap

As we come to the close of a long chapter, let us recap the important topics we covered here.

2.8.1 Concepts introduced in this chapter

- list
- vector arithmetic
- factor
- tibbles
- pipes
- escape sequences
- tidy select
- subsetting
- geom
- aesthetic mapping

2.8.2 Commands introduced in this chapter

- graphics::barplot
- base::matrix
- base::rownames
- base::colnames
- base::prop.table
- stats::chisq.test
- utils::str
- base::`$`
- stats::binom.test
- stats::prop.test
- base::min
- base::max
- stats::quantile
- utils::install.packages
- base::library
- readr::read_table
- readr::cols
- readr::col_number
- readr::col_factor
- base::range
- stats::IQR
- graphics::boxplot
- base::`-`
- base::`+`
- base::`*`
- base::`/`
- base::mean
- stats::var
- stats::sd
- base::sqrt
- base::length
- base::is.na
- base::`==`
- dplyr::summarise
- dplyr::across
- stat::median
- dplyr::reframe
- dplyr::bind_rows
- dplyr::mutate
- dplyr::select
- base::t
- base::nrow
- base::as.data.frame
- dplyr::bind_cols
- base::`|>`
- tinytable::tt
- tinytable::format_tt
- tinytable::style_tt
- tinytable::theme_tt
- base::table
- base::cut
- base::seq
- base::cbind
- tibble::rownames_to_column
- base::cumsum
- graphics::hist
- base::diff
- base::plot
- stats::density
- stats::ecdf
- stats::qqnorm
- stats::qqline
- stats::t.test
- stats::var.test
- base::subset
- base::attach
- stats::wilcox.test
- base::log
- ggplot2::ggplot
- ggplot2::aes
- ggplot2::geom_point
- ggplot2::labs
- ggplot2::theme
- ggplot2::element_blank
- ggplot2::element_rect
- ggplot2::geom_hline
- DescTools::CohenKappa
- base::abs
- base::sapply
- base::`list`
- DescTools::Conf
- ggplot2::geom_line
- ggplot2::annotate

3

Assessing risk factors

The third chapter of the textbook deals with assessing risk factors. We will need the `DescTools` package to rework some of the problems in this chapter. Most of the commands need a two-by-two table which is expected as matrix. We also use the package `epiR`. Remember to install the packages using the command `install.packages` as described in Chapter 2.

3.1 Risk and relative risk

Example 3.1 (page 91)

```
library(DescTools)
matrix(c(31,15,1386,1883),
       nrow = 2,
       dimnames = list(smoker = c("Yes", "No"),
                       cvdeath = c("Yes", "No"))) -> tbl_3_2
RelRisk(tbl_3_2, conf.level = 0.95, method = "wald")
```

We use the `RelRisk` function of `DescTools`. We supply it the data stored as *tbl_3_2*, a two-by-two matrix. Without the `conf.level` argument only the point estimate is printed. The function also allow us to provide our preferred method for calculating the confidence interval.

```
rel. risk    lwr.ci    upr.ci
   2.7682    1.5129    5.0652
```

Note that the confidence interval given by the function (1.513, 5.065) is narrower than in our text (1.500,5.108). The probable explanation is the approximate nature of the formula used in the textbook.

3.2 Odds and odds ratio

Example 3.2 (page 93)

To calculate the odds ratio, we have a similar function `OddsRatio`.

DOI: 10.1201/9781003589563-3

```
OddsRatio(tbl_3_2, conf.level = 0.95, method = "wald")
```

`OddsRatio` accepts the same arguments as `RelRisk`. However, the choice of methods it allows is different from `RelRisk`.

```
odds ratio     lwr.ci     upr.ci
    2.8077     1.5099     5.2212
```

3.3 Prevalence studies

Example 3.4 (page 97)

Calculating the prevalence risk and odds ratios given in example 3.4 is done using the same functions given above.

```
matrix(c(15,41,1727,3229),
       nrow = 2,
       dimnames = list(pvd = c("Yes", "No"),
                       smoker = c("Yes", "No"))) -> tbl_3_4
RelRisk(tbl_3_4)
OddsRatio(tbl_3_4)
```

```
[1] 0.68676
[1] 0.68404
```

Here again, the risk ratio (0.69) is slightly different from the textbook value of 0.68.

Vector arithmetic

Table 3.5 (page 98)

Let us try to build table 3.5.

```
c(15,33,8) -> pvdy
c(1712,1897,1291) -> pvdn
c(pvdy, sum(pvdy)) -> pvdy
c(pvdn, sum(pvdn)) -> pvdn
pvdy + pvdn -> total
pvdy / total -> Prevalence
c("Current smoker", "Ex-smoker", "Never smoked", "Total") -> ciglabels
```

```
cbind.data.frame(ciglabels, pvdy, pvdn, total, Prevalence) -> tbl_3_5

tbl_3_5
```

First, we assign the data of those with PVD to *pvdy* and those without PVD to *pvdn*. We then modify them to add the total to the end of both vectors. The *total* and *prvl* are calculated using simple arithmetic. Finally the vectors are joined together as a dataframe for prettying.

TABLE 3.1
Replication of table 3.5

	Peripheral vascular disease?			
Cigarette smoking status	Yes	No	Total	Prevalence
Current smoker	15	1712	1727	0.0086856
Ex-smoker	33	1897	1930	0.0170984
Never smoked	8	1291	1299	0.0061586
Total	56	4900	4956	0.0112994

Note that when we say *pvdy + pvdn*, the first element of the vector *pvdy* is added with the first element of *pvdn*. In other words, R does **vector arithmetic**. This is a very useful property and easily understandable when the vectors being added or multiplied are of the same length. When the vectors are of unequal length, the shorter vector is **recycled** to the same length as the longer one. Recycling is at work, say, when we divide a vector by a single number. The single number is recycled to match the length of the longer vector and then vector division is done. Things gets confusing when the length of the longer vector is not an integer multiple of the length of the shorter one. Then, one copy of some elements of the recycled shorter vector will not be used in the vector operation. In practice, be careful when you need to perform arithmetic operations on two vectors of unequal length if the shorter one has more than one element.

3.4 Testing association

3.4.1 Chi square test

Example 3.6 (page 99)

We used `chisq.test` in the previous chapter. We rework example 3.6 here.

```
chisq.test(tbl_3_2, correct = FALSE)
```

Note that we need to say `correct = FALSE` to avoid continuity correction and get the same result as in the textbook.

```
    Pearson's Chi-squared test

data:  tbl_3_2
X-squared = 11.6, df = 1, p-value = 0.00067
```

Subsetting

Example 3.7 (page 100)

We saw two commands to test a proportion against another proportion. The `prop.test` that we saw in the previous chapter uses chi square methodology and not the normal approximation used in the textbook. The `binom.test` uses an exact method. Thus both functions won't give the same answer as in the textbook when we rework the example 3.7. However, we will use `binom.test` to rework example 3.7.

```
binom.test(x = tbl_3_2[1,1],
           n = sum(tbl_3_2[1, ]),
           p = tbl_3_2[2,1] / sum(tbl_3_2[2,]))
```

The above formula may look menacing. We will break it down for better understanding. We are providing three arguments to `binom.test` viz. `x`, `n` and `p`. We may refer to one or more elements of a vector, matrix or array. This process is called **subsetting**. To do this, we follow the variable's name with a pair of square brackets and inside the square brackets provide the indices of the elements we need, separated by commas. The argument `x` should represent the number of positive events. We subset *tbl_3_2* with `[1,1]` to say that we want the first row's first column. We need to specify two indices to get one element of matrix because matrix has two dimensions. The argument `n` should provide the total events. To `sum` we supply `tbl_3_2[1, ]`, to mean all elements of the first row of the matrix. Notice that though we put a comma within the square brackets after 1, we didn't specify any numbers after that. This is to say that we want all elements in that (here, second) dimension. The third argument is `p`, which is the hypothesized probability against which we want the previous two arguments compared. We want this to be the proportion of CV death in non smokers. We calculate this by dividing the number of CV deaths in non smokers obtained by `tbl_3_2[2,1]` by the sum of non-smokers obtained by `tbl_3_2[2,]`.

Matrices and arrays are vectors with more than one dimension. While it is better to subset them with multiple indices to suit their dimensions, they may be subset with a single number too. Thus we may refer to `tbl_3_2[1]` to refer to the first element of our matrix, the number of CV deaths among smokers.

```
    Exact binomial test

data:  tbl_3_2[1, 1] and sum(tbl_3_2[1, ])
number of successes = 31, number of trials = 1417, p-value =
7.4e-07
```

```
alternative hypothesis: true probability of success is not equal to 0.0079031
95 percent confidence interval:
 0.014912 0.030910
sample estimates:
probability of success
              0.021877
```

As expected, the p value we get (7×10^{-7}) is different from the textbook's value of 0.0007, though the interpretation is strengthened.

3.4.2 Fisher exact test

Example 3.9 (page 103)

Fisher exact test is done with `fisher.test`.

```
matrix(c(3,9,55,51),
       nrow = 2,
       dimnames = list(activiy = c("bed rest", "normal"),
                       hypertension = c("yes", "no"))) -> tbl_3_6
fisher.test(tbl_3_6, alternative = "less")
```

Note that the default is to test against the two sided alternative. Notice how we specified a list as the `dimnames` argument, to specify the labels for the rows and columns of the matrix. Dimension names are not mandatory. But, they help us to make sure that we have entered the right value for the appropriate cell.

```

    Fisher's Exact Test for Count Data

data:  tbl_3_6
p-value = 0.071
alternative hypothesis: true odds ratio is less than 1
95 percent confidence interval:
 0.0000 1.1098
sample estimates:
odds ratio
   0.31199
```

3.5 Risk factors measured at several levels

3.5.1 Linear trend

Example 3.14 (page 110)

We will use `MHChisqTest` from `DescTools` to test for linear trend.

```
matrix(c(100,382,183,668,279,109,492,1872,834,2482,974,306),
       ncol = 2,
       dimnames = list(class = c( "I","II","IIIn","IIIm","IV","V"),
                       chd = c("CHD Yes", "CHD No"))) -> tbl_2_2
MHChisqTest(tbl_2_2)
```

The matrix we supply as the argument to the command is structured so that there are two columns, with a row each for each level. Which row comes first doesn't matter.

```
    Mantel-Haenszel Chi-Square

data:  tbl_2_2
X-squared = 33.6, df = 1, p-value = 6.7e-09
```

3.5.2 Non-linear trend

Example 3.15 (page 111)

There aren't any built-in tests for non-linearity. So, we need to calculate it "by hand".

```
MHChisqTest(tbl_2_2)$statistic -> chil
chisq.test(tbl_2_2)$statistic -> chio
MHChisqTest(tbl_2_2)$parameter -> dfl
chisq.test(tbl_2_2)$parameter -> dfo
pchisq(chio-chil, dfo-dfl, lower.tail = FALSE)
```

We saw earlier that most tests in R provide much more than what is printed by default. Here, we assign the `statistic` and `parameter` values returned by `MHChisqTest` and `chisq.test` to a variable each. The `statistic` returns the statistic calculated by the test, the value of chi square in our case. The `parameter` returns the value of degrees of freedom that was calculated by the tests. We pass the difference between the chi squares and the degrees of freedom to the function `pchisq`.

R provides a family of functions to deal with statistical distributions. Typically, there are four functions for each statistical distribution. The names of these functions start with `d` for density, `q` for quantile, `r` for random and `p`, for probability. To these initial letters, a word to indicate the statistical distribution is added, `chisq` in our case. The `pchisq` like other `p`-functions returns the area under the curve of the statistical distribution to the left or right of the value we supply. When we say `pchisq(q,df, lower.tail = FALSE)`, we are asking for the area under the chisquare distribution that falls to the right of *q* in a chi square distribution with *df* degrees of freedom. In our example, as taught by the textbook, we supply the difference between the chi square values returned by the two functions as *q* and the difference in degrees of freedom returned by the two functions as the `df` for `pchisq`.

The `q`-functions returns the quantile for any given probability. That is, it is the inverse of the `p`-functions. The `r`-functions returns random values from the specified distribution. The `d`-functions returns the density of the distribution for specified quantiles.

```
X-squared
  0.59675
```

3.6 Attributable risk

Example 3.16 (page 112)

Neither base R, nor `DescTools` provide functions to calculate attributable risk. We will use the package `epiR` to rework example 3.16. Remember to install it as discussed in chapter 1.

```
library(epiR)
epi.2by2(tbl_3_2,
         method = "cohort.count")$massoc.detail$PAFRisk.strata.wald
```

The `epi.2by2` functions calculates a number of measures used to analyse two-by-two tables. The `method` argument specifies the study design. The choices we have are `cohort.count`, `cohort.time`, `case.control`, or `cross.sectional`. The individual results are available as a list in the `massoc.detail` element of the result returned. Here we use the `PAFRisk.crude.wald` sub-component. Note that the terminology used by the function and the textbook differs. What is called as attributable risk is called as population attributable fraction in this function.

```
      est   lower   upper
1 0.43046 0.13907 0.62323
```

The confidence interval returned by the function (0.139, 0.623) is different from that in the textbook (0.224,0.664).

3.7 Rates and relative rates

Example 3.18 (page 116)

Rates and relative rates are handled by `poisson.test`. Its usage is similar to `prop.test` and `binom.test`. Let us see an example, from table 3.15.

```
poisson.test(81,166582)
```

The command requires the number of events as the first argument and the time base, the denominator, as the second argument.

```
    Exact Poisson test

data:  81 time base: 166582
```

```
number of events = 81, time base = 166582, p-value <2e-16
alternative hypothesis: true event rate is not equal to 1
95 percent confidence interval:
 0.00038615 0.00060436
sample estimates:
event rate
0.00048625
```

The result returned includes the calculated rate (without multiplying it with any number), its confidence interval calculated by an exact method and the probability for the hypothesis test against the alternative that the true event rate is not one.

Example 3.19 (page 118)

Let us do the example 3.19. First the confidence interval. The `poisson.test` returns the confidence interval for the rate, not the number of events. We may convert it to events by multiplying with the population. For example, to get the confidence interval of male CHD death in the same population from which the rate was derived, we use

```
poisson.test(1080,612955)$conf.int * 612955
```

```
[1] 1016.5 1146.4
attr(,"conf.level")
[1] 0.95
```

The confidence interval calculated by the exact method of `poisson.test` (1016.541, 1146.383) is different from the values given by the textbook (1016.548, 1146.402) only at the decimal places.

The function `poisson.test` allows us to calculate rate ratio and its confidence interval directly.

```
poisson.test(c(1080, 306), c(612995, 634103))
```

We provide a vector of length two as the first two arguments. They are the number of events in the two groups that are being compared in the first vector and the time base for comparison (mid year population in our example) for the two groups. We are comparing men to women from table 3.16. Note that `poisson.test` accepts `conf.level` argument to specify the confidence interval we want. It also accepts `r` argument, a hypothesized rate or rate ratio against which the calculated rate or rate ratio needs to be tested. The type of hypothesis testing could be specified through the `alternative` argument.

```
    Comparison of Poisson rates

data:  c(1080, 306) time base: c(612995, 634103)
count1 = 1080, expected count1 = 681, p-value <2e-16
alternative hypothesis: true rate ratio is not equal to 1
95 percent confidence interval:
```

```
 3.2128 4.1587
sample estimates:
rate ratio
    3.6509
```

Again, the result given by the exact method (3.21, 4.16) differs from the textbook values (3.23, 4.17) at the decimal places.

3.8 Measures of difference

3.8.1 Risk difference

Section 3.9 (page 119)

Risk difference and its confidence intervals are calculated by `epi.2by2`.

```
epi.2by2(tbl_3_2,method ="cohort.count")$massoc.detail$ARisk.strata.wald
```

Here, we select the `ARisk.strata.wald` sub-component of `massoc.detail`. Note that `massoc.summary` component of the result given by `epi.2by2` gives a three column dataframe similar to output 3.1 given in the textbook.

```
     est   lower upper
1 1.3974 0.53788 2.257
```

The result given by the function is multiplied by 100. We may change this by supplying our preferred multiplication unit as the `units` argument.

3.9 Recap

We will end this chapter with a recap.

3.9.1 Concepts introduced in this chapter

- vector arithmetic
- recycling
- subsetting

3.9.2 Commands introduced in this chapter

- DescTools::RelRisk
- DescTools::OddsRatio
- DescTools::MHChisqTest
- base::sum
- stats::fisher.test
- stats::pchisq
- stats::poisson.test
- epiR::epi.2by2

4

Confounding and interaction

The fourth chapter of the textbook deals with confounding. We will use the add on package `epiR` and `ggplot2` in this chapter too. Make sure they are installed. To most of the commands we see in this chapter we provide arrays or matrices as arguments. First, we will rework example 4.3 for which we use `epiR`.

4.1 The concept of confounding

Example 4.3 (page 128)

```
library(epiR)
array(c(33,48,923,1722,52,29,898,678),
      dim = c(2,2,2),
      dimnames = list(house = c("rented", "owner"),
                      chd = c("yes", "no"),
                      smoke = c("nonsmoker","smoker"))) -> tbl_4_6
```

Here, first we build an **array** to hold our data. An array is needed as the data has three dimensions – smoking status, chd status and housing status. To build the array, we use the command `array`, provide it with a vector containing the data, specify the `dim` argument which is a vector specifying the maximum indices in each dimensions. We provide *c(2,2,2)* to say that our array has maximum two elements in the first, second and third dimensions. Thus, conceptually, we get two two-by-two tables. The `dimnames` is a list that contains the label for each component of each dimension. Print the contents of the array by executing *tbl_4_6* and see how the data is arranged.

```
epi.2by2(tbl_4_6)$massoc.detail$RR.strata.wald
```

We pass the array to `epi.2by2` and select the `RR.strata.wald` component of the `massoc.detail` component of the result using the `$` operator.

```
     est   lower  upper
1 1.2729 0.82292 1.9689
2 1.3344 0.85627 2.0797
```

DOI: 10.1201/9781003589563-4

We can see the relative risk for each strata. If we want the overall measure, we should use the `massoc.summary` component of the result.

4.2 Standardisation

4.2.1 Direct standardisation

Example 4.6 (page 136)

We will use `epi.directadj` from `epiR` to rework the example 4.6.

```
matrix(c(0,0,1,6,7,16,17,25,0,0,4,7,13,11,28,44,
         0,0,1,9,17,19,43,53,0,1,5,10,15,24,28,56),
       nrow = 4,
       byrow = TRUE,
       dimnames = list(deprive = c("I", "II", "III", "IV"),
                       age = c("25-29","30-34","35-39","40-44",
                               "45-49","50-54","55-59","60-64"))) -> grp_e

matrix(c(4784,4210,3396,3226,2391,2156,2182,2054,
         4972,4045,3094,2655,2343,2394,2597,2667,
         4351,3232,2438,2241,2360,2708,2968,2802,
         4440,3685,2966,2763,2388,2566,2387,2380),
       nrow = 4,
       byrow = TRUE,
       dimnames = list(deprive = c("I", "II", "III", "IV"),
                       age = c("25-29","30-34","35-39","40-44",
                               "45-49","50-54","55-59","60-64"))) -> grp_p

matrix(c(8,6,6,6,6,5,4,4),
       nrow = 1,
       byrow = TRUE) -> std_dir
epi.directadj(grp_e, grp_p, std_dir, unit = 1000)$adj.strata
```

The function requires three matrices. The first argument is called `obs` and it should be a matrix with as many rows as the number of groups we have. It should have as many columns as there are covariates (age groups in our case). The value in the argument is the number of events. The second argument `tar` has a similar structure; but, the value should be the population for the appropriate group. The third argument `std` is a matrix with the number of columns similar to the other two arguments. It however should have only one row. The value of each cell would be the standard population for the group. We also supply the argument `unit` which is the multiplier that we want the result to be multiplied with, *1000* in our case.

```
  strata obs   tar    est  lower  upper
1      I  72 24399 3.2766 2.5593 4.1449
2     II 107 24767 4.1991 3.4245 5.1162
```

```
3    III 142 23100 5.2993 4.4334 6.3086
4     IV 139 23575 5.7545 4.8282 6.8230
```

We are interested in only the standardised rates. Hence we print only the `adj.strata` component of the result. We get the direct standardised rate per thousand for each group and its 95% confidence interval. The function supports `conf.level` argument in case we want another confidence interval. The unadjusted crude rates are available in the component `crude.strata` if we want to see that. The `crude` component provides crude covariate specific (age specific, in our example) rates for each category along with their confidence intervals.

4.2.2 Indirect standardisation

Example 4.7 (page 139)

We will use the `epi.indirectadj` function from `epiR` to calculate the indirect standardised rates. We will use the same matrices we used for the direct standardisation except for `std`.

```
c(margin.table(grp_e, margin = 2) * 1000 /
    margin.table(grp_p, margin = 2),
  margin.table(grp_e) * 1000 /
    margin.table(grp_p)) -> std_indir
```

First we calculate *std_indir* to supply as the argument `std`. It is calculated from the other two matrices as we are using an internal standard. We divide the column totals of the events data matrix by the column total of the population data matrix. The column totals are calculated using the function `margin.table`, specifying the `margin` as `2`. To this we append the grand total of events by the grand total of the population. The function `c` encloses the two sets of proportions, joining them together as one vector. Thus the matrix supplied as `std` argument is a matrix with one row. The number of columns it has is one more than the number of columns in the other two matrices. Though the function will not report any error even if the last element of the `std` is omitted, including it enables the calculation of indirect standardised rates.

```
epi.indirectadj(grp_e,
                grp_p,
                std_indir,
                units = 1000)$smr.strata
```

The `smr.strata` component of the result contains the standardised event ratio and their confidence intervals.

```
    obs    exp        est      lower     upper
I    72 103273 0.00069718 0.00054225 0.0008618
II  107 118505 0.00090291 0.00073415 0.0010801
III 142 125632 0.00113029 0.00094721 0.0013213
IV  139 112590 0.00123456 0.00103028 0.0014477
```

```
epi.indirectadj(grp_e,
                grp_p,
                std_indir,
                units = 1000)$adj.strata
```

The indirect standard rates are available in the `adj.strata` component of the result. The crude rates are available in the `crude` component of the result.

```
       est  lower  upper
I   3.3462 2.6026 4.1363
II  4.3336 3.5236 5.1842
III 5.4250 4.5463 6.3418
IV  5.9254 4.9450 6.9485
```

Figure 4.6 (page 140)

To make the graph in figure 4.6, we need to collect the three groups of estimates together. We will be using `ggplot2`. Make sure `tidyverse` was installed following the instructions in chapter 1.

```
epi.directadj(grp_e, grp_p,std_dir,unit = 1000)$adj.strata[,4] -> dire
epi.indirectadj(grp_e, grp_p,std_indir,units = 1000)$adj.strata[,1] -> inde
epi.directadj(grp_e, grp_p,std_dir,unit = 1000)$crude.strata[,4] -> crude
epi.directadj(grp_e, grp_p,std_dir,unit = 1000)$adj.strata[,1] -> dgroup
dire/dire[1] -> dirrr
inde/inde[1] -> indrr
crude/crude[1] -> crudrr
library(tidyverse)
data.frame(dgroup,dirrr, indrr, crudrr) -> rrdata
ggplot(rrdata) +
  geom_line(aes(x = dgroup, y = dirrr, group=1),
            color = "#004B73", linetype = 3) +
  geom_point(aes(x = dgroup, y = dirrr, group=1),
             color = "#004B73", shape = 3) +
  geom_line(aes(x = dgroup, y = indrr, group=1),
            color = "#713430", linetype = 2) +
  geom_point(aes(x = dgroup, y = indrr, group=1),
             color = "#713430", shape = 16) +
  geom_line(aes(x = dgroup, y = crudrr, group=1),
            color = "#111111", linetype = 1) +
  geom_point(aes(x = dgroup, y = crudrr, group=1),
             color = "#111111", shape = 16) +
  labs(x = "Deprivation group",
       y = "Relative rate") +
  annotate("text", x = c(2.15,3),
           y = c(1.75,1.5),
           label = c("Unadjusted", "Adjusted")) +
  scale_y_continuous(breaks = seq(1,2,by=0.5))
```

We subset the results returned by the functions to obtain just the estimates. We then calculate relative rates by dividing each of the estimates by the estimate of the reference group. Then all the three relative rates and the group labels are combined together to form a dataframe. The `ggplot` function is called with this dataframe. We use `geom_line` and `geom_point` three times each, one for each group of estimates. In each geom specification, we ask for a different colour specified as an RGB string starting with `#` followed by three pairs of two hexadecimal numbers to represent the contribution of red, green and blue and line type or point style. Finally, the axes are labelled and the lines annotated. The `scale_y_continuous` is used to change the default axis ticks placement. Its `breaks` argument will decide the axis ticks placement. Here we ask them to be placed at 1,1.5 and 2, the numbers being generated by `seq`.

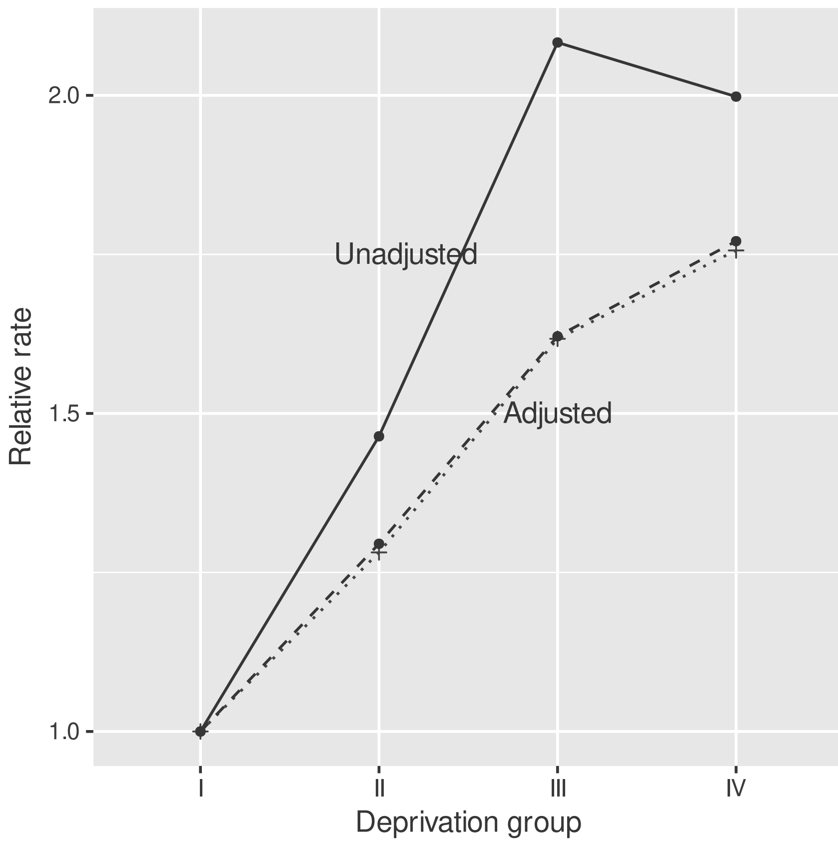


FIGURE 4.1
Replication of figure 4.6

4.2.3 Standardisation of risks

Example 4.8 (page 143)

I couldn't find a ready made function in R that can standardise risks assuming binomial probability distribution. So, we do the calculations "by hand". This may look intimidating at first. It is okay if you don't understand it in the first go. You can come back and try this at a later stage too. That said, I assure you that it is simple. First we prepare the data.

```
array(c(1,3,4,7,10,16,
        1,6,7,16,17,25,
        0,4,8,5,12,24,
        4,7,13,11,28,44,
        1,4,7,10,19,31,
        1,9,17,19,43,53,
        2,5,6,11,15,38,
        5,10,15,24,28,56),
      dim = c(6,2,4),
      dimnames = list(agegrps = c("35-39", "40-44", "45-49",
                                  "50-54","55-59","60-64"),
                      events = c("deaths", "coronaries"),
                      deprive = c("I","II","III","IV"))) ->  tbl_4_11
```

The data is input as an array. We use the `array` function for this purpose. Our array has three dimensions specified as its `dim` argument. In the first dimension we will have a maximum index of 6, in the second dimension a maximum index of 2 and a maximum index of 4 in the third dimension. The names for each index is given using the `dimnames` argument. The array is named *tbl_4_11*. Print it and see the structure.

```
result <- list(stdrisk = c(), serr = c())
```

Next, we use `list` to initiate a **list** to hold the intermediate stage of our result. A list is another data structure in R. It can have multiple sub-components which needn't be of the same type or length. Our *result* will have two components *stdrisk* and *serr*. Both the components are empty initially.

```
for (i in 1:4) {
    result$stdrisk[i] <- sum(
      apply(tbl_4_11,
            "agegrps",
            function(x) {x["deaths",i] * sum(x["coronaries",]) /
                                       x["coronaries",i]}))
    result$serr[i] <- sqrt(
      sum(
        apply(tbl_4_11,
              "agegrps",
              function(x) {sum(x["coronaries",])^2 * x["deaths",i] *
                                         (x["coronaries",i] -
                                            x["deaths", i]) /
                                         x["coronaries", i]^3})))}
```

Then we build a **loop** using `for`. The sequence given after the keyword `in` within the parentheses that follow the keyword `for` determines the number of times the commands provided to the loop will be executed. The first time the loop is executed, the first value of the sequence is made available for the commands provided to the loop as a variable with the name that we specify inside the parentheses that follow `for` and before `in`. Then with each repetition, the value of this variable gets incremented by one till the maximum value we specified. In our case, the value 1 will be available for the commands inside the loop under the variable name *i* when the loop is executed the first time. Subsequently with each repetition, its value increases by one until the value 4.

What happens with each repetition of the loop? The commands specified within the curly braces gets executed. In our case, we make two assignments – to the *i*th element of the two components of the *result* that we created. Thus, we will get four values each in the two components. What value will get assigned? The workhorse in both cases is the function `apply`. It accepts an array, in our case *tbl_4_11*, and executes the function specified as its third argument passing to that function a subset of its first argument got by incrementing the values of the dimension specified as its second argument. We supply an **anonymous function** as the third argument to do part of the mathematical manipulations (the summation across each deprivation group) as given in the textbook. Note that while subsetting the array, we subset the required elements using the labels we assigned, which is less confusing than using numeric indices. The anonymous function is anonymous because it doesn't have a name. It is a temporary function that exists only inside the `apply` calls. Thus, `apply` calls the anonymous function with all values for each age group and collects the value returned by the anonymous function in a vector. As *tbl_4_11* has 6 age groups, the result of `array` will have six elements. The `apply` is an argument for `sum`. It sums all the six components and assigns it to the appropriate element of *result*. In case of the *serr* component, it is the square root of the sum calculated using `sqrt` that is assigned.

```
result$stdrisk / marginSums(tbl_4_11[,"coronaries",])
result$serr / marginSums(tbl_4_11[,"coronaries",])
```

After the `for` loop, both the components of the *result* are divided by the sum of all coronary events to get the final result.

```
[1] 0.58713 0.49871 0.52036 0.55820
[1] 0.057491 0.048311 0.040947 0.041810
```

4.3 Mantel Haenszel methods

Examples 4.9, 4.10, 4.11 (pages 144, 146, 148)

The Mantel Haenszel calculations are done by `epi.2by2`.

```
epi.2by2(tbl_4_6)$massoc.detail$OR.mh.wald
epi.2by2(tbl_4_6)$massoc.detail$RR.mh.wald
epi.2by2(tbl_4_6)$massoc.detail$chi2.mh
```

To rework examples 4.9, 4.10 and 4.11 all we need do is select the appropriate component.

```
     est   lower  upper
1 1.3176 0.95375 1.8203
```

```
     est   lower  upper
1 1.3035 0.95497 1.7792
```

```
  test.statistic df p.value.1s p.value.2s
1         2.8049  1   0.046989   0.093978
```

The Mantel Haenszel chi square value returned by `epi.2by2.` 2.8, is different from the textbook value of 2.54 because continuity correction is not applied by `epi.2by2`. The function `mantelhaen.test` in the automatically loaded `stats` package also can perform the Mantel Haenszel test. It allows us to specify if we want to apply continuity correction. We may also specify whether we want a one sided hypothesis test.

```
mantelhaen.test(tbl_4_6, correct = TRUE)
```

```

    Mantel-Haenszel chi-squared test with continuity correction

data:  tbl_4_6
Mantel-Haenszel X-squared = 2.54, df = 1, p-value = 0.11
alternative hypothesis: true common odds ratio is not equal to 1
95 percent confidence interval:
 0.95375 1.82033
sample estimates:
common odds ratio
           1.3176
```

The result returned by `mantelhaen.test` includes the common odds ratio calculated and its confidence interval in addition to the Mantel Haenszel chi square statistic and its p value. However, there is no option to obtain common relative risks. On the other hand, it permits us to use an exact method.

Example 4.12 (page 148)

The example 4.12 uses table 4.15, which has four dimensions – disease status, smoking status, age group and occupation. However, both `mantelhaen.test` and `epi.2by2` accept an array of not more than three dimensions. However, this restriction is not really important. Mantel Haenszel method works on two-by-two tables. The higher order arrangement of two-by-two tables doesn't matter. So, we can arrange all the two-by-two tables in the third dimension rather than arranging them in third and fourth dimensions. The labels will make things clearer.

```
array(c(0,2,0,7,
        3,0,2,6,
        1,0,3,10,
        2,5,1,24,
        2,2,2,18,
        4,1,1,12,
        3,6,0,49,
        2,4,2,23,
        0,6,1,19,
        0,11,0,42,
        0,6,1,11,
        1,3,0,15),
      dim = c(2,2,12),
      dimnames = list(disease = c("diseased", "no disease"),
                      smoke = c("smoker", "non smoker"),
                      strata = c("house wife and < 45",
                                 "white collar and < 45",
                                 "other occup and < 45",
                                 "house wife and 45 - 54",
                                 "white collar and 45 - 54",
                                 "other occup and 45 - 54",
                                 "house wife and 55 - 64",
                                 "white collar and 55 - 64",
                                 "other occup and 55 - 64",
                                 "house wife and > 65",
                                 "white collar and > 65",
                                 "other occup and > 65"))) -> tbl_4_15
```

Print the *tbl_4_15* to see its structure. Both `epi.2by2` and `mantelhans.test` should accept this array. However, `epi.2by2` returns an error, probably because of multiple zero values. We will use `mantelhans.test`

Though the default options give results similar to that in the textbook, it probably is better to use an exact test.

```
mantelhaen.test(tbl_4_15, exact = TRUE)
```

```

    Exact conditional test of independence in 2 x 2 x k tables

data:  tbl_4_15
S = 18, p-value = 1.5e-07
alternative hypothesis: true common odds ratio is not equal to 1
95 percent confidence interval:
  4.0474 33.5511
sample estimates:
common odds ratio
           11.098
```

4.4 Testing for interaction

Example 4.13, 4.15 (pages 153, 156)

To rework the example 4.13, we need to reproduce table 4.16.

```
array(c(67,46,2061,3454,8,11,51,41),
      dim = c(2,2,2),
      dimnames = list(smoke = c("Smoker", "Non_smoker"),
                      chd = c("CHD_Yes", "CHD_No"),
                      previous = c( "No previous MI",
                                    "previous MI"))) -> tbl_4_16
```

Woolf test of homogeneity is calculated by `epi.2by2`, using relative risk and odds ratio.

```
epi.2by2(tbl_4_16)$massoc.detail$wRR.homog
epi.2by2(tbl_4_16)$massoc.detail$wOR.homog
```

```
  test.statistic df   p.value
1         8.0604  1 0.0045244
```

```
  test.statistic df   p.value
1         6.9418  1 0.0084202
```

The chi square calculated using both RR (8.06) and OR (6.94) differ slightly from the values given in the textbook (8.67 and 7.33) respectively. However, the interpretation does not change. `WoolfTest` available in `DescTools` also performs Woolf test of homogeneity. However, it is based on odds ratio only. `DescTools` has `BreslowDayTest` which performs the Breslow-Day test of homogeneity, again, based on odds ratio. `BreslowDayTest` accepts a `correct` argument, which, if set as `TRUE` performs Tarone correction.

4.4.1 Interaction plots

Figure 4.8 (page 154)

First, we need to prepare the data for plotting.

```
bind_cols( lrisk = as.vector(log(tbl_4_16[,1,] /
                                 margin.table(tbl_4_16,c(1,3)))),
           smoke = rep(c("Smoker", "Nonsmoker"), times = 2),
           previous = gl(2,2,labels = c("No MI","MI"))) -> data_4_8
```

With the above command, we are building a dataframe named *data_4_8* using the `bind_cols` function. We are building a dataframe with three columns names *lrisk*, *smoke* and *previous*. We pass three named arguments to match our requirement.

To build the first column, we subset the *tbl_4_16*. We select the elements in the first column. It is the **first** column because we specified `1` inside the square brackets. It is first of the **columns** because `1` is written as the second component inside the square brackets, separated from the empty first and third component by commas. If we wanted the first row, we would have used `[1,,]` and if we wanted the first of the two two-by-two tables, we would have used `[,,1]`. By doing this subsetting, we are isolating the chd numbers in both strata. We then divide these numbers by their corresponding row totals to calculate the risk. Totalling is carried out by `margin.table(tbl_4_16,c(1,3))`. This function accepts an array and computes the sum of the elements for the specified margins. We specify `c(1,3)` as our `margin` argument to mean that we want row totals (dimension 1) to be calculated for each two-by-two tables (dimension 3). We divide the first set of numbers by the calculated totals to obtain the risks and pass them to `log`, which calculates the log of the risks. Because we started with an array, we will be left with an array, which will result in two columns when we pass that as such to `bind_cols`. To have only one column, we convert the array to a vector using `as.vector`.

The columns *smoke* is built using `rep`, which repeats the argument according to our specification. In case of *smoke*, we provide the argument `times` resulting in two copies of the vector we supplied strung together. In case of *previous*, we use `gl` which is similar to `rep`, but returns a factor. While we could have used `rep` as well, providing in place of `times`, the argument `each` which causes each of the element of the vector we supplied to be repeated twice, the resulting vector when converted as a factor will have a different reference base, which will determine which of the values is plotted near the origin of our graph. You should compare

```
factor(rep(c("No MI", "MI"), each = 2))
```

and

```
gl(2,2, labels = c("No MI", "MI"))
```

We now plot the data we prepared.

```
ggplot(data_4_8, aes(x = previous,
                     y = lrisk,
                     color = smoke,
                     linetype = smoke)) +
  geom_line(aes(group = smoke), show.legend = FALSE) +
  geom_point(show.legend = FALSE) +
  annotate("text",
           x = c(1.1,1.5),
           y = c(-3,-3.75),
           label = c("Smoker", "Nonsmoker")) +
```

```
labs(x = "Previous MI status",
     y = "Log(risk of CHD)") +
scale_y_continuous(breaks = seq(-4.5, -1.5, 1),
                   limits = c(-4.5,-1.5)) +
scale_colour_manual(values = c( "#111111", "#004B73"))
```

We provide the dataframe we built to `ggplot`. We also provide an `aes` call to `ggplot`. In this `aes` we pass the aesthetics that are common to all the geoms that we will be using. Even if we provide an aesthetic in the `ggplot` call, we are free to override its value inside any geom if we want. When we say `colour = smoke` and `linetype = smoke`, `ggplot` will assign a different colour and linetype to each set of values in the *smoke* column. Thus, we will get different colours and linetypes for smokers and nonsmokers. We use `geom_line` to plot lines. But, we add `group = smoke` as an aesthetic definition inside that geom. This leads to different lines being drawn for different values of the column *smoke*. Without this aesthetic specification, the `colour = smoke` and `linetype = smoke` specifications won't work. If we include only `group` aesthetic, but not the `colour`, we will still get different lines, but with the same colour and linetype. The `geom_point` plots points based on the common aesthetics specified inside `ggplot`. We don't want a legend to be shown for either of these geoms. Hence, the argument `show.legend = FALSE`. Instead of a legend, we annotate the plot using `annotate`. We modify the axis labels using `labs`. The graph plotted with this specification doesn't show -4.5 in the y-axis, presumably because the lowest value in our dataframe is only slightly bigger than that. Hence, we specify the y limits of the graph as well as the locations of the tick labels using `scale_y_continuous`. We use `scale_y_continuous` because we are concerned with the y axis and it represents a continuous value. The `limits` argument accepts a vector with two values specifying the lower and upper limits. The `breaks` argument tells `ggplot` where tick-labels should be placed in the y-axis.

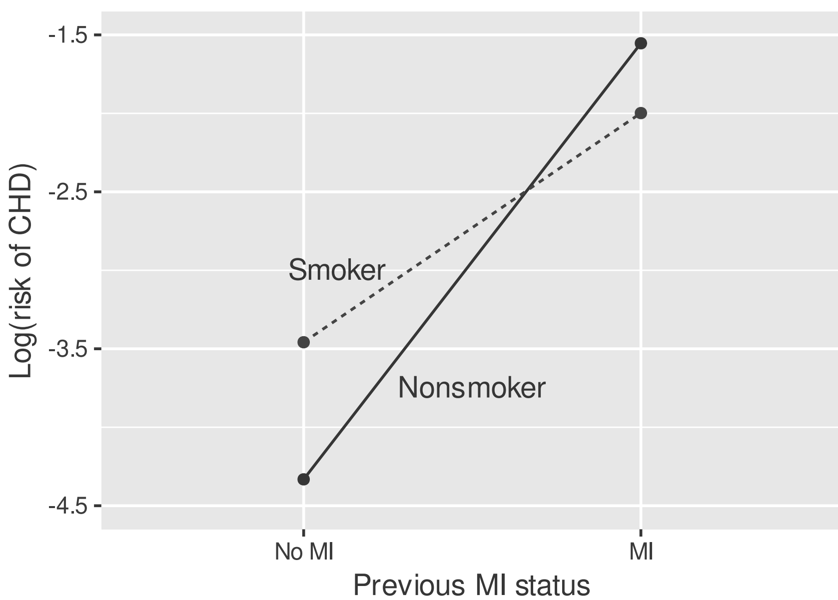


FIGURE 4.2
Replication of figure 4.8

4.4.2 Using the risk difference

Example 4.16 (page 158)

I do not know of any readymade R functions that tests for interaction using risk difference in two-by-two tables. So we will do it by hand.

```
apply(tbl_4_16,"chd",c) -> ftbl
rowSums(ftbl)  -> totals
ftbl/totals -> risks
diff(risks[c(1,3),"CHD_Yes"] - risks[c(2,4),"CHD_Yes"]) ^ 2 /
  sum(risks[,"CHD_Yes"] * risks[,"CHD_No"] / totals) -> ststc
ststc
pchisq(ststc,1,lower.tail = FALSE)
```

The first line converts the array into a matrix with four rows and two columns which we name *ftbl*. We use `apply` to apply the function `c` over *chd* margin of *tbl_4_16*, our original array to accomplish this. Then we use `rowSums` to get the row totals. We divide *ftbl* with the *totals* to get a new matrix. The values in the *risks* created will be the proportion of each cell value to the row total. In the next step, we calculate the chi square statistic according to the formula given in the textbook. We subset the appropriate elements of *risks* and find the difference between them. The numerator for calculating our test statistic, the difference of the risk differences is calculated by `diff` which is then squared using `^` operator. The denominator is calculated by multiplying the risk of CHD with its complement and dividing by the total of cases across each row, which is then summed. Finally this statistic is passed to `pchisq`, providing `df` as 1 to obtain the upper tail probability.

```
[1] 1.7057
[1] 0.19155
```

4.5 Recap

Let us recap what we learned in this chapter.

4.5.1 Concepts introduced in this chapter

- array
- array subsetting
- list
- loop
- anonymous function

4.5.2 Commands introduced in this chapter

- base::array
- epiR::epi.directadj
- epiR::epi.indirectadj
- base::margin.table
- base::data.frame
- ggplot2::scale_y_continuous
- base::for
- base::apply
- base::marginSums
- stats::mantelhaen.test
- base::as.vector
- base::rep
- base::gl
- base::factor
- base::rowSums

5

Cohort studies

The fifth chapter of the textbook deals with cohort studies. We use the packages `survival`, `dplyr`, `ggplot2`, `purrr`, `lubridate`, `epiR` and `popEpi` to work the examples in this chapter. Dataframes and vectors continue to be our main data objects. The vector that encodes events for survival analysis is expected to be numerical with the numbers 0 and 1 or 1 and 2 or logical with event indicated by 1,2 or TRUE respectively. It may also be a factor with as many levels as there are competing risks, the reference level taken as indicating censoring. We start with example 5.2. First, the data.

5.1 Cohort life tables

Example 5.2 (page 174)

```
library(tidyverse)
c(1000,995,985,965,930) -> free
c(5,10,20,35,50) -> events
events / free -> irisk
1 - irisk -> isurvival
cumprod(isurvival) -> csurvival
0:4 -> period

data.frame(period = c(period,5),
           free = c(free, 880),
           events = c(events, NA),
           irisk = c(irisk, NA),
           isurvival = c(isurvival, NA),
           csurvival = c(1,csurvival)) -> tbl_5_1
tbl_5_1
```

We create vectors to hold the serial number of periods, the number free of disease at the start of each period and the number of events during each period. Note that we don't include period 5 here. The interval risk is calculated from *events* and *free* by division and stored in *irisk*. Interval survival is calculated by subtracting *irisk* from one. Note that vector arithmetic is at work here so that each value of *irisk* is subtracted from one, one being recycled to the length of *irisk*. The cumulative survival is calculated using `cumprod` which returns the cumulative product of the supplied argument. That is, for each element of the supplied argument, product of all elements upto and inclusive of that element is calculated. Finally, we join together the vectors to form a dataframe. It is here that we add the period 5. We also shift the calculated cumulative survival one element down by adding 1 at the beginning

DOI: 10.1201/9781003589563-5

of the vector, thus making it clear that the column reflects the state at the beginning of the period.

`NA` is a special value in R which stands for missing values. Though classed as a logical data type, it is compatible with all the usual datatypes. Thus when a value in a numerical or character vector is missing, `NA` is inserted there. Calculations using `NA` propagates. For example, if you do an arithmetic operation on a vector with one or more `NA`s, the result corresponding to those `NA`s would be `NA`s. If you do an operation that uses all the elements of a numerical vector, some of which are `NA`s, then the result would be `NA`. An example would be `mean`. While this may surprise us, it really is the right answer – If you do not know one or more numbers in a collection, then you cannot know their average. R expects you to explicitly instruct it to remove `NA`s and do calculation on the reduced collection. Most functions provide an `na.rm` argument for this purpose.

```
  period free events    irisk isurvival csurvival
1      0 1000      5 0.005000   0.99500     1.000
2      1  995     10 0.010050   0.98995     0.995
3      2  985     20 0.020305   0.97970     0.985
4      3  965     35 0.036269   0.96373     0.965
5      4  930     50 0.053763   0.94624     0.930
6      5  880     NA       NA        NA     0.880
```

Figure 5.3 (page 174)

```
ggplot(tbl_5_1) +
  geom_step(aes(x = period, y = csurvival))+
  labs(x = "Survival times (years)",
       y = "Probability of survival")
```

We supply the dataframe we prepared earlier to `ggplot`, and plot the data using `geom_step`, which uses lines that change direction by 90 degrees to create a stairstep plot.

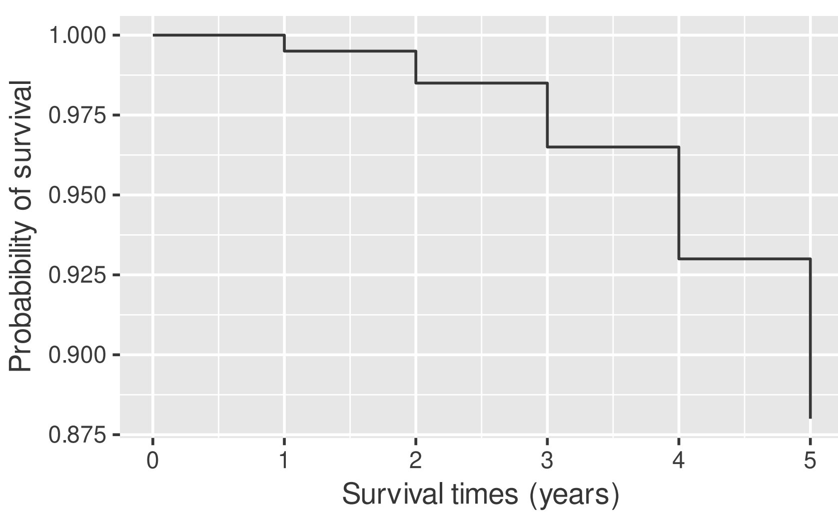


FIGURE 5.1
Replication of figure 5.3

Example 5.3 (page 175)

We rework example 5.3 to calculate the standard error of cumulative survival.

```
sqrt(
  sum(tbl_5_1$irisk / (tbl_5_1$free - tbl_5_1$events),
      na.rm = TRUE)) *
  tbl_5_1$csurvival[6] -> serr_5_3
```

Each value of interval risk is divided by the difference between the number free of disease and events. These values are summed taking care to exclude `NA`s. The square root of this sum is multiplied with the last cumulative survival to obtain the standard error which is saved with the name *serr_5_3*. The index of the last value is given by `length` which returns the length of its argument, which will be the index of the last element in that vector.

```
0.95 -> ci
(qnorm((1 - ci) / 2, lower.tail = FALSE) * c(-1,1) * serr_5_3) +
  tbl_5_1$csurvival[6]
```

We use `qnorm` to get the z value rather than typing it directly. We provide `qnorm` half of one minus the confidence interval we want (which we stored as *ci*). The argument `lower.tail = FALSE` is provided so that we get a positive value. We multiply this with the vector `c(-1,1)`. Thus we get two copies of the z value, one positive and negative. We multiply the standard error we calculated with this vector. Next, we add the last cumulative survival to this. The positive value gets added and negative value gets subtracted, giving us the confidence interval.

```
[1] 0.85986 0.90014
```

Example 5.4 (page 177)

We now turn to example 5.4.

```
c(7,12,24,19,21,15,501,2143,1375,66) -> n.censor
c(17,22,26,23,37,38,31,20,5,0) -> n.event

n.censor + n.event -> n.total

4402 -> start
c(start,(start - cumsum(n.total))[-length(n.total)]) -> n.risk
n.risk - 0.5 * n.censor -> n.adj
```

Example 5.4 is reworked similar to our previous example. To keep the table similar to table 5.3, we have not added 1 to the top of cumulative survival. We start with the vectors to represent the value of censored and events. We calculate their sum. The at risk is calculated by subtracting cumulative sum of censored plus events from the number of participants at the start. **Negative indexing** is used to exclude the last value and the number of participants at start is added as the first value. The adjusted number at risk is calculated by subtracting half of censored from the number at risk.

```
n.event / n.adj -> i.risk
1 - i.risk -> i.survival
cumprod(i.survival) -> cum.survival
```

The interval risk, interval survival and cumulative survival are calculated as in the previous example except that the adjusted at risk number is used.

```
i.risk / (n.adj - n.event) -> riskbyadj
sqrt(cumsum(riskbyadj)) -> sqrtsumriskbyadj
cum.survival * sqrtsumriskbyadj -> serr

data.frame(time = 0:9,
           number = n.risk,
           censored = n.censor,
           adjusted = n.adj,
           events = n.event,
           int.risk = i.risk,
           cum.survival = cum.survival,
           std.err = serr)-> tbl_5_3
tbl_5_3
```

As we want to calculate the standard error at each time, we save intermediate results of std error calculations in vectors. Finally, we join together the relevant columns to re-produce table 5.3.

```
   time number censored adjusted events int.risk cum.survival std.err
1     0   4402        7   4398.5     17    0.004        0.996   0.001
2     1   4378       12   4372.0     22    0.005        0.991   0.001
3     2   4344       24   4332.0     26    0.006        0.985   0.002
4     3   4294       19   4284.5     23    0.005        0.980   0.002
5     4   4252       21   4241.5     37    0.009        0.971   0.003
6     5   4194       15   4186.5     38    0.009        0.963   0.003
7     6   4141      501   3890.5     31    0.008        0.955   0.003
8     7   3609     2143   2537.5     20    0.008        0.947   0.004
9     8   1446     1375    758.5      5    0.007        0.941   0.005
10    9     66       66     33.0      0    0.000        0.941   0.005
```

Figure 5.4 (page 177)

We will now try to plot the graph in figure 5.4.

```
bind_rows(list(cum.survival = 1, ll = 1, ul = 1, time = 0),
          mutate(tbl_5_3,
                 ll = cum.survival -
```

```
                    (qnorm(0.025,lower.tail = FALSE) *std.err),
                  ul = cum.survival +
                    (qnorm(0.025,lower.tail = FALSE) *std.err),
                  time = time + 1)) -> tbl_5_3
ggplot(tbl_5_3) +
  geom_step(aes(x = time, y = ll), linetype = 2) +
  geom_step(aes(x = time, y = ul), linetype = 2) +
  geom_step(aes(x = time, y = cum.survival), linetype = 1) +
  scale_x_continuous(breaks = 0:9) +
    labs(x = "Survival times (years)",
         y = "Probabibility of survival")
```

We use `mutate` to calculate *ll* and *ul*, the lower and upper limits of confidence interval. We also add one to the time periods, effectively shifting the cumulative survival and other values to reflect the start of the period. To this we use `bind_rows` to add a row with values at the start of study. This row is made with `list` and its elements have the same names as the columns of the dataframe. Those columns not in the list will have a value of `NA`. We use the mutated *tbl_5_3* as the `data` argument to `ggplot`. We use `geom_step` to plot the cumulative survival, the *ll* and the *ul*; the point estimate using a line type different from that used for the upper and lower limits.

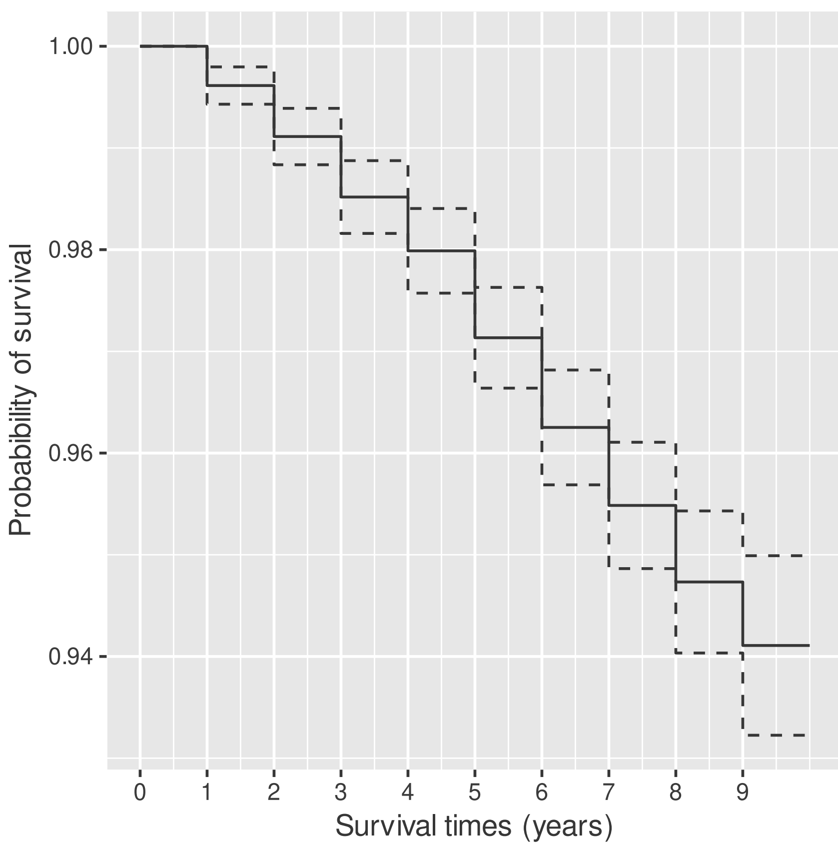


FIGURE 5.2
Replication of figure 5.4

5.2 Kaplan Meier estimation

Example 5.6 (page 182)

Though there are examples that use actuarial methods that we haven't covered yet, we will turn our attention to Kaplan Meier estimations now. The package to do survival analysis in R is `survival`. It is a recommended package which means that it should be installed already. If not, remember to install it using `install.packages` as detailed in the first chapter. First, we read in the relevant data.

```
read_table("./K11828 supplements/Datasets/Example 5.9.DAT",
           col_names=c("age", "tenure", "chd", "survival"),
           col_types = cols(
             age = col_number(),
             tenure = col_factor(),
             chd = col_factor(),
             survival = col_number())) -> data_5_9
```

```
library(survival)
Surv(data_5_9$survival,data_5_9$chd=="1")
```

Note that *tenure* and *chd* are both specified as factors while importing. An essential step in doing survival analysis is the construction of **survival object**. This is done by the function `Surv`. For right censored data, as is our case, it requires two arguments. The first one named `time` is the follow-up time. The second argument named `event` is either numerical or logical. If numerical, it can have two possible values, either `0` and `1` to indicate censored and event status respectively or `1` and `2` to indicate censored and event status respectively. If logical, `TRUE` will stand for event and `FALSE` for censored status. We specified *chd* as a factor. So, we need to use `chd=="1"` to convert it to logical values.

Printing a survival object results in something similar to that given in example 5.6. Instead of printing all 4402 values, we can limit our print to the first year as in the example, with the following code.

```
subset(data_5_9, survival < 365) |>
  with(Surv(survival,chd=="1")) |>
  sort()
```

```
 [1]   1   46   91+ 101  101  103  119  133+ 137  145+ 156  186+ 208  215
[15] 235  242  251  294  299  300  309+ 312  336+ 357+
```

We use `subset` function to restrict the data to the first year. This is achieved by specifying `survival < 365`. The result of `subset(data_5_9, survival < 365)` is the first argument to `with` which it receives through the pipe. The second argument to `with` is `Surv(survival,chd=='1')`. The function `with` helps us to specify the columns of a dataframe without prefixing the dataframe's name before each mention of the various columns. In other words, it evaluates its second argument `expr` in a local environment constructed from its first argument `data`. The entire result is sorted by `sort` for printing according to the sequence in which the event or censoring occurred.

Table 5.5, 5.6 (page 183)

```
survfit(Surv(survival,chd=="1") ~ 1 ,data =data_5_9) -> sf_5_9
```

The function `survfit` constructs survival curves. It doesn't plot the graph, but prepares the data. Its first argument is a **formula**, the left-hand side of which is a survival object. Formulas are R's symbolic way to represent model specifications. It consists of three parts – the `~` operator, the response to the left of `~` and model terms to the right of `~`. In our case, we want to construct a single curve for the entire data. Hence, the model term is `1`. Note that we don't prefix the dataframe's name to the arguments supplied to `Surv` as `survfit` accepts a `data` argument. We store the survival curve with the variable name *sf_5_9*. We now use the survival curve to print table 5.5 and 5.6, as well as to plot the Kaplan Meier curves.

```
summary(sf_5_9)
summary(sf_5_9, times= (1:9)*365.25,scale=365.25) -> sf_5_9_yrs
```

The function `summary` is a **generic function**. Generic functions can be thought of as a common name for a group of functions; the specific function that is called automatically depends on the class of data that is passed on. When we pass on a survival curve, the actual function that does the work is `summary.survfit`. With just a survival curve as its argument, `summary` will print the details similar to that in table 5.5. When we supply a `times` argument, it will return the same info, for the time points specified in the vector. Note that the `n.event` returned in this circumstance is the cumulative number of events **since** last time until the current time rather than **at** the current time. The `scale` argument is used to display the survival period. In our example, the value 365.25 is used to display the survival period in years instead of days. Thus we get the columns relevant to Kaplan Meier analysis of table 5.6.

Figure 5.8 (page 184)

```
plot(sf_5_9, ylim = c(1, 0.92), xscale = 365.25)
```

To plot the Kaplan Meier curve, we pass the survfit object to `plot`. `plot` is another generic function. For survfit objects, `plot` accepts `xscale` and `ylim` arguments amongst others. The `ylim` argument sets the limits of the y-axis displayed. We use it here to limit the display between 1 and 0.92, instead of the default 1 to 0. The `xscale` argument rescales the x-axis to display the survival period in years.

TABLE 5.1
Replication of table 5.6

		Survival Probability				
Time (years)	Number at risk	Events	KM estimate	KM std error	Upper 95% CI	Lower 95% CI
1	4378	17	0.99614	9.3541	0.99430	0.99797
2	4344	22	0.99113	14.1481	0.98836	0.99390
3	4294	26	0.98518	18.2493	0.98161	0.98876
4	4252	23	0.97989	21.2263	0.97573	0.98405
5	4194	37	0.97134	25.2672	0.96640	0.97630
6	4141	38	0.96252	28.8006	0.95690	0.96819
7	3609	31	0.95498	31.6077	0.94881	0.96120
8	1447	20	0.94714	36.3086	0.94005	0.95429
9	66	5	0.93661	70.0043	0.92299	0.95043

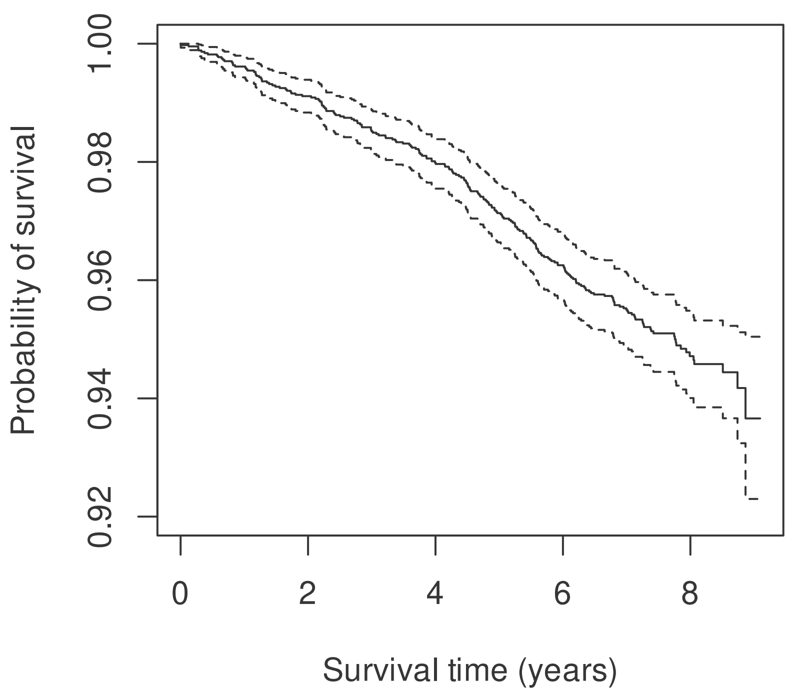


FIGURE 5.3
Replication of figure 5.8

5.3 Comparison of two sets of survival probabilities

5.3.1 Mantel Haenszel method

Example 5.7 (page 184)

To use Mantel Haenszel method for calculating common odds ratio across the various intervals given in table 5.7 we can use `mantelhaen.test`. We can type the data to create the three-dimensional array with the two-by-two tables that we feed to `mantelhaen.test`. Here, we collect the summary prepared by `survfit` and reshape it, primarily to show array manipulation.

```
summary(survfit(Surv(survival,chd=="1") ~ tenure, data =data_5_9),
        times= (1:9)*365.25,
        scale=365.25) -> sf_5_7
```

We save the results of `summary` of `survfit` with the name *sf_5_7*. It is a list with multiple components. See its structure by passing it as the argument to `str`.

```
cbind(rbind(matrix(sf_5_7$n.event, ncol = 2, byrow = FALSE), c(0,0)),
      rbind(sf_5_7$n, matrix(sf_5_7$n.risk, ncol = 2, byrow = FALSE)) -
        rbind(matrix(sf_5_7$n.event, ncol = 2, byrow = FALSE),c(0,0))
      ) -> tbl_5_7
```

We now bind the different components to make *tbl_5_7*. The component `n.event` is a vector containing the number of events, where the values for each of *tenure* is joined end to end. We convert this to a matrix with two columns, column-wise. We add a row to this two column matrix with both values set to zero using `rbind`. Similarly, the at risk population component `n.risk` is also converted to a two column matrix. The at risk population is calculated at the end of the time period. As we want the at risk population value at the beginning of the period, we add the numbers with which the study starts available as `n` as a row at the top. This will shift every row of the at risk data one step down, making them valid at the start of the period. We subtract from this the number of events to get the number without the events. We bind these columns together using `cbind` to get a four column matrix. Do print *tbl_5_7* to see the arrangement of the data.

```
matrix(c(rep(1:10, each = 4), rep(c(2,1,4,3), times = 10)),
       ncol = 2,
       byrow = FALSE) -> indx
```

Now, we have all the data in table 5.9. But, it is a matrix and not a three dimensional array of two-by-two tables. We need to rearrange the data to get the desired array. To achieve this, we first create an **index matrix**. An index matrix is a matrix with as many columns as there are dimensions in the array we want to subset. As we are going to subset a matrix which is an array with two dimensions, our index matrix will have two columns. When we supply an index matrix for subsetting, the value in each column will decide the element that will be selected from the array that is being subset – the value in the first column representing the index along the first dimension, the value in the second column determining the index of the second dimension and so on. The selected elements will be in the order specified by the index matrix. So, we will specify all the elements in the order we want. Thus through subsetting we will have rearranged the matrix in the order we want.

We build the index matrix *indx* using two calls of `rep`. The first call repeats the numbers 1 to 10 four times each. Thus, we will have four repetitions of 1 followed by four repetitions of 2 etc. The second `rep` repeats the vector `c(2,1,4,3)` 10 times. Thus we will get the specified sequence one after the other ten times. This sequence of vectors is converted to a two column matrix using `matrix`. Print the index matrix and see its value.

```
array(tbl_5_7[indx], dim = (c(2,2,10))) -> tbl_5_9
mantelhaen.test(tbl_5_9, correct = TRUE)
```

Now, we use `array` to build the argument for `mantelhaen.test` from the vector made by subsetting the *tbl_5_7* on the *indx* matrix.

```

    Mantel-Haenszel chi-squared test with continuity correction

data:  tbl_5_9
Mantel-Haenszel X-squared = 7.51, df = 1, p-value = 0.0061
alternative hypothesis: true common odds ratio is not equal to 1
95 percent confidence interval:
 1.1198 1.9066
sample estimates:
common odds ratio
           1.4612
```

5.3.2 The log-rank test

Example 5.8 (page 186)

The function provided by `survival` to test if two or more survival probabilities are different is `survdiff`.

```
survdiff(Surv(survival,chd=="1") ~ tenure,
          data =data_5_9,
          rho = 0,
          subset = survival < 365.25)
```

The first argument to `survdiff` is a formula, the left-hand side of which is a survival object. The right-hand side of the formula should consist of the predictors, the columns by which the different groups are determined. In our case, we use *tenure*, to mean that we want the survival probabilities to be compared between the different group of housing tenure. The `rho` argument is used to specify the actual test used, `0` to signify log-rank test. The `subset` argument is used to restrict the comparison to the first year.

```
Call:
survdiff(formula = Surv(survival, chd == "1") ~ tenure, data = data_5_9,
    subset = survival < 365.25, rho = 0)

          N Observed Expected (O-E)^2/E (O-E)^2/V
```

```
tenure=1 10        8     7.05    0.1289     0.223
tenure=2 14        9     9.95    0.0913     0.223

 Chisq= 0.2  on 1 degrees of freedom, p= 0.6
```

Though the statistic calculated by `survdiff` (0.2234) doesn't have the textbook's exact value (0.2850) , the p value 0.6365 results in the same interpretation. Possible reason for the difference is that the textbook value is derived with continuity correction.

5.3.3 Weighted log rank test

Example 5.9 (page 188)

The weighted logrank test is also done using `survdiff`. The `rho` argument is changed to less than zero to give weight to the later part of data, while values more than 0 gives more weight to the initial part of data. According to the help documents of `survival`, specifying `rho` as 1 is equivalent to the Peto & Peto modification of the Gehan-Wilcoxon test.

```
survdiff(Surv(survival,chd=='1') ~ tenure ,
         data =data_5_9,
         rho = 1)
```

The weighted log-rank test too returns a value similar to that in the textbook.

```
Call:
survdiff(formula = Surv(survival, chd == "1") ~ tenure, data = data_5_9,
    rho = 1)

            N Observed Expected (O-E)^2/E (O-E)^2/V
tenure=1 2482      101    121.3      3.27      7.78
tenure=2 1920      112     92.2      4.31      7.78

 Chisq= 7.8  on 1 degrees of freedom, p= 0.005
```

Figure 5.9 (page 189)

To plot the graph of figure 5.9, we supply `plot` with a `survfit` object with the appropriate formula.

```
plot(survfit(formula = Surv(survival, chd == "1") ~ tenure,
             data = data_5_9),
     ylim = c(1, 0.91),
     xscale = 365.25,
     xlab = "Survival time (years)",
     ylab = "Probability of survival")
```

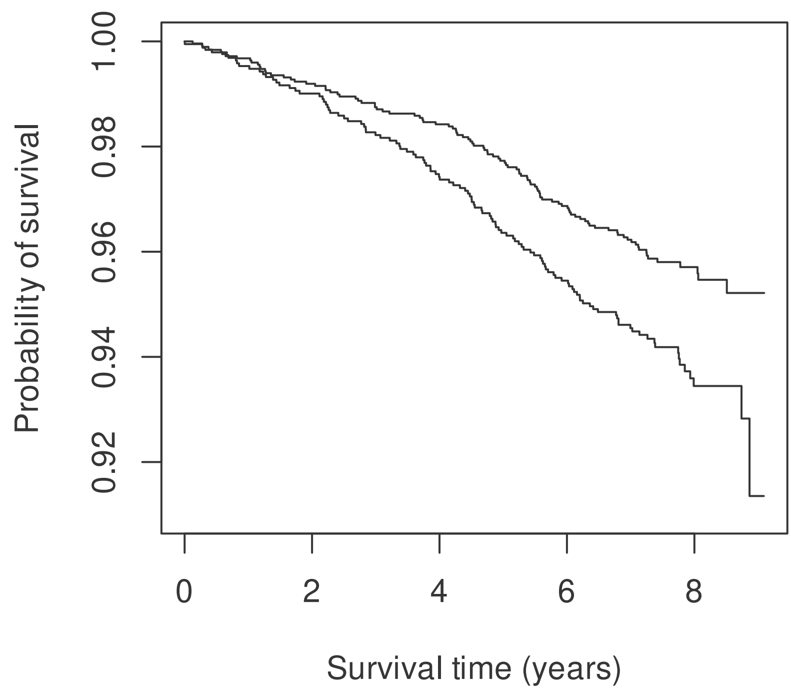


FIGURE 5.4
Replication of figure 5.9

5.4 Competing risk

Table 5.11 (page 192)

The same functions that were used for simple Kaplan Meier estimations can be used for competing risk analysis. The data, however, needs to be presented differently. The `events` column needs to be a factor and the first level of the factor should indicate censoring. Analysis is better done using individual level data rather than grouped data given in table 5.11. So, to replicate the analysis of table 5.11, we need to produce individual level data from the grouped data. Our data should thus have 4402 rows and two columns – one to indicate the outcome, either survival, death or chd and one to indicate the time when the outcome occurred.

```
c(1,46,91,101,103,119,133,137,145,156,186,208,
  215,235,242,251,294,299,300,309,312,336,357) -> time
c(50,50,0,100,50,50,0,50,0,50,0,50,
  50,50,50,50,50,50,50,0,50,0,0) -> chd
rep(0,23) -> dead
dead[c(3,7,9,11,20,22,23)] <- 60
```

```
rep(0,23) -> censor
censor[c(3,7,9,11,20,22,23)] <- 40
c(rep(time, chd),
  rep(time, dead),
  rep(time, censor),
  rep(365, 2852)) -> period
c(rep("chd", sum(chd)),
  rep("dead", sum(dead)),
  rep("censor", sum(censor) + 2852)) -> outcome
```

First, we input the data in time, chd, death and censored columns of the table into individual vectors. While *time* and *chd* are built obviously, *dead* and *censor* is built in two steps to reduce typing. First all the elements of both the vectors are set to zero using `rep`. In the second step, we use an index vector to subset those elements of the vectors *dead* and *censor* which have a value other than zero and assign them with the different value.

We now build *period* by repeating *time* as many times as *chd*, *dead* and *censor*. We also add the time 365 as many times as the number at the end of 365 days. Similarly, we build *outcome*, a character vector with the value "chd" repeated as many times as the total of *chd*, with value "dead" repeated as many times as the total of *dead* and with value *censor* repeated as many times the total of *censor*. To its end we add the value "censor" as many times as the number at the end of 365 days. Thus we get two vectors, one with the time periods and one with the outcome labels corresponding to the time periods. We should get the data we started with if we cross tabulate the vectors using `table`. Do print the vectors at each of the intermediate steps to see how they are being built. Also see the result of the final cross tabulation.

```
table(period, outcome)
```

We now use `Surv` and `survfit` to do the competing risk analysis.

```
survfit(Surv(period, factor(outcome)) ~ 1) -> sf_5_11
summary(sf_5_11)
```

Our call to `survfit` is, as in earlier examples, a formula with a survival object on its left-hand side. The right-hand side is `1` because we are not using any predictors. Inside `Surv`, the `event` argument is specified as `factor(outcome)` to coerce our character vector to factor. Remember the requirement that the first level of the `event` argument should indicate censoring. The default order of levels of a factor is alphabetical. So, our requirement is taken care of without explicitly stating the order of levels in the factor. Also, we haven't supplied `data` argument to `survfit` as the two vectors are not joined together as a dataframe.

TABLE 5.2
Replication of table 5.11

Time	Number at risk	CHD	Died	Censored	Cumulative joint surv. ($s^{(J)}$)	Cumulative failure ($f^{(l)}$)
1	4402	50	0	0	0.98864	0.011358
46	4352	50	0	0	0.97728	0.022717
91	4302	0	60	40	0.96365	0.022717
101	4202	100	0	0	0.94072	0.045650
103	4102	50	0	0	0.92925	0.057117
119	4052	50	0	0	0.91779	0.068583
133	4002	0	60	40	0.90403	0.068583
137	3902	50	0	0	0.89244	0.080167
145	3852	0	60	40	0.87854	0.080167
156	3752	50	0	0	0.86683	0.091875
186	3702	0	60	40	0.85278	0.091875
208	3602	50	0	0	0.84095	0.103713
215	3552	50	0	0	0.82911	0.115550
235	3502	50	0	0	0.81727	0.127388
242	3452	50	0	0	0.80543	0.139226
251	3402	50	0	0	0.79360	0.151063
294	3352	50	0	0	0.78176	0.162901
299	3302	50	0	0	0.76992	0.174739
300	3252	50	0	0	0.75808	0.186576
309	3202	0	60	40	0.74388	0.186576
312	3102	50	0	0	0.73189	0.198567
336	3052	0	60	40	0.71750	0.198567
357	2952	0	60	40	0.70292	0.198567

The cumulative joint survival given in table 5.11 is shown under P((s0)) printed by `summary` and cumulative failure under P(chd).

5.5 The person-years method

Example 5.10 (page 194)

We now turn to person years analyses. Until now, we dealt with periods directly. The examples here require us to calculate the periods from given dates. A `tidyverse` package `lubridate` makes it easy to handle dates.

```
library(lubridate)
dmy(c("5 Oct 1962","10 Oct 1975","10 June 1985","30 Aug 1990",
      "8 May 1968","1 Nov 1972","21 Mar 1960","8 June 1967")) -> entry
dmy(c("1 Dec 1999","31 Dec 2003","31 Dec 2003","28 Sep 2000",
      "8 Jul 1997","10 May 1985","30 Jun 1997","29 Jul 1971")) -> exit
```

```
c(TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE, TRUE) -> death
pyears(Surv(difftime(time1 = exit, time2 = entry,unit = "days"),
            death) ~ 1) -> result
poisson.test(result$event, result$pyears)
```

We input the dates of entry into study, exit from study and the outcome from table 5.12 into suitably named vectors. The vectors for the dates are typed in as text within quotes. This character vector is then passed onto `dmy` which converts it to date assuming that it is specified in the order date, month, year. There are other functions like `ymd`, `ydm`, `mdy`, `myd`, `dym` and many others that handle date specifications with missing date components or additional time components. Outcome is input as a logical vector. We then use `difftime` to calculate the number of days between the dates. Note that the later date is supplied as the first argument `time1`. The function `difftime` will return the number of `unit` between the two periods. We have asked for `days`. This function is the first argument to `Surv`, the `event` argument being *death*. We supply this survival object as the left-hand side of the formula argument to `pyears`, the function from `survival` that is used for person years calculations. The result of `pyears` is stored in *result*. We calculate the rate and its confidence interval using `poisson.test` by passing on to it the `event` and `pyears` components of *result*.

```
    Exact Poisson test

data:  result$event time base: result$pyears
number of events = 4, time base = 177, p-value <2e-16
alternative hypothesis: true event rate is not equal to 1
95 percent confidence interval:
 0.0061533 0.0578230
sample estimates:
event rate
  0.022584
```

5.5.1 Age specific rates

Example 5.11 (page 195)

To rework example 5.11, we need to enter the date of birth given in table 5.13 first.

```
dmy(c("21 Jul 1935","1 Aug 1939","8 June 1957","17 June 1950",
      "3 Jan 1937","14 May 1942","30 June 1932","10 Aug 1932")) -> dob
```

We use `dmy` to convert the character vector carrying the date of birth information to dates and store it under the name *dob*. The next step is to create the specified age groups. For this, we use the `tcut` function from `survival`.

```
tcut(entry - dob,
```

```
     c(0, 40,55,110)*365.25,
     labels = c('< 40', '40-54', '55+')) -> agegrp
```

The first argument to `tcut` is the age at entry calculated by subtracting *dob* from *entry*. The second argument is the break points we desire. As age is in days, we need the `breaks` too in days. Hence, we multiply our desired break points with 365.25. Note that we supply break points for the lower end of the first group (0) and the higher end of last group (110). Finally, we supply `labels` for the groups. As the groups created are one less than the break points specified, we need to specify only three labels.

```
pyears(Surv(exit - entry, death) ~ agegrp,
       data.frame = TRUE)$data -> py_5_13
```

Now, we use `pyears` supplying it with a formula. The left-hand side of the formula is a survival object. Its `time` argument is the period of follow-up, calculated by subtracting *entry* from *exit* and its `event` argument is *death*. The right-hand side of the formula is *agegrp* we prepared in the previous step. We also specify `data.frame = TRUE` so that we get the result arranged as a data frame. We assign `data`, the dataframe returned by the function to *py_5_13*. If we examine this object, we will see that it contains the person years and events calculated for each age group. We need to extend this result by calculating the rate for each age group.

```
mutate(py_5_13,
       rate = map2_dbl(event,
                       pyears,
                       function(x,y) {poisson.test(x,y)$estimate}))
```

We use `mutate` from `dplyr`, a member of `tidyverse`. This function creates or modifies columns of a data frame. We need to supply the dataframe which needs to be modified. Additional arguments are pairs of column names and the values that they must carry. We are creating only one column *rate*. This column is assigned a value returned by `map2_dbl`, a function from `purrr`, part of tidyverse. `map2_dbl` accepts two vectors, and performs the third argument, a function using their values in parallel. We supply an anonymous function to `map2dbl`, which accepts two arguments, calls `poission.test` with those two arguments and returns the `estimate`. Thus, each pair of *events* and *pyears* from *py_5_13* is passed on to `poisson.test` and the `estimate` from the result gets stored in a new column named *rate*.

```
  agegrp pyears n event     rate
1   < 40 63.198 7     1 0.015823
2  40-54 79.633 7     1 0.012558
3    55+ 34.289 4     2 0.058328
```

5.5.2 Summarisation of rates

Example 5.12 (page 196)

To calculate the standardised mortality ratio, we use the function `epi.indirectadj`, which we have seen already. As this function expects matrices as its arguments, we convert the relevant columns from *py_5_13* using `matrix`.

```
library(epiR)
epi.indirectadj(obs = matrix(py_5_13$event,
                             nrow = 1,
                             byrow = TRUE,
                             dimnames = list(row ="",age = py_5_13$agegrp)),
                pop = matrix(py_5_13$pyears,
                             nrow = 1,
                             byrow = TRUE,
                             dimnames = list(row ="",age = py_5_13$agegrp)),
                std = matrix(c(1.8,9,19.2)/1000,
                             nrow = 1,
                             byrow = TRUE),units = 1) -> smr_5_12
```

The `obs` argument is derived from *event* column of *py_5_13* and the `pop` argument from *pyears*. The `std` is supplied as given in the textbook, but divided by 1000. All arguments have one row only. The `dimnames` argument is a list with two components – the first being the name for the row, which is needed though there is only one row. The columns are named using the *agegrp* column. We need the `smr` component of the result, which provides the expected number of deaths, the SMR and its confidence interval.

```
smr_5_12$smr
```

```
 obs    exp    est   lower  upper
   4 1.4888 2.6867 0.67168 5.3735
```

5.5.3 Comparison of two SERs

Example 5.13 (page 198)

To calculate relative SER, we need a new package `popEpi`. Remember to install it using `install.packages` as discussed in chapter 1. We use the function `sir_ratio` to compute the relative SER.

```
library(popEpi)
sir_ratio(c(43,38.755), c(4,1.488))
```

The function requires two numeric vectors, each with the number of events and the person times for the two groups that are to be compared.

```
sir_ratio     lower     upper
    0.413     0.150     1.583
```

We get the relative SER and its confidence interval.

As I couldn't find a readymade function to test the hypothesis that relative SER is 1, we will do it by hand.

```
events <- c(4,43)
xpctd <- c(1.488,38.755)
xpctdn <- xpctd * sum(events) / sum(xpctd)
sum(((abs(events - xpctdn) - 0.5 ) ^2) /  xpctdn) -> ststc
ststc
pchisq(ststc,1,lower.tail = FALSE)
```

We enter the number of deaths observed and expected into two vectors, then multiply the overall SER with expected which is saved as *xpctdn*. It is used to calculate the test statistic using the formula given in the textbook. The p value for this statistic is calculated using `pchisq`.

```
[1] 1.8554
[1] 0.17315
```

5.5.4 Mantel-Haenszel methods

Example 5.14 (page 201)

To re-work example 5.14, we will be using `epi.2by2`. We input the data in table 5.14 in an array suitable for use by `epi.2by2`.

```
array(c(2,3,1107.447,1619.328,
        24,19,3058.986,4550.166,
        31,25,3506.530,4857.904,
        28,27,3756.650,4536.832,
        28,26,2419.622,2680.843,
        2,4,351.710,356.394),
      dim = c(2,2,6),
      dimnames = list(house = c("rented", "owned"),
                      data = c("event", "period"),
                      age = c("40-44", "45-49","50-54",
                              "55-59","60-64", "65-69"))) -> tbl_5_14
```

The third dimension of the array are the different levels of strata. In each strata, the two-by-two tables are arranged with events in the first column and person time in the second column. The first row corresponds to those exposed to risk factor, renters in our case and the second row to those unexposed, owners in our case. We pass the array to `epi.2by2` and store the `massoc.detail` component of the result. Note that we have used `method = "cohort.time"` to say that we are using person time analysis.

```
epi.2by2(tbl_5_14,method = "cohort.time")$massoc.detail -> result_5_14
```

We can extract the required results from this object. If we desire the relative rates for each strata, it is available as `IRR.strata.wald`. The overall crude relative rate is available in `IRR.crude.wald`. The Mantel Haenszel relative rate is available in `IRR.mh.wald`. The chi square test on MH estimate is available in `chi2.mh`. Here we print the MH estimate with its confidence interval.

```
result_5_14$IRR.mh.wald
```

```
     est  lower  upper
1 1.4074 1.0809 1.8327
```

The MH relative rates is not different from the textbook. The chi square statistic calculated (6.46) is slightly different from the textbook value (6.16), though the p value is similar. This is because `epi.2by2` does not use continuity correction.

As there aren't any examples under period cohort analysis that requires calculations, we end this chapter with a recap.

5.6 Recap

5.6.1 Concepts introduced in this chapter

- survival object
- formula
- generic function
- index matrix

5.6.2 Commands introduced in this chapter

- base::cumprod
- ggplot2::geom_step
- stats::qnorm
- ggplot2::scale_x_continuous
- survival::Surv
- survival::survfit
- base::with
- base::sort
- base::summary
- base::rbind
- survival::survdiff
- lubridate::dmy
- base::difftime
- survival::pyears
- survival::tcut
- purrr::map2_dbl
- popEpi::sir_ratio

6

Case-control studies

This chapter deals with case control studies. We will use the add-on packages `epiR` and `epiDisplay`. Remember to install them using `install.packages` as discussed in chapter 1. Matrices and vectors continue to be the predominant data types we use. We start with reworking example 6.3.

6.1 Basic methods of analysis

Example 6.3 (page 216)

```
library(epiR)
matrix(c(99,303,132,290),
       byrow = TRUE,
       nrow = 2,
       dimnames = list(sun = c("protected","unprotected"),
                       status = c("cases","controls"))) -> tbl_6_2
epi.2by2(tbl_6_2,
         method = "case.control")$massoc.detail -> result_6_3
```

We use `epi.2by2` function from `epiR`. We pass our data to the function as a two-by-two matrix and specify `case.control` as the `method`. The `massoc.detail` component of the result is stored.

```
result_6_3$OR.strata.wald
result_6_3$chi2.strata.yates
```

We use the `$` operator to get `OR.strata.wald` of this object to get the odds ratio and its confidence interval & the `chi2.strata.yates` to get the continuity corrected chi-square test's value and its probability.

```
      est   lower  upper
1 0.71782 0.52843 0.9751
```

```
  test.statistic df p.value.1s p.value.2s
1         4.1928  1   0.020298   0.040597
```

DOI: 10.1201/9781003589563-6

6.1.1 Polytomous exposure

Example 6.4 (page 217)

To rework the example 6.4, we need an array to represent the four two-by-two tables. As the unexposed row of all the two-by-two tables are the same, the straight forward way would be to repeat those values as required in the vector we supply to `array` command. However, we will follow a different path.

```
array(0,
      dim = c(2,2,4),
      dimnames = list(exposure = c("exposed", "base"),
                      status = c("cases", "controls"),
                      ethnicity = c("african","hispanic","asian","others"))
      ) -> tbl_6_4
tbl_6_4[1,,] <- c(25,25,13,5,32,21,20,37)
tbl_6_4[2,,] <- rep(c(514,541),4)
```

First, we build an array of suitable dimensions, but with a dummy value zero for all elements. In the second step, we subset the array to select all elements in the first row in all other dimensions and assign the correct values from table 6.4 to them. In the final step, we subset the second row in all other dimensions and assign to them the values in the first row of table 6.4 repeated four times. Thus we get four two-by-two tables, each with the same values in the second row. We now pass the array to `epi.2by2`.

```
epi.2by2(tbl_6_4,
         method = "case.control")$massoc.detail$OR.strata.wald
```

The odds ratio and its confidence intervals for each of the two-by-two tables is in the `OR.strata.wald` sub-component of the `massoc.detail` component of the result returned by `epi.2by2`.

```
      est   lower   upper
1 1.05253 0.59681 1.85624
2 2.73658 0.96876 7.73032
3 1.60385 0.91292 2.81770
4 0.56893 0.32589 0.99323
```

6.1.2 Attributable risk

Example 6.5 (page 219)

```
matrix(c(639,373,593,419),
       byrow = FALSE,
       nrow = 2,
       dimnames = list(psmoke = c("yes", "no"),
                       status = c("cases","controls"))) -> tbl_6_5
```

```
epi.2by2(tbl_6_5,
         method = "case.control",
         interpret = TRUE)$massoc.detail$PAFest.strata.wald
```

While the result returned by `epi.2by2` does include estimates of attributable fraction and population attributable fractions, they are named differently from the textbook. We use the `PAFest.strata.wald` to get the measure called attributable fraction in the textbook.

```
      est     lower  upper
1 0.10979 0.0072926 0.2017
```

While the point estimate is the same as in the textbook, the confidence interval is different, probably because a method different from the textbook is followed.

6.2 The analysis of matched studies

6.2.1 1:1 matching

Example 6.8 (page 233)

R doesn't really favour grouped data for matched analysis. However, it does have Mc Nemar's test, which we use here to re-work example 6.8.

```
matrix(c(6,25,12,66),
       byrow = TRUE,
       nrow = 2,
       dimnames = list(control = c("dementia", "no dementia"),
                       relative = c("dementia","no dementia"))) -> tbl_6_11
mcnemar.test(tbl_6_11)
```

Usage of `mcnemar.test` is rather straight forward. It is called with the two-by-two matrix. It returns the chi square value and its associated p value. Continuity correction is applied by default.

```
    McNemar's Chi-squared test with continuity correction

data:  tbl_6_11
McNemar's chi-squared = 3.89, df = 1, p-value = 0.049
```

The package `epiDisplay` provides `matchTab` function to calculate odds ratio for the matched cases. However, it does not accept grouped data. We need to expand the grouped data to recreate the individual level data.

```
library(epiDisplay)
rep(c(0,1), 109) -> status
rep(c(1: 109), each = 2) -> id
c(rep(1, 12),
  rep(c(1,0), 12),
  rep(c(0,1), 25),
  rep(0,132)) -> expose
matchTab(status, expose, id)
```

The `matchTab` expects three numeric vectors. One to denote case / control status, one to indicate the exposure status and one to indicate the identification number of each set. The first vector *status* is the pair *c(0,1)* repeated 109 times. The number 109 denotes the sum of all numbers in table 6.11, the total number of case control pairs. Thus we get alternate zero and ones for a total of 218. Zero stands for controls and one for cases. The vector *id* should carry the same number for each case control pair. We thus repeat the numbers one to 109, each twice. The length of the vector is again 218. The final vector *expose* is also numeric, with zero standing for unexposed (in our case, not having a relative with dementia) and one for exposed status. For the first six pairs, both cases and controls have a relative with dementia. So, we use *rep(1,12)*. For the next 12 pairs, the controls have a relative with dementia, cases don't have. So, we use *rep(c(1,0), 12)*. Remember that when building *status* we put `0` first, to indicate controls. Thus, their exposure status `1` should come first when building the *expose* vector. The *expose* vector is thus built by joining together the four `rep` results together. We finally call `matchTab` with the vectors we have built.

```
Exposure status: expose = 1

Total number of match sets in the tabulation = 109

Number of controls = 1
                    No. of controls exposed
No. of cases exposed  0  1
                   0 66 12
                   1 25  6

Odds ratio by Mantel-Haenszel method = 2.083

Odds ratio by maximum likelihood estimate (MLE) method = 2.083
 95%CI= 1.047 , 4.147
```

The confidence interval is calculated by a method different from the textbook, hence slightly different from the textbook value.

6.2.2 1:c matching

Example 6.9 (page 234)

Re-working the example 6.9 follows the same steps as in the previous example. Rather than following the calculations using the formula given in the textbook, we build appropriate vectors from the grouped data and feed it to `matchTab`.

```
rep(1:(57+54), each = 6) -> id
rep(c(1,0,0,0,0,0),times = 57+54) -> status
c(c(1,0,0,0,0,0),
  rep(c(0,0,0,0,0,0),11),
  rep(c(1,1,0,0,0,0), 5),
  rep(c(0,1,0,0,0,0),15),
  c(1,1,1,0,0,0),
  rep(c(0,1,1,0,0,0),11),
  rep(c(1,1,1,1,0,0),3),
  rep(c(0,1,1,1,0,0),5),
  rep(c(1,1,1,1,1,0),20),
  rep(c(0,1,1,1,1,0),7),
  rep(c(1,1,1,1,1,1),27),
  rep(c(0,1,1,1,1,1),5)) -> expose
matchTab(status, expose, id)
```

The result is similar to that in the textbook.

```
Exposure status: expose = 1

Total number of match sets in the tabulation = 111

Number of controls = 5
                    No. of controls exposed
No. of cases exposed  0  1  2  3  4  5
                   0 11 15 11  5  7  5
                   1  1  5  1  3 20 27

Odds ratio by Mantel-Haenszel method = 0.514

Odds ratio by maximum likelihood estimate (MLE) method = 0.519
 95%CI= 0.301 , 0.897
```

6.2.3 1: variable matching

Example 6.10 (page 241)

The function `matchTab` supports 1: variable matching. However, in the case of our example it returns an error due to low numbers. So, we will re-work example 6.10 “by hand” using the steps given in the textbook. First, we enter the data from table 6.18.

```
c(2,3,3,3) -> j
c(2,1,2,3) -> i
```

```
c(1,2,2,2) -> m
c(1,3,2,2) -> t
```

Next, we calculate the intermediate values for each of the rows.

```
i * t /(j+1) -> Ei
i * (j +1 - i) * t / (j + 1) ^2 -> Vi
(j + 1 - i) * m / (j +1) -> Ti
i * (t - m) / (j +1) -> Bi
```

From these we calculate the Mantel Haenszel odds ratio, the CMH test statistic and its probability given null hypothesis.

```
sum(Ti) / sum(Bi) -> mhor_6_10
(abs(sum(m) - sum(Ei)) - 0.5 ) ^ 2 / sum(Vi) -> cmh_6_10
mhor_6_10
cmh_6_10
pchisq(cmh_6_10,1, lower.tail = FALSE)
```

The function `abs` returns the absolute value of its argument. The `pchisq` returns the upper tail probability of finding the specified value (*cmh_6_10* in our case) in the chi squared distribution of the specified degrees of freedom (1 in our case).

```
[1] 13.333
[1] 4.0209
[1] 0.044939
```

6.2.4 Many: many matching

Example 6.11 (page 244)

The function `matchTab` doesn't support many:many matching. So, we will re-work example 6.11 similar to how we did the previous example.

```
c(rep(1,12), rep(2,4), 3,3) -> r
c(2,3,rep(4,9),rep(7,3),8,8,11,12) -> s
c(1,2,1,1,1,2,3,2,2,3,4,4,3,6,8,5,5,10) -> i
c(0,1,rep(0,5),rep(1,8),2,1,2) -> k
i * r / (r +s) -> Eik
( i *  r * s * (r + s - i)) /
  ((r + s) ^ 2 * (r + s -1)) -> Vik
k * (s - i  + k) / (r +s) -> Tik
(i - k) * (r - k) / (r +s) -> Bik
sum(Tik) / sum(Bik) -> mhor_6_11
```

```
(abs(sum(k) - sum(Eik)) - 0.5) ^ 2 / sum(Vik) -> cmh_6_11
cmh_6_11
pchisq(cmh_6_11,1, lower.tail = FALSE)
mhor_6_11
```

The steps are similar to the previous example and so are not elaborated.

```
[1] 0.092541
[1] 0.76097
[1] 1.2647
```

We will end this chapter here with a recall of the important points.

6.3 Recap

6.3.1 Commands introduced in this chapter

- stats::mcnemar.test
- epiDisplay::matchTab

7

Intervention studies

This chapter introduces intervention studies. The add-on packages we use include `readr`, `ggplot2`, `dplyr`, all part of tidyverse, `epiR` and `DescTools`.

Example 7.1 (page 257)

We start with t test, which we learned in a previous chapter. In example 7.1 of this chapter, we have a table with individual level data upon which we need to perform the t test. First, we import the given data.

```
library(tidyverse)
read_table("./K11828 supplements/Datasets/Table 7.1.DAT",
           col_names=c("treat","nvinitial","nvfinal","vinitial","vfinal"),
           col_types = cols(treat = col_factor(),
                            nvinitial = col_number(),
                            nvfinal = col_number(),
                            vinitial = col_number(),
                            vfinal = col_number())) -> data_7_1
mutate(data_7_1,
       nvdiff = nvfinal - nvinitial,
       vdiff = vfinal - vinitial) -> data_7_1
```

Note that the data provided is formed essentially by joining the two halves of table 7.1 one below the other. An additional column is also there, as marker to indicate the treatment group to which each row belongs. Except for *treat*, we specify all other columns as number. After the data is imported, we make two new columns *nvdiff* and *vdiff*, to calculate the difference between the two readings of each type of IQ tests.

```
t.test(nvdiff ~ treat, data = data_7_1)
t.test(vdiff ~ treat, data = data_7_1)
```

We now call `t.test` with a formula. The left-hand side of the formula is the variable on which t test is to be performed, the right-hand side is the variable by which the left-hand side variable should be grouped (*treat* in our case). We also specify the `data` argument to inform `t.test` that the variables that we specify in the formula are in *data_7_1* data frame.

```
    Welch Two Sample t-test
```

DOI: 10.1201/9781003589563-7

```
data:  nvdiff by treat
t = -1.24, df = 81.9, p-value = 0.22
alternative hypothesis: true difference in means between
      group 1 and group 2 is not equal to 0
95 percent confidence interval:
 -6.2543  1.4448
sample estimates:
mean in group 1 mean in group 2
         1.5000          3.9048

    Welch Two Sample t-test

data:  vdiff by treat
t = -0.364, df = 82.3, p-value = 0.72
alternative hypothesis: true difference in means between
      group 1 and group 2 is not equal to 0
95 percent confidence interval:
 -3.2749  2.2619
sample estimates:
mean in group 1 mean in group 2
         2.6364          3.1429
```

The sign of the statistic is different from the textbook based on which factor level is taken as the reference. This makes no difference to the interpretation.

7.1 Parallel group studies

7.1.1 Numbers needed to treat

Example 7.6 (page 269)

The function `epi.2by2` calculates NNT when we provide it the relevant two-by-two tables. Here, we use the data in example 7.6.

```
matrix(c(26, 1176 - 26, 48, 1176 - 48 ),
       nrow = 2,
       byrow = TRUE,
       dimnames = list(status = c("case", "control"),
                       pneumonia = c("yes","no"))) -> data_7_6
epi.2by2(data_7_6)$massoc.detail$NNT.strata.wald
```

NNT is returned in the `NNT.strata.wald` of `maasoc.detail` component of the result. Our result has a negative sign, indicating that "risk factor" (treatment in our case), reduces the risk of disease.

```
      est   lower   upper
1 -53.455 -216.54 -30.491
```

As NNT is calculated directly from numbers rather than from rates, adjusting the NNT is not possible directly. If we can algebraically manipulate the NNT formula given in the textbook to derive the extrapolated numbers rather than rates, we can feed those numbers to `epi.2by2` to obtain NNTs extrapolated to treatment duration other than the actual study duration. For our example, using the vector *c(30.89, 1176-30.89,56.92, 1176-56.92)* in place of the original will get us the NNT for five years.

7.2 Cross-over studies

Figure 7.4 (page 276)

To rework the example 7.9, we need the data in table 7.5. The data is provided in **long format** to mean that there is one row for one observation. Thus, for any individual, there are two rows of data – one for the first period and one for the second.

```
read_table("./K11828 supplements/Datasets/Example 9.20.DAT",
           col_names=c("id", "period","group", "med", "score"),
           col_types = cols(id = col_number(),
                            period = col_factor(),
                            group = col_factor(),
                            med = col_factor(),
                            score = col_number())) -> data_7_5
```

Now, we will try to reproduce the graph in figure 7.4. We need to calculate the mean pain score for each period and each treatment.

```
group_by(data_7_5,period, med) |>
  summarise(mps = mean(score)) -> data_7_5s
```

We use the function `group_by` to first group the data by both *period* and *med*. We then pipe the result using `|>` to the next function `summarise`. The first argument of `summarise` is thus received from the pipe. For each group, `summarise` calculates the value of a new field using the definition we provide. In our case, the field is named *mps* and its value is calculated by applying the function `mean` to the values in the *score* column. Thus, we get a new dataframe with three columns, two with the values of the columns we used for grouping and the third one *mps* with the mean pain score for that group. There will be as many rows as there are unique combination of values of the grouping variables.

```
ggplot(data_7_5s) +
  geom_line(aes(x=period,y = mps,group = med,colour = med,linetype = med)) +
  labs(x = "Period",
       y = "Mean pain score") +
  scale_linetype_manual(labels = c("Ibuprofen", "Aspergesic"),
                        values = c(1,2),
```

```
                      name = "Medicines") +
  scale_colour_manual(labels = c("Ibuprofen", "Aspergesic"),
                    values =  c( "#111111", "#004B73"),
                    name = "Medicines")
```

Then, we call `ggplot` with the data frame we prepared. We use the `geom_line` to draw lines between the mean pain score for each period. We ask `geom_line` to use different line type and colour for the different *med* groups. As we want the legend to use labels different from the values in the dataframe, we use `scale_linetype_manual` and `scale_colour_manual`. We use both the manual scales as we use `linetype` and `colour` as aesthetics. Both require `labels`, the values of which will be used for the legend labels, `values` which determines line type or colour used and `name` which determines the heading used for the legend.

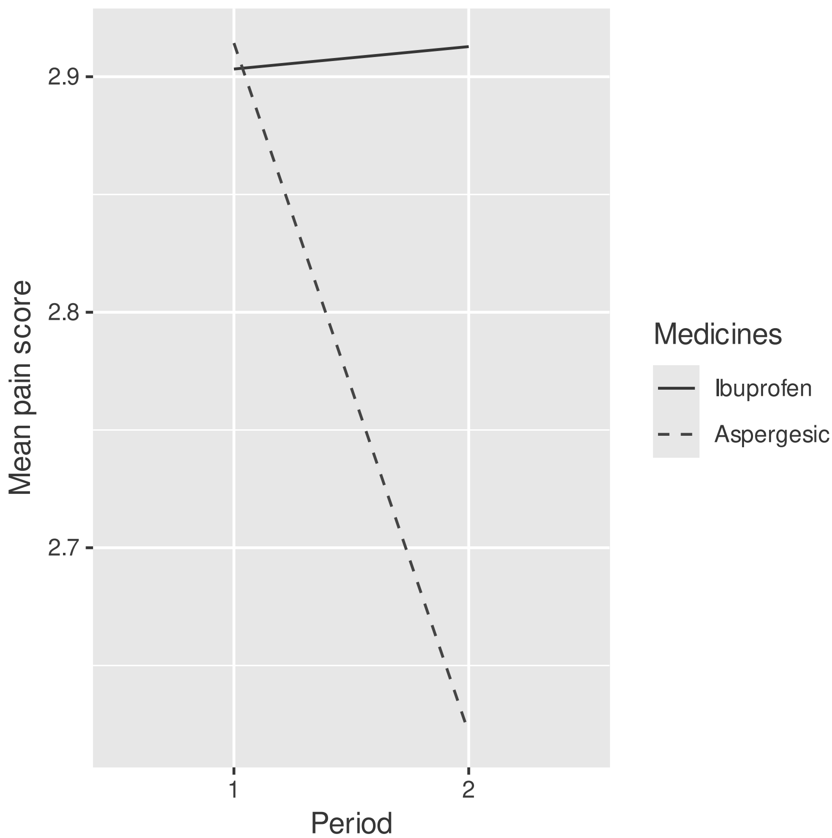


FIGURE 7.1
Replication of figure 7.4

Figure 7.5 (page 277)

To prepare the graph in figure 7.5, we need to reshape our data. We need to do this because `geom_segment` which we will be using to draw line segments for each individual, wants the data to draw both points in one row of the data frame. We use `pivot_wider`.

```
pivot_wider(data_7_5,
            id_cols =  c(id,group),
            names_from = period,
            names_prefix = "score",
            values_from = score) -> data_7_5w
```

Our aim is to make two columns for each individual – one for the first pain score and the other for the second pain score. The `id_cols` specify which set of columns uniquely identify a row of observation. Though *id* is sufficient to make our rows unique, we include *group* too as we need the info in *group* at a later stage. If, we don't include *group*, we will not have the info in that column for later use. The option `names_from` determines the name of the new columns that will be created. As many columns as there are unique values in the column specified for this option will be created. The columns will be named with the unique values prefixing the string we specify for `names_prefix`. What will be the values under these columns? The values will be sourced from the column we specify for `values_from`. The end result for us is a wider data frame with as many rows as are dictated by the combination of values of *id* and *group*, with two new columns *score1* and *score2*, the first column carrying the value from *score* when the *period* has the value 1 and the second one with the value from *score* when the period has the value 2.

```
levels(data_7_5w$group) <- c("Ibuprofen-Aspergesic","Aspergesic-Ibuprofen")
```

We now change the labels associated with the codes used in *group*. Remember that *group* was imported as factor. A factor variable uses numerical codes to represent categorical values. We can change the labels used for each of the codes of a factor variable by using `levels` providing it with the factor vector's name. Note that `levels` is assigned a set of values by using the `<-` assignment operator. In our example, the value 1 will get associated with "Ibuprofen-Aspergesic" and 2 with "Aspergesic-Ibuprofen" when printed.

```
ggplot(data_7_5w) +
  geom_segment(aes(x = "1", y = score1,xend = "2", yend = score2)) +
  facet_wrap(c("group")) +
  labs(y = "Mean pain score", x= "Period")
```

We now draw the widened data frame using `ggplot`. We use `geom_segment`. The first set of points is specified with `x` and `y`. We use a string "1" for `x` for all the rows of data, and `y` is sourced from *score1*. The second set of points is specified with `xend`, for which we specify a common "2" and `yend` which is sourced from `score2`. We need two graphs, one for each value of group. This is achieved by `facet_wrap`, to which we supply *group* as a character vector. Note how `facet_wrap` uses the labels attached with the factor as headers.

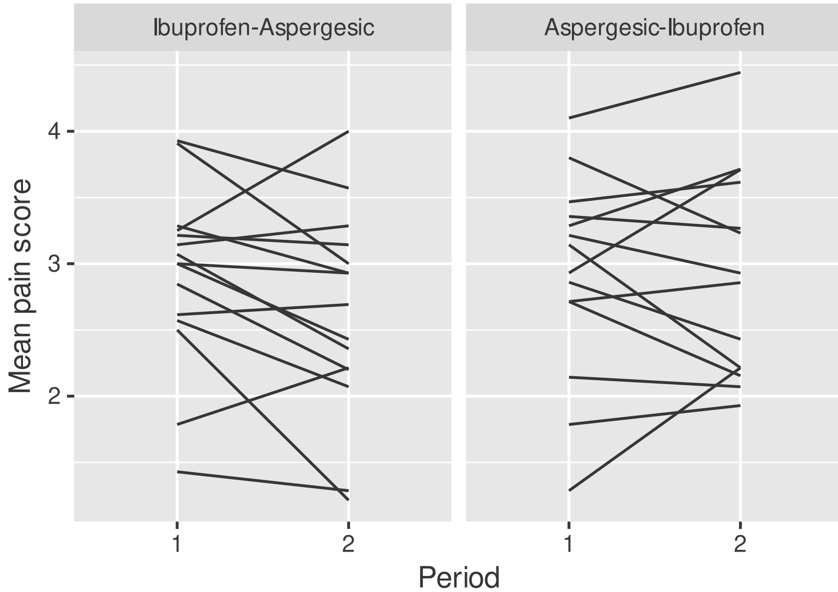


Figure 7.6 (page 277)

We will now use the same data frame we prepared for the earlier graph to prepare the graph in figure 7.6.

```
ggplot() +
  geom_point(data = data_7_5w,
             mapping = aes(x=score1,y= score2,colour= group,shape= group),
             show.legend = FALSE) +
  geom_line(aes(x = 1:5, y=1:5), linetype = 2) +
  scale_shape_manual(values = c(1,20)) +
  scale_colour_manual(values = c( "#111111", "#004B73")) +
  labs(x = "Period 1",
       y = "Period 2")
```

We use `ggplot`, but doesn't provide it with a data argument. We do so because the two geoms we use require two different sets of data. For `geom_point`, we specify the `data` argument as the wider data frame we prepared. We ask it to use *score1* and *score2* to determine the x y location of the point. We use `shape` and `colour` arguments to specify that we want different colours and shapes for each of the different values of *group*. We provide `show.legend = FALSE` to say that we don't want a legend to be printed. We use `geom_line` to draw the central guideline. The `x` and `y` arguments for this geom are the same, a sequence from 1 to 5. The `scale_shape_manual` is used to restrict the shapes used for drawing the points. The values we selected restrict the shapes to the filled and empty circles. If we don't specify that, we will get the shapes represented by 1 and 2 – circle and triangle.

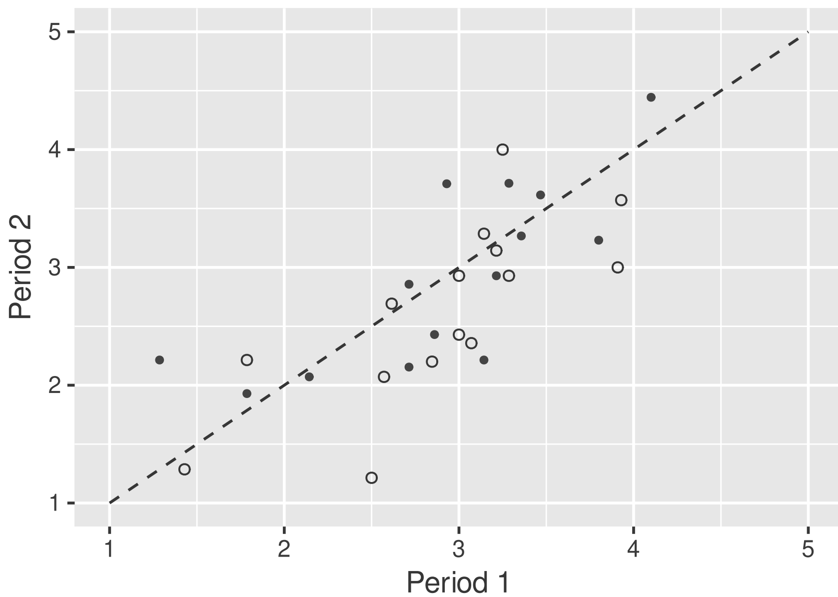


FIGURE 7.2
Replication of figure 7.6

Example 7.9 (page 279)

We now turn to the hypothesis tests. We need to modify our dataframe to calculate the total of and difference between the pain scores.

```
mutate(data_7_5w,
       total = score1 + score2,
       diff = score1-score2) -> table_7_5
```

We use the function `mutate` and supply the dataframe we prepared earlier as its first argument. We provide names and specification for the new columns we need. The column *total* is defined as the sum of *score1* and *score2* and the column *diff* as the difference between *score1* and *sore2*. The modified dataframe is stored as *table_7_5*.

```
t.test(total ~ group, data = table_7_5, var.equal = TRUE)
```

We use `t.test` to test for treatment by period interaction. The first argument is a formula, the left-hand side of which is the *total* column that was calculated in the previous step. The right-hand side of the formula is *group*. Thus, we are instructing `t.test` to test for differences in the mean of *total* in the two groups defined by the value of *group*.

```
    Two Sample t-test

data:  total by group
t = -0.583, df = 27, p-value = 0.56
alternative hypothesis: true difference in means between
```

```
      group Ibuprofen-Aspergesic and
      group Aspergesic-Ibuprofen is not equal to 0
95 percent confidence interval:
 -1.36733  0.76252
sample estimates:
mean in group Ibuprofen-Aspergesic mean in group Aspergesic-Ibuprofen
                            5.5247                             5.8271
```

```
t.test(diff ~ group, data = table_7_5, var.equal = TRUE)
```

Similarly, we test for treatment effect, by using *diff* in place of *total*.

```
    Two Sample t-test

data:  diff by group
t = 1.43, df = 27, p-value = 0.16
alternative hypothesis: true difference in means between
      group Ibuprofen-Aspergesic and
      group Aspergesic-Ibuprofen is not equal to 0
95 percent confidence interval:
 -0.12052  0.68125
sample estimates:
mean in group Ibuprofen-Aspergesic mean in group Aspergesic-Ibuprofen
                           0.28187                            0.00150
```

Testing for period effect is slightly different.

```
t.test(subset(table_7_5, group == "Ibuprofen-Aspergesic")$diff ,
       -1 *(subset(table_7_5,group == "Aspergesic-Ibuprofen")$diff),
       var.equal = TRUE)
```

Here, we use `t.test`; but, we don't give it a formula. Instead, we supply it with two vectors corresponding to the two groups. We use `subset` to select only some of the records of the dataframe. The first argument to `subset` is the data frame we want to subset and the second argument is the criteria for selecting the records. In our case, for the first vector, we ask it to select only those records having the value *Ibuprofen-Aspergesic* in the column *group* and then select only the *diff* column of those records. For the second vector, we select only those records with the value *Aspergesic-Ibuprofen* in the column *group*. We select the *diff* column of those records and multiply it with `-1`.

```
    Two Sample t-test

data:  subset(table_7_5,
     group == "Ibuprofen-Aspergesic")$diff and
     -1 * (subset(table_7_5, group == "Aspergesic-Ibuprofen")$diff)
t = 1.45, df = 27, p-value = 0.16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.11752  0.68425
```

```
sample estimates:
mean of x mean of y
  0.28187  -0.00150
```

We have no direct way to calculate the mean difference in pain score and its confidence interval. However, calculating it is not difficult. First, we need to store the result of the relevant t test.

```
t.test(diff ~ group,
       data = table_7_5,
       var.equal = TRUE) -> tdiff
(tdiff$estimate[1] - tdiff$estimate[2]) * 0.5
```

The mean of the difference in pain score when using aspergesic and ibuprofen is calculated as half of the difference between the mean for the two groups returned by `t.test` in its `estimate` component.

```
(tdiff$estimate[1] - tdiff$estimate[2]) * 0.5  +
  (0.5 * qt(c(0.05/2, 1-0.05/2), df =tdiff$parameter) * tdiff$stderr)
```

The standard error is returned by `t.test` in the component `stderr`. We multiply the standard error with the appropriate t value returned by `qt`. The function `qt` requires two parameters. The first one is the probability for which t value is required. We provide a vector with two values, one for the lower tail (0.05/2) and one for the upper tail (1-0.05/2). The second argument is the `df`, the degrees of freedom which we collect from the `parameter` component of the result returned by `t.test`. We add this to the mean of differences calculated in the previous step to get the confidence interval.

```
mean in group Ibuprofen-Aspergesic
                           0.14018
[1] -0.060258  0.340625
```

The steps for calculating the confidence interval for period differences is similar except for the t test used.

```
t.test(subset(table_7_5,group == "Ibuprofen-Aspergesic")$diff,
       -1 *(subset(table_7_5,group == "Aspergesic-Ibuprofen")$diff),
       var.equal = TRUE) -> pdiff
(pdiff$estimate[1] - pdiff$estimate[2]) * 0.5
(pdiff$estimate[1] - pdiff$estimate[2]) * 0.5  +
  (0.5 * qt(c(0.05/2, 1-0.05/2),df =pdiff$parameter) * pdiff$stderr)
```

```
mean of x
  0.14168
[1] -0.058758  0.342125
```

7.2.1 Analysing preferences

Example 7.10 (page 283)

I couldn't find a direct way to perform the Prescott's test or Gart's test. Instead, we may perform the MH chi square test of linear trend or Fischer's exact test.

```
DescTools::MHChisqTest(matrix(c(6,3,8,7,3,3),nrow = 2,byrow = TRUE))
fisher.test(matrix(c(6,8,7,3), nrow = 2, byrow = TRUE))
```

The syntax "package::function" used as `DescTools::MHChisqTest` in our example, can be used to call a function of a particular library. This is useful when that library has not been loaded or when we want to specify the function of that particular library though loaded is masked by another function with the same name in another library that was loaded later.

```
    Mantel-Haenszel Chi-Square

data:  matrix(c(6, 3, 8, 7, 3, 3), nrow = 2, byrow = TRUE)
X-squared = 1.62, df = 1, p-value = 0.2

    Fisher's Exact Test for Count Data

data:  matrix(c(6, 8, 7, 3), nrow = 2, byrow = TRUE)
p-value = 0.24
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.038833 2.316053
sample estimates:
odds ratio
   0.33757
```

7.3 Allocation to treatment group

7.3.1 Randomisation

We will look at a command that is concerned with randomisation. The function `sample` generates a random permutation of its supplied argument.

```
sample(c("A", "B", "C", "D"))
```

Each time, the above code is run, we will get a different sequence. This may be useful for global randomisation.

If `sample` is provided just a positive number, it gives us a random permutation of the numbers from 1 to the supplied number.

```
sample(10)
```

We can specify a bias by providing the `prob` argument to `sample`.

```
sample(c("A", "B"), prob =c(1,2),  size = 1)
```

When we run this code, we will get either “A” or “B” as we have specified `size=1`. The chance of getting “B” is two times the chance of getting “A”. This could be useful for biased coin method.

The package `Minirand` has functions `blkrandomisation` that generates treatment allocation sequences based on random permuted blocks and `Minirand` that generates treatment allocation sequences based on minimisation algorithms. Another package `blockrand` provides the functions `blockrand` that generates block randomised treatment allocation sequences and `plotblockrand` which generates a pdf file of individual randomisation cards from the treatment allocation sequences.

We are now at the end of this chapter.

7.4 Recap

7.4.1 Concepts introduced in this chapter

- long format

7.4.2 Commands introduced in this chapter

- dplyr::group_by
- tidyr::pivot_wider
- base::levels
- ggplot2::geom_segment
- ggplot2::facet_wrap
- ggplot2::scale_shape_manual
- stats::qt
- base::sample

8

Sample size determination

This chapter deals with power and sample size calculations. We will be using the `epiR` package. All the functions that we use have a similar way of specifying the parameter we want it to calculate; the parameter we want the function to calculate should be specified as `NULL` and all other arguments specified with appropriate values. We start with example 8.1.

8.1 Power

Example 8.1 (page 298)

```
power.t.test(power = NULL,
             n = 50,
             delta = 0.5,
             sd = 1.4,
             type = "one.sample",
             alternative = "one.sided")
```

The function that we use is `power.t.test` of `stats` package. As we want the power to be calculated, our first argument is `power = NULL`. The argument `n` stands for the number of observations, `delta` for the difference in means and `sd` for the standard deviation. The `type` argument specifies whether our test is one sample, two sample or paired. In our case, we are testing one sample against a hypothesised value. The type of hypothesis testing is specified by `alternative`. In our case, we want a one sided test.

```
     One-sample t test power calculation

              n = 50
          delta = 0.5
             sd = 1.4
      sig.level = 0.05
          power = 0.80106
    alternative = one.sided
```

The default print of the function shows the value of all arguments, not just the one calculated. If we want to refer to only one component, we can use the `$` operator to select the component we want. The value we obtain (0.8011) is slightly different from our textbook value of 0.8107.

DOI: 10.1201/9781003589563-8

Example 8.2 (page 302)

Calculating power for the two sided alternative of example 8.2 requires only changing the argument named `alternative`.

```
power.t.test(power = NULL,
             n = 50,
             delta = 0.5,
             sd = 1.4,
             type = "one.sample",
             alternative = "two.sided")$power
```

```
[1] 0.69696
```

The result given by `power.t.test` 0.697 is slightly different from the textbook value of 0.7141 . This probably is because of the normal approximation used in the textbook.

8.2 Testing a mean value

Example 8.3 (page 303)

Calculating the sample size instead of power, as in example 8.3, requires us to change the argument that is given the value `NULL`.

```
power.t.test(n = NULL,
             power = 0.9,
             delta = 0.5,
             sd = 1.4,
             type = "one.sample",
             alternative = "one.sided")$n
```

```
[1] 68.516
```

Again, the value is slightly different from the textbook values.

Example 8.4 (page 304)

Changing the value of `power` and `delta` suffices to replicate example 8.4.

```
power.t.test(n = NULL,
             power = 0.95,
             delta = 0.6,
             sd = 1.4,
             type = "one.sample",
             alternative = "one.sided")$n
```

```
[1] 60.302
```

Example 8.5 (page 305)

Example 8.5 and 8.6 are done in a similar fashion.

```
power.t.test(n = NULL,
             power = 0.99,
             delta = 0.1,
             sd = 0.2,
             type = "one.sample",
             alternative = "one.sided")$n
```

```
[1] 64.465
```

Example 8.6 (page 306)

```
power.t.test(n = NULL,
             power = 0.99,
             delta = 0.1,
             sd = 0.2,
             type = "one.sample",
             alternative = "two.sided")$n
```

```
[1] 75.446
```

Example 8.7 (page 307)

In example 8.7, the minimum detectable difference is calculated. In order to redo the example, we use the same function, but specify the value of `d` as `NULL`.

```
power.t.test(power = 0.9,
             n = 50,
             delta = NULL,
             sd = 1.4,
             type = "one.sample",
             alternative = "one.sided")$d
```

```
[1] 0.58759
```

8.3 Testing a difference between means

Example 8.8 (page 308)

To redo the example 8.8 which deals with two sample situation, we need to change the value of the argument `type`.

```
power.t.test(power = 0.95,
             n = NULL,
             delta = 0.5,
             sd = 1.4,
             type = "two.sample",
             alternative = "one.sided")
```

```
     Two-sample t test power calculation

              n = 170.37
          delta = 0.5
             sd = 1.4
      sig.level = 0.05
          power = 0.95
    alternative = one.sided

NOTE: n is number in *each* group
```

The result returned is very different from the textbook value. If we pay attention, we will see that it is nearly half of the textbook value. In other words, the function returns the number required in each of the two samples.

Example 8.9 (page 309)

We cannot use `power.t.test` directly to calculate sample sizes when the two samples are not equal. As an alternative to calculating $(\mathrm{r}+1)^{2/\mathrm{r}}$ and multiplying it with the result given by `power.t.test`, we can use `epi.sscompc` of `epiR` package. Remember to install the package using `install.packages` as discussed in the first chapter.

```
library(epiR)
epi.sscompc( n = NA,
             treat = 8,
             control = 4,
             sigma = 4,
             power = 0.9,
             r = 4,
             sided.test = 2,
             conf.level = 0.95)
```

The arguments to `epi.sscompc` is slightly different from `power.t.test`. The function expects `NA` as the value of the argument which we want to be calculated. Instead of `d`, it requires `treat` and `control`, the values in the two groups, from which the difference is calculated. It supports `r` which is used to specify the ratio of observations in the treatment group to that in the control group. The type of hypothesis test is specified as a number assigned to the argument `sided.test`. The argument `sd` of `power.t.test` is named `sigma` in `epi.sscompc`.

```
$n.total
[1] 67

$n.treat
[1] 53

$n.control
[1] 14

$power
[1] 0.9

$delta
[1] 4
```

The values returned by `epi.sscompc` are accessible using `$` operator includes `n.total`, `n.treat`, `n.control`, `power` and `delta`.

Example 8.10 (page 310)

To redo the example 8.10, we use the `epi.sscompc`. As, the function doesn't have an argument named delta, we need to specify both `treat` and `control` (from which delta is calculated) as `NA`. Also, we need to remember that `n` is the total number of subjects from both groups combined.

```
epi.sscompc( n = 150,
             treat = NA,
             control = NA,
             sigma = 1.4,
             power = 0.95,
             r = 2,
             sided.test = 1,
             conf.level = 0.95)$delta
```

```
[1] 0.79771
```

Example 8.11 (page 311)

To redo the paired sample example 8.11, we use `power.t.test`. We need to specify `type` as `paired`.

```
power.t.test(power = 0.95,
             n = NULL,
             delta = 0.5,
             sd = 1,
             type = "paired",
             alternative = "one.sided")$n
```

```
[1] 44.68
```

8.4 Testing a proportion

Example 8.12 (page 312)

To redo the example 8.12, we use `epi.sscompb` of `epiR`.

```
epi.sscompb(n = NA,
            treat = 0.3,
            control = 0.32,
            power = 0.9,
            sided.test = 1 )$n.total
```

The result returned by `epi.sscomph` is as in the textbook. For `epi.sscompb`, we specify `n` as `NA` to say that we want it to be calculated. The `treat` and `control` are used to specify the proportions in the treatment and control groups. We specify `sided.test = 1` to say that we want to evaluate a one sided hypothesis.

```
[1] 4568
```

8.5 Testing a relative risk

Example 8.13 (page 314)

We will use `epi.sscohortc` function to redo the example 8.13.

```
epi.sscohortc(n = NA,
              irexp0 = (5 * 413 /100000),
              irexp1 = (1.4 * 5 * 413 /100000),
              power = 0.9,
              sided.test = 1)$n.total
```

The function, similar to the earlier functions, require the parameter that needs to be calculated to be given the value `NA`. So, here we use `n = NA`. The arguments `irexp0` and `irexp1` indicate the incidence risk in the non-exposed and exposed group. We calculate those values from the information given in the textbook. We multiply the average annual death rate for non-smokers given in the textbook with 5 to get the death risk among non-smokers for five year period. We multiply this figure with 1.4, the relative risk of smokers combined with non-smokers as given in the textbook, to obtain the 5 year risk of death among smokers. The arguments `power` and `sided.test` are as in the previous examples.

```
[1] 12130
```

Example 8.14 (page 314)

```
epi.sscohortc(n = NA,
              irexp0 = (5 * 413 /100000),
              irexp1 = (1.4 * 5 * 413 /100000),
              r = 0.5,
              power = 0.9,
              sided.test = 1,
              conf.level = 0.95)$n.total
```

To redo example 8.14, we need to specify the additional argument `r` to `epi.sscohortc`. It is the ratio of the number of exposed to the number of unexposed. As the number of unexposed is higher, value of `r` in our example is less than one.

```
[1] 13544
```

Example 8.15 (page 315)

Example 8.15 is done in a similar way.

```
epi.sscohortc(n = NA,
              irexp0 = 0.2,
              irexp1 = (0.4 * 0.2),
              r = 1,
              power = 0.9,
              sided.test = 2,
              conf.level = 0.95)$n.total
```

```
[1] 348
```

Example 8.16 (page 316)

To calculate the minimum relative risk, as given in example 8.16, we use the same `epi.sscohortc`.

```
epi.sscohortc(n = 12132,
              irexp0 = (5 * 413 /100000),
              irexp1 = NA,
              r = 0.5,
              power = 0.9,
              sided.test = 1,
              conf.level = 0.95)$irr
```

The argument that is specified as `NA` is `irexp1`, the incidence risk in the exposed. The value that we want from the result is of the component `irr`.

```
[1] 0.65824 1.43987
```

The result includes two values, one solved for the positive root and one for the negative root as explained in the textbook. We take the value above unity as we are assuming a risk factor in contrast to a protective factor.

8.6 Case control studies

Example 8.17 (page 318)

For sample size and power calculation related to case control studies, the function `epi.sscc` is used. Here, we rework example 8.17.

```
epi.sscc(n = NA,
         OR = 2.0,
         p0 = 0.30,
         power = 0.90,
         sided.test = 2,
         conf.level = 0.95,
         method = "unmatched")$n.total
```

Similar to the previous commands, `epi.sscc` requires that the parameter that needs to be calculated specified by the value `NA`. Here, we use `n = NA`. The argument `OR` is used to supply the approximate relative risk. The argument `p0` is used to supply the expected population prevalence of the risk factor. The arguments `power`, `sided.test` and `conf.level` are used as in the previous examples. The argument `method` is provided the string `unmatched` to specify that the proposed design is not a matched study. We get the total number of subjects in the `n.total` component of the result.

```
[1] 376
```

Example 8.18 (page 320)

Example 8.18 requires power to be calculated in a study design with a case:control ratio of 1:5. We use `epi.sscc` providing it with the argument `r` to specify the ratio.

```
epi.sscc(power = NA,
         n = 188 + 940,
         OR = 2.0,
         p0 = 0.3,
```

```
         r = 188/940,
         sided.test = 2,
         conf.level = 0.95,
         method = "unmatched")$power
```

Here, we specify `power=NA` to indicate that we want to calculate power. We supply the sum of the number of cases and of controls as given in the textbook as the value of the argument `n`. The argument `r` is the ratio between the cases and controls, calculated from the figures given in the textbook. The power calculated is available in the `power` component of the result.

```
[1] 0.98748
```

Example 8.19 (page 323)

The command `epi.sscc` can handle matched study design scenarios. To redo the example 8.19, we use

```
epi.sscc(n = NA,
         OR = 2.0,
         p0 = 0.30,
         power = 0.90,
         r = 1,
         phi.coef = 0.2,
         sided.test = 2,
         conf.level = 0.95,
         method = "matched")
```

The argument `phi.coef` is used to provide the correlation between case and control exposure for matched pairs. This, I have been told (personal email), is not equivalent to the chance of a discordant pair as discussed in the textbook. Whether it is possible to calculate the correlation between case and control exposure for matched pairs from the chance of a discordant pair given in our text, is not known to me. However, even if we could calculate the exact value of `phi.coef` corresponding to 0.5 chance of a discordant pair, `epi.sscc` would return a value different from that in the textbook as it follows a calculation method different from that described in the textbook.

So, let us make a function that would do the calculations in the textbook.

```
ss.ccmatched <- function(rr, power = 0.90,
                         conf.level = 0.95, r = 1,
                         sided = 2, prop.discord = 0.5) {
  zalpha <- qnorm((1 - conf.level)/sided,
                  lower.tail = FALSE)
  zbeta <- qnorm((1 - power)/2,
                 lower.tail = FALSE)
  n <- 2 * ((zalpha * (rr +1)) +
```

```
            (2 * zbeta * sqrt(rr)))^2 /
    (rr - 1) ^ 2 / prop.discord
  nc <- n *  ((r+1) ^2) / (4 * r)
  ceiling(nc/ (r + 1)) * (r + 1)}
```

We name the function *ss.ccmatched*. The reserved word `function` tells R that we are going to define a function. We expect the function to accept six parameters. All of them except *rr* have default values. Thus when we call the function, if we provide only one unnamed argument, it will be taken as *rr*. Inside the function, we use `qnorm` to calculate *zalpha* considering whether one sided or two sided hypothesis is being tested and *zbeta*. We calculate n as described in the textbook. We use r to correct it for the number of controls for a case. In the final step we use `ceiling` which returns the smallest integer that is not less than its argument. The argument we provide it is the corrected sample size divided by the size of the matched set (number of matched controls for a case plus 1 for case). The value returned by ceiling is multiplied with the same matched set size. Thus we will get the sample size we calculated rounded to the next multiple of the sum of case and the number of matched controls in one set. If there is no explicit `return` statement, the function will return the result of the last statement in its body. Thus our function will return this rounded up number.

Let us try our function

```
ss.ccmatched(2, power = 0.8)
```

```
[1] 362
```

Example 8.19 (page 323)

```
ss.ccmatched(2, power = 0.8, r = 3)
```

```
[1] 484
```

Our function works well. However, it is incomplete. It does not have any error checking. Say, what if I (accidentally) call it with the value 3 for *sided*? In the final chapter, I will point you to some resources to hone your R programming skills.

Before we conclude this chapter, note that, though we do not have a textbook example to demonstrate, there are many more “epi.ss” group of functions that are suited for complex sampling designs including cluster designs.

8.7 Recap

8.7.1 Commands introduced in this chapter

- stats::power.t.test
- epiR::epi.sscompb
- epiR::epi.sscohortc
- epiR::epi.sscc
- base::ceiling

9

Modelling quantitative outcome variables

From this chapter onward we learn about statistical modelling. The main function to fit linear models is `lm`. Its main argument is a formula specifying the model. It can handle data frames. The results are stored and required information extracted from it using helper functions. For this chapter we use the add on package `readr`, `ggplot2` part of tidyverse, `car`, `emmeans`, `gridExtra`, `broom`, `lspline` `ggeffects` and `geepack`. Remember to install them using `install.packages` as discussed in chapter 1. We start with ANOVA by reworking example 9.2.

9.1 One categorical explanatory variable

Example 9.2 (page 335)

```
library(tidyverse)
read_table("K11828 supplements/Datasets/Table 9.1.DAT",
           col_names = c("diet", "chol"),
           col_types = cols(diet = col_factor(),
                            chol = col_double())) -> tbl_9_1
lm(chol ~ diet, data = tbl_9_1) -> lm_9_1
```

First, we import the relevant table using `read_table` supplying it with the file name, the list of column names and the list of column types that is represented in the columns. Note that we specify that *diet* is a factor, to say that the numbers in that column are actually codes that stand for a category and don't have any numerical significance. The analysis of variance is done by `lm` which accepts a formula. Our formula is *chol ~ diet*, to mean that we want the mean of the column *chol* to be tested for equality across the groups dictated by the value in the column *diet*. The `data` argument tells `lm` that the columns mentioned in the formula are to be found in the dataframe *tbl_9_1*. We save the result of the command using the name *lm_9_1*.

```
anova(lm_9_1)
```

To prepare the ANOVA table, we pass the result of `lm` to `anova`.

```
Analysis of Variance Table
```

DOI: 10.1201/9781003589563-9

```
Response: chol
          Df Sum Sq Mean Sq F value  Pr(>F)
diet       2   1.25   0.622    17.6 0.00012 ***
Residuals 15   0.53   0.035
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The values in the ANOVA table printed is similar to table 9.4, except for rounding errors. The term "Error" in table 9.4, is called "Residuals" by `anova`. In addition to the value of F ratio, `anova` computes and prints the probability associated with the F value, as well as the "significance stars" for that probability. Note that there are two alternate functions to perform ANOVA. The function `aov`, prints an abbreviated ANOVA table directly. The function `oneway.test` can be instructed to perform tests with the assumption of equal or unequal variance by changing the value of `var.equal`. Both these functions accept the `formula` and `data` arguments.

R provides many helper functions to display requisite information about fitted models. To display a succinct summary, we can use

```
summary(lm_9_1)
```

```
Call:
lm(formula = chol ~ diet, data = tbl_9_1)

Residuals:
   Min     1Q Median     3Q    Max
-0.253 -0.188  0.045  0.141  0.337

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   6.3150     0.0767   82.30  < 2e-16 ***
diet2        -0.3700     0.1085   -3.41   0.0039 **
diet3        -0.6417     0.1085   -5.91  2.8e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.188 on 15 degrees of freedom
Multiple R-squared:  0.701, Adjusted R-squared:  0.662
F-statistic: 17.6 on 2 and 15 DF,  p-value: 0.000116
```

In the summary displayed, we have the five number summary of residuals, the value, standard error, t test statistic and probability of the t test of the coefficients and summary information on the model fitted. If we want only the coefficients, we can use `coef`. If we want confidence intervals for the coefficients, we may use `confint`.

```
confint(lm_9_1, level = 0.95)
```

```
               2.5 %   97.5 %
(Intercept)  6.15146  6.47854
diet2       -0.60128 -0.13872
diet3       -0.87295 -0.41039
```

Table 9.5 (page 338)

Following the steps outlined in the textbook, we may use this information to build the confidence intervals shown in table 9.5. However, an easier way is to use a model without an explicit intercept term.

```
lm(chol ~ diet - 1, data = tbl_9_1) -> lm_9_5
confint(lm_9_5, level = 0.95)
```

```
       2.5 % 97.5 %
diet1 6.1515 6.4785
diet2 5.7815 6.1085
diet3 5.5098 5.8369
```

The formula argument of our call to `lm` is modified to tell R that we don't want an explicit intercept. This is done by including `-1` on the right-hand side of the formula. We pass this model to `confint`. The result is that the coefficients we get are absolute values rather than the difference from the intercept when we include an explicit intercept. Thus, we need not add together the value of the intercept and the coefficients to calculate the mean cholesterol of each group.

To include the coefficients in the print, we need to use `cbind` to bind together the result of `confint` and `coef` or the `coefficient` component of the result returned by `lm`.

```
cbind( lm_9_5$coefficients, confint(lm_9_5, level = 0.95))
```

TABLE 9.1
Replication of table 9.5

Diet group	Mean	2.5%	97.5%
diet1	6.3150	6.1515	6.4785
diet2	5.9450	5.7815	6.1085
diet3	5.6733	5.5098	5.8369

Example 9.4 (page 338)

To rework the example demonstrating the Bonferroni correction for pair wise comparison, we need to calculate the difference in the coefficients for *diet2* and *diet3*. Our model specification returns as intercept, the coefficient of *diet1* and the other coefficients are calculated as the difference from this intercept. While, we have the option to algebraically manipulate these values to get the difference between the coefficients of *diet2* and *diet3*, we will follow a different path.

```
relevel(tbl_9_1$diet,ref = "2") -> tbl_9_1$diet
lm(chol ~ diet, data = tbl_9_1) -> lmr_9_1
```

The function `relevel` is used to change the reference level of a factor vector. By default, in R, the reference level for any factor vector is determined alphabetically. However, we can modify it using `relevel`, which accepts the factor vector the reference level of which needs to be changed. The new reference level is passed through the argument `ref`. Here, we say that we want the factor value *2* to be used as the reference level. Note that we need to save the changed factor name. Here, we use the same name, overwriting the old one. We then use `lm` to build the model with the changed factor vector.

```
(Intercept)       diet1       diet3
    5.94500     0.37000    -0.27167
```

We can see that the coefficients are calculated with *diet2* as the base. The coefficient for *diet3* directly shows the difference between *diet3* and *diet2* group means. Similarly, the `summary` function will show the standard error, the t value and its probability given the null hypothesis.

We can use this value directly in the remainder of calculations.

```
summary(lmr_9_1)$coefficients
```

```
            Estimate Std. Error t value   Pr(>|t|)
(Intercept)  5.94500   0.076727 77.4825 6.0484e-21
diet1        0.37000   0.108508  3.4099 3.8784e-03
diet3       -0.27167   0.108508 -2.5036 2.4330e-02
```

To obtain the 95% confidence interval for the difference between the group means, we multiply the standard error with the critical value for t distribution obtained using `qt`. The function `qt` requires the probability for which the critical value need to be calculated and the `df`. We specify 1-0.05/2 as our probability as we require 95% confidence interval. The `df` we specify is the error df. We need to subtract and add this spread to the calculated difference. To achieve this, we multiply the calculated spread with the vector `c(-1,1)` and add it to the difference of the group means as returned by our model.

```
abs(lmr_9_1$coefficients[3]) + (c(-1,1) * qt(1-0.05/2, 15) *
                                      summary(lmr_9_1)$coefficients[3,2])
```

```
[1] 0.040386 0.502947
```

To apply the Bonferroni correction, we need to multiply the probability returned by `summary` with the number of comparisons made and judge it against the nominal significance level. Otherwise, we can adjust the nominal significant level by dividing it with the number of comparisons made. Thus, if we are doing all three pair wise comparison, the Bonferroni corrected p value would be `summary(lmr_9_1)$coefficients[3,4] * 3` if we keep the significant level as 5%.

However, R provides `TukeyHSD`, a better alternative. It requires the result returned by `aov`.

```
aov(chol ~ diet, data = tbl_9_1) |> TukeyHSD()
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = chol ~ diet, data = tbl_9_1)

$diet
        diff       lwr       upr   p adj
1-2  0.37000  0.088153  0.651847 0.01019
3-2 -0.27167 -0.553514  0.010181 0.05963
3-1 -0.64167 -0.923514 -0.359819 0.00008
```

The result shows the Tukey honest significant differences calculated for each pair of the factor values (diet in our case).

9.2 One quantitative explanatory variable

Figure 9.2 (page 348)

To rework example 9.5 that demonstrates simple linear regression, we need to import the data.

```
read_table("K11828 supplements/Datasets/Table 9.8.DAT",
            col_names = c("country", "sugar", "dmft"),
            col_types = cols(country = col_factor(),
                                 sugar = col_double(),
                                 dmft = col_double())) -> tbl_9_18
filter(tbl_9_18, country == "2") -> tbl_9_8
```

We use `read_table` to import the relevant data into R. In addition to the columns *sugar* and *dmft*, there is a column of data to distinguish between industrialised countries and developing countries. We import it as a factor calling it *country*. As we need only the data for developing countries, we select only that data using `filter`. We save the filtered dataframe with the name *tbl_9_8*.

The command to generate the graph in figure 9.2 is

```
ggplot(tbl_9_8) +
```

```
  geom_point(aes(x = sugar, y = dmft)) +
  labs(x = "Sugar consumption (kg/person/year)",
       y = "DMFT") +
  theme_bw()
```

We have seen both `ggplot` and `geom_point` earlier. Instead of modifying the default theme, we use `theme_bw`.

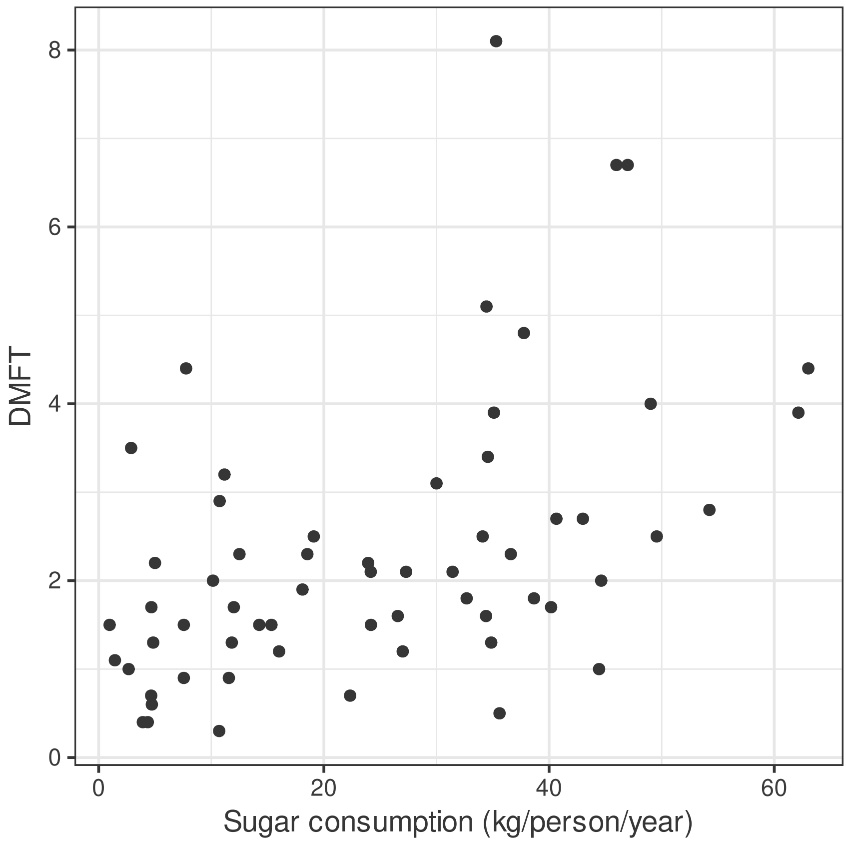


FIGURE 9.1
Replication of figure 9.2

Figure 9.3 (page 350)

To generate the graph in figure 9.3, we need another geom function.

```
ggplot(tbl_9_8, aes(x = sugar, y = dmft)) +
  geom_point() +
  geom_smooth(method = lm, se = FALSE) +
  labs(x = "Sugar consumption (kg/person/year)",
       y = "DMFT") +
  theme_bw()
```

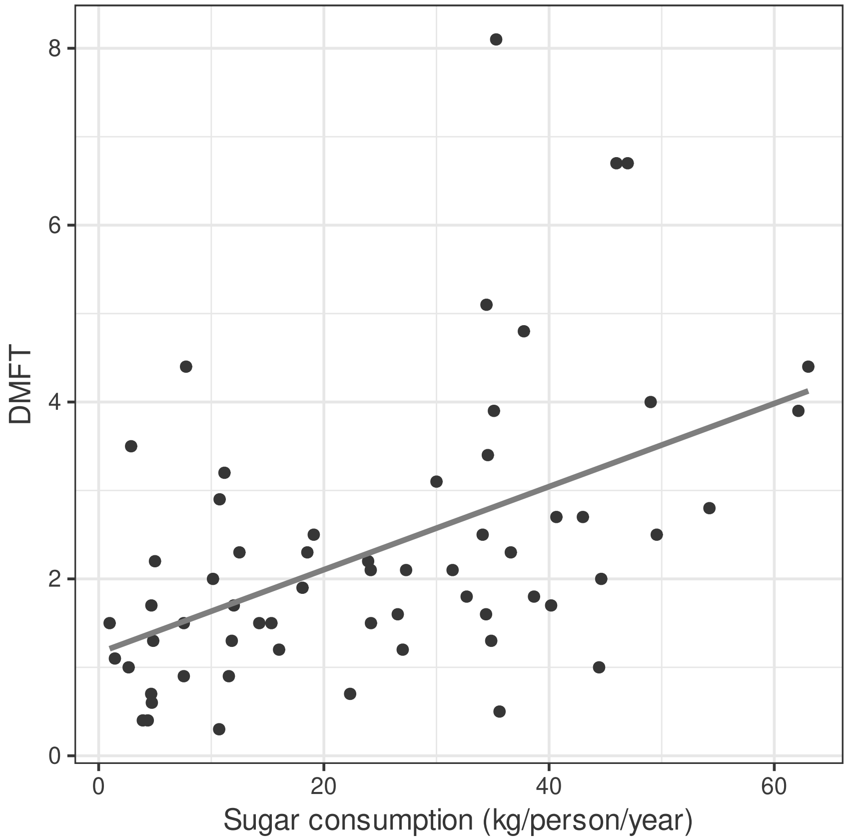


FIGURE 9.2
Replication of figure 9.3

We provide the `aes` definition inside `ggplot` because it is common to both geoms. We have no further arguments to `geom_point`; hence the empty parentheses. The function `geom_smooth` adds a smoothed curve of the values calculated based on its argument. The main argument to `geom_smooth` is `method`, which we specified as `lm`. Thus, it will use the `lm` function to produce the overlaid curve.

Now, we try to redo the actual modelling.

Example 9.5 (page 348)

```
lm( dmft ~ sugar, data = tbl_9_8) -> lm_9_5
```

For simple linear regression, we use `lm`, similar to how we used it for anova. We store the result with a name and use helper function as per our requirement.

We use `anova` to display the analysis of variance table given in table 9.9.

```r
anova(lm_9_5)
```

```
Analysis of Variance Table

Response: dmft
          Df Sum Sq Mean Sq F value  Pr(>F)
sugar      1   36.6    36.6    18.8 5.7e-05 ***
Residuals 59  114.7     1.9
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

To display the value of the coefficients, their standard errors and marginal t tests, we use `summary`.

```r
summary(lm_9_5)
```

```
Call:
lm(formula = dmft ~ sugar, data = tbl_9_8)

Residuals:
   Min     1Q Median     3Q    Max
-2.337 -0.812 -0.290  0.438  5.277

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.1647     0.3204    3.63  0.00059 ***
sugar         0.0470     0.0108    4.34  5.7e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.39 on 59 degrees of freedom
Multiple R-squared:  0.242, Adjusted R-squared:  0.229
F-statistic: 18.8 on 1 and 59 DF,  p-value: 5.66e-05
```

We may use `confint` to obtain confidence intervals for the coefficients.

```r
confint(lm_9_5)
```

```
               2.5 %   97.5 %
(Intercept) 0.523484 1.805877
sugar       0.025318 0.068635
```

We have a helper function to make predictions too.

```r
predict(lm_9_5,
        newdata = data.frame(sugar = 35),
        interval = "predict")
```

```
predict(lm_9_5,
        newdata = data.frame(sugar = 35),
        interval = "confidence")
```

The `predict` function requires a model object which in our case is *lm_9_5*. The `newdata` should be a dataframe containing columns with the same name as used for the explanatory variable in the model. In our example, we build our `newdata` using `data.frame` which is supplied just one value for the variable sugar. The argument `interval` decides what type of interval is needed. For the first of the predictions (of individual values), we use `interval ="predict"`. For the second prediction (of average values), we use `interval = "confidence"`.

```
     fit      lwr    upr
1 2.8088 -0.01254 5.6302
```

```
     fit    lwr    upr
1 2.8088 2.3864 3.2313
```

Figure 9.4 (page 353)

To plot the graph in figure 9.4, we need only ask `geom_smooth` to include the confidence interval.

```
ggplot(tbl_9_8, aes(x = sugar, y = dmft)) +
  geom_point() +
  geom_smooth(method = lm, se = TRUE) +
  labs(x = "Sugar consumption (kg/person/year)",
       y = "DMFT") +
  theme_bw()
```

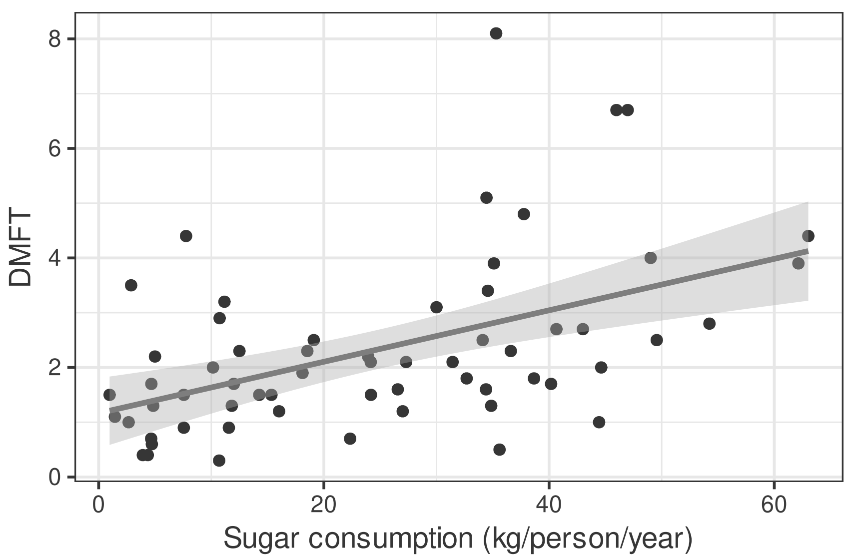


FIGURE 9.3
Replication of figure 9.4

This is achieved by specifying `se =TRUE` in the call for `geom_smooth`. When we ask `geom_smooth` to plot the confidence interval band for the smoothed curve, it uses `predict` to calculate the confidence interval.

9.2.1 Correlation

Section 9.3.2 (page 352)

R provides functions to calculate correlation coefficients.

```
cor(tbl_9_8$sugar, tbl_9_8$dmft, method = "pearson")
cor(tbl_9_8$sugar, tbl_9_8$dmft, method = "spearman")
```

The function `cor` accepts two vectors, the correlation between which needs to be calculated. It also accepts the `method` argument, which we use to specify whether we want Pearson's correlation coefficient or Spearman's.

```
[1] 0.49194
[1] 0.52651
```

The function `cor.test` can be used to test the hypothesis that the correlation coefficient is zero.

```
cor.test(tbl_9_8$sugar, tbl_9_8$dmft, method = "pearson")
```

It also produces a confidence interval for the correlation coefficient calculated. Its arguments are similar to that of `cor`. In addition, it can accept `alternative` to specify the type of hypothesis testing required and `conf.level` to specify the confidence level for the confidence interval calculated.

```

    Pearson's product-moment correlation

data:  tbl_9_8$sugar and tbl_9_8$dmft
t = 4.34, df = 59, p-value = 5.7e-05
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.27407 0.66178
sample estimates:
    cor
0.49194
```

9.2.2 Non linear regression

Figure 9.8 (page 357)

To redo the example 9.6, we need to modify the formula given to `lm`.

```
lm(log(dmft) ~ sugar, data = tbl_9_8) -> lm_9_6
```

Now, the left-hand side of the formula indicating the response variable is `log(`*dmft*`)`, to indicate that we want the log transformation of the *dmft* variable. We can use the model as we used the earlier models, but remembering that it is log of DMFT that is returned by the helper function. To convert the log back to the original scale, we need to use the function `exp`.

Building the graph in figure 9.8 is not directly possible using `geom_smooth`. Instead, we use `geom_line` and `predict`.

```
ggplot(tbl_9_8) +
  geom_point(aes(x = sugar, y = dmft)) +
  geom_line( aes(y = exp(predict(lm_9_6)), x = sugar)) +
  labs(x = "Sugar consumption (kg/person/year)",
       y = "DMFT") +
  theme_bw()
```

The scatter plot is on the original scale. So, we specify `x` and `y` for `geom_point` as such. Over the scatterplots, we draw a line using `geom_line`. The y coordinates for this line is obtained using `predict`. If `newdata` is not given to `predict`, it will use the data used for the model fitting. Thus, our use of `predict` here will return a vector containing predicted y value for each of the x value supplied by the *sugar* column of the dataframe we used for model fitting. Remembering that we used `log` for model fitting, we need to use `exp` to convert back these predicted values to the original scale.

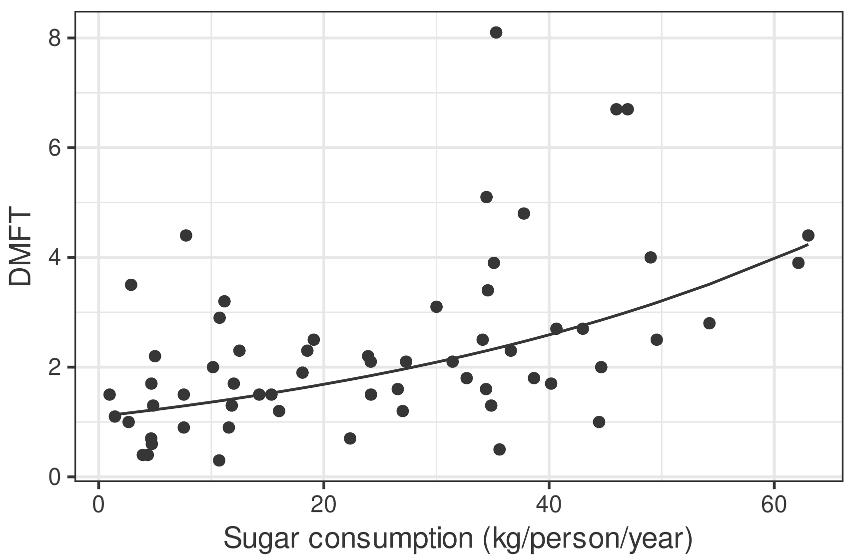


FIGURE 9.4
Replication of figure 9.8

9.3 Two categorical explanatory variables

Example 9.7 (page 360)

To rework the example 9.7 that demonstrates two way anova, we need to get our data into shape.

```
factor(rep("F", 18), levels = c("F", "M")) -> sex
sex[c(1,4,10,13,16,17,3)] <- "M"
cbind.data.frame(tbl_9_1, sex) -> tbl_9_7
```

First, we use `factor` to create a vector containing 18 values, all of them "F". By means of `level` argument, we say that only the values "M" and "F" will be allowed in this vector. We then modify the value of seven elements of this vector, selected by subsetting using an index vector, to "M". We thus get a vector that denotes the sex of each observation as said in example 9.7. We join this vector with the tbl_9_1 we created earlier to make a new table which we name *tbl_9_7*.

We can confirm that the mean given for each cross category is as in table 9.11 using

```
tbl_9_7 |>
  group_by(sex, diet) |>
  summarise(cholesterol = mean(chol))
```

```
# A tibble: 6 x 3
# Groups:   sex [2]
  sex   diet  cholesterol
  <fct> <fct>       <dbl>
1 F     2            5.91
2 F     1            6.09
3 F     3            5.61
4 M     2            6.11
5 M     1            6.36
6 M     3            6.01
```

Again, we use `lm` to perform two way anova.

```
lm(chol ~ diet + sex, data = tbl_9_7) -> lm_9_7
```

The difference from our previous examples is that we have two variables on the right-hand side of the formula.

To get the sequential anova table shown in table 9.12, we need to pass the model object to `anova`.

```
anova(lm_9_7)
```

```
Analysis of Variance Table

Response: chol
          Df Sum Sq Mean Sq F value  Pr(>F)
diet       2  1.245   0.622   27.35 1.5e-05 ***
sex        1  0.211   0.211    9.28  0.0087 **
Residuals 14  0.319   0.023
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We get the same results. However, note that though this is a sequential anova, the labels for the explanatory variables don't make this clear. In other words, we get "sex" as the label instead of "sex|diet".

To obtain type 2 or 3 anova results in base R requires some extra effort. The easier way is to use the function `Anova` from `car` package. Note that the function name starts with uppercase. Install the package using `install.packages` as discussed in chapter 1.

```
library(car)
Anova(lm(chol ~ diet + sex,
         data = tbl_9_7,
         contrasts =list(diet=contr.sum, sex=contr.sum)),
      type=3)
```

```
Anova Table (Type III tests)

Response: chol
            Sum Sq Df  F value Pr(>F)
(Intercept)    597  1 26236.73 <2e-16 ***
diet             0  2     9.87 0.0021 **
sex              0  1     9.28 0.0087 **
Residuals        0 14
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

`Anova` accepts the model object as its first argument. We also need to specify `type`, which may be 2 or 3. The model object we specified here includes a `contrasts` option which is used to specify the contrast used for the factor variables. Of the different options we have, we chose `contr.sum` for both the factors used in the formula. Though, the right contrast is relevant when we have interaction terms in the model, we get into the habit right away. Instead of specifying the contrast in the call to `lm`, we may set our choice of contrast globally by using `options(contrasts = c("contr.sum","contr.poly"))`. The first choice is used for unordered factors and the second is used for ordered factor.

If we pass the model object to `summary` or `coefficients` we won't get the results shown in table 9.13. You should be able to guess the reason for this apparent discrepancy – the factor level that is considered as the reference is different from that in the textbook. We use `relevel` to set the appropriate base level of the factor to get the values in the textbook.

```
tbl_9_7$diet <- relevel(tbl_9_7$diet, ref = "3")
summary(lm(chol ~ diet + sex, data = tbl_9_7))
```

```
Call:
lm(formula = chol ~ diet + sex, data = tbl_9_7)

Residuals:
    Min      1Q  Median      3Q     Max
-0.2534 -0.0842  0.0119  0.1062  0.1951

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.6249     0.0636   88.43  < 2e-16 ***
diet2         0.2717     0.0871    3.12  0.00754 **
diet1         0.4479     0.1079    4.15  0.00098 ***
sexM          0.2907     0.0954    3.05  0.00871 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.151 on 14 degrees of freedom
Multiple R-squared:  0.82,  Adjusted R-squared:  0.782
F-statistic: 21.3 on 3 and 14 DF,  p-value: 1.73e-05
```

9.3.1 Fitted values

Table 9.14 (page 363)

The model object has a component `fitted.values` which may be accessed directly using the `$` operator or by passing the model object to the helper function `fitted`. However, it gives the fitted values for all the observations and hence is inconvenient to work with. We will use the package `emmeans` to calculate the various fitted values.

```
library(emmeans)
emmeans(lm_9_7, c("diet", "sex"))
```

Note that the function we use has the same name as the package – `emmeans`. It requires the model object as its first argument. The second argument `specs` is a character vector that specifies the explanatory variables used in the model specification, from which the cross categories are built.

TABLE 9.2
Replication of table 9.14

Diet group	Sex	Fitted value	S.E.	df	Lower CI	Upper CI
2	F	5.8966	0.063607	14	5.7601	6.0330
1	F	6.0728	0.100572	14	5.8571	6.2885
3	F	5.6249	0.063607	14	5.4885	5.7613
2	M	6.1872	0.100572	14	5.9715	6.4029
1	M	6.3634	0.063607	14	6.2270	6.4999
3	M	5.9156	0.100572	14	5.6998	6.1313

We get a neat grid showing the fitted values of table 9.14. In addition, the confidence interval of the estimates are also shown.

Output 9.5 (page 364)

We get the adjusted means using the same function.

```
emmeans(lm_9_7, specs = c("diet"), weights = "proportional")
```

Here we want only the means for the various levels of *diet*. Hence, the `specs` argument is provided only that name. The `weights` argument is used to specify the weighting scheme that we want to use for adjusting the values. To weight according to the observed frequencies, we specify `weights = "proportional"`. If we want to use the balanced margins weight, we need to specify `weights = "equal"`.

```
emmeans(lm_9_7, specs = c("diet"), weights = "equal")
```

```
 diet emmean     SE df lower.CL upper.CL
 2      6.01 0.0651 14     5.87     6.15
 1      6.19 0.0748 14     6.03     6.35
 3      5.74 0.0651 14     5.60     5.88

Results are averaged over the levels of: sex
Confidence level used: 0.95


 diet emmean     SE df lower.CL upper.CL
 2      6.04 0.0693 14     5.89     6.19
 1      6.22 0.0693 14     6.07     6.37
 3      5.77 0.0693 14     5.62     5.92

Results are averaged over the levels of: sex
Confidence level used: 0.95
```

To get the result of pair wise comparisons of the adjusted means, we have a couple of options. First, we use the function `pwpm`, again from `emmeans`.

```
pwpm( emmeans(lm_9_7,
              specs = c("diet"),
              weights = "proportional"))
```

All we do is pass the result returned by `emmeans` to `pwpm`. As we did not store the result of `emmeans`, we pass the command call directly.

```
       2      1      3
2 [6.01] 0.2646 0.0194
1 -0.176 [6.19] 0.0026
3  0.272  0.448 [5.74]

Row and column labels: diet
Upper triangle: P values   adjust = "tukey"
Diagonal: [Estimates] (emmean)
Lower triangle: Comparisons (estimate)   earlier vs. later
```

The result returned by `pwpm` is a grid similar to output 9.5. However, the p value of the pair wise comparison is given only in the upper triangle, not repeated as in the output from SAS. The diagonal cells are not empty, but contains the adjusted mean calculated. The lower triangle shows the difference between the adjusted means, the p value for which is shown in the upper triangle.

Another option is use to use `pairs` from `emmeans`.

```
pairs(emmeans(lm_9_7,
              specs = c("diet"),
              weights = "proportional"))
```

```
 contrast      estimate     SE df t.ratio p.value
 diet2 - diet1   -0.176 0.1080 14  -1.634  0.2646
 diet2 - diet3    0.272 0.0871 14   3.119  0.0194
 diet1 - diet3    0.448 0.1080 14   4.153  0.0026

Results are averaged over the levels of: sex
P value adjustment: tukey method for comparing a family of 3 estimates
```

The result of `pairs` is similar to that of `pwpd`, but in a rectangular grid omitting the adjusted means themselves. Note that the p values of multiple comparison are different from that in the textbook, probably because `emmeans` uses Tukey method for adjustment.

9.3.2 Interaction

Example 9.9 (page 365)

In order to specify an interaction term, we need to modify the call to `lm`.

```
lm(chol ~ diet * sex, data = tbl_9_7) -> lm_9_9
summary(lm_9_9)
```

We changed the formula to `chol ~ diet * sex`. The `*` instructs that the model should include the interaction terms between the explanatory variables in addition to the main effects. We can pass the model object *lm_9_9* to `summary` and `anova` to confirm that results are similar to that in output 9.6.

```
Call:
lm(formula = chol ~ diet * sex, data = tbl_9_7)

Residuals:
   Min     1Q Median     3Q    Max
-0.250 -0.079  0.004  0.116  0.214

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.6060     0.0708   79.23   <2e-16 ***
diet2         0.3060     0.1001    3.06   0.0099 **
diet1         0.4840     0.1733    2.79   0.0163 *
sexM          0.4040     0.1733    2.33   0.0380 *
diet2:sexM   -0.2060     0.2451   -0.84   0.4171
diet1:sexM   -0.1340     0.2451   -0.55   0.5946
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.158 on 12 degrees of freedom
Multiple R-squared:  0.831, Adjusted R-squared:  0.76
F-statistic: 11.8 on 5 and 12 DF,  p-value: 0.000273
```

9.4 Model building

Example 9.10 (page 366)

We need to import the data to rework the example 9.10 and build the model objects.

```
read_table("K11828 supplements/Datasets/Table 9.15.DAT",
           col_names = c("bmi", "sex", "smoke"),
           col_types = cols(bmi = col_double(),
                            sex = col_factor(),
                            smoke = col_factor())) -> tbl_9_15
lm(bmi ~ sex, data = tbl_9_15) -> lm_9_15_1
lm(bmi ~ smoke, data = tbl_9_15) -> lm_9_15_2
lm(bmi ~ sex + smoke, data = tbl_9_15) -> lm_9_15_3
lm(bmi ~ sex * smoke, data = tbl_9_15) -> lm_9_15_4
```

We use `read_table` to import the data, specifying *sex* and *smoke* as factors. We build 4 model objects using `lm` and save them. The last digit in the model names correspond to that used in example 9.10. As with the previous examples, we can use helper functions to print anova tables, coefficients etc. We may use `Anova` if we want type 2 or 3 anova tables, `emmeans` to get adjusted means and `pairs` or `pwpd` to obtain pairwise comparison of adjusted means as with our previous examples.

9.5 General linear models

Figure 9.9 (page 375)

To rework example 9.11 we import the data, as we did earlier. This time, we don't filter the data as we want data of both developing and industrialised countries.

```
read_table("K11828 supplements/Datasets/Table 9.8.DAT",
           col_names = c("country", "sugar", "dmft"),
           col_types = cols(country = col_factor(),
                            sugar = col_double(),
                            dmft = col_double())) -> tbl_9_18
lm(log(dmft) ~ sugar, data = tbl_9_18) -> lm_9_11_1
lm(log(dmft) ~ country, data = tbl_9_18) -> lm_9_11_2
lm(log(dmft) ~ sugar + country, data = tbl_9_18) -> lm_9_11_3
lm(log(dmft) ~ sugar * country, data = tbl_9_18) -> lm_9_11_4
```

We will not repeat what we saw in previous examples. We will try to reproduce the graphs in figure 9.9. Towards that end, we will extend *tbl_9_18* with the fitted values returned by the models.

```
tbl_9_18$mdl_1 <- fitted(lm_9_11_1)
tbl_9_18$mdl_2 <- fitted(lm_9_11_2)
tbl_9_18$mdl_3 <- fitted(lm_9_11_3)
tbl_9_18$mdl_4 <- fitted(lm_9_11_4)
```

We have used `$` operator to assign the fitted values of each of the model objects to non_existent columns named *mdl_1* to *mdl_4* of *tbl_9_18*. As the columns to which we assign new values are non existent, the columns will be created. Thus, *tbl_9.9* gets extended by four new columns carrying the fitted values according to each model. We now create the four plots.

```
ggplot(tbl_9_18) +
  geom_line(aes( x = sugar,
                 y = mdl_1,
                 group = country,
                 linetype = country)) +
```

```
  scale_y_continuous(breaks = seq(0,1.5, by = 0.5)) +
  ylim(0,1.5)+
  labs(x = "Sugar consumption (kg/person/year)",
       y = "Fitted values",
       title = "Common regression line") +
  scale_linetype_manual(labels = c("Industrialised", "Developing"),
                        values = c(2,1),
                        name = NA) +
  theme_minimal() -> plt_9_9_1
ggplot(tbl_9_18) +
  geom_line(aes( x = sugar,
                 y = mdl_2,
                 group = country,
                 linetype = country)) +
  scale_linetype_manual(labels = c("Industrialised", "Developing"),
                        values = c(2,1),
                        name = NA) +
  scale_y_continuous(breaks = seq(0,1.5, by = 0.5)) +
  ylim(0,1.5)+
  labs(x = "Sugar consumption (kg/person/year)",
       y = "Fitted values",
       title = "No regression line") +
  theme_minimal() -> plt_9_9_2
ggplot(tbl_9_18) +
  geom_line(aes( x = sugar,
                 y = mdl_3,
                 group = country,
                 linetype = country)) +
  scale_linetype_manual(labels = c("Industrialised", "Developing"),
                        values = c(2,1),
                        name = NA) +
  scale_y_continuous(breaks = seq(0,1.5, by = 0.5)) +
  ylim(0,1.5)+labs(x = "Sugar consumption (kg/person/year)",
       y = "Fitted values",
       title = "Parallel regression lines") +
  theme_minimal() -> plt_9_9_3
ggplot(tbl_9_18) +
  geom_line(aes( x = sugar,
                 y = mdl_4,
                 group = country,
                 linetype = country)) +
  scale_linetype_manual(labels = c("Industrialised", "Developing"),
                        values = c(2,1),
                        name = NA) +
  scale_y_continuous(breaks = seq(0,1.5, by = 0.5)) +
  ylim(0,1.5)+
  labs(x = "Sugar consumption (kg/person/year)",
       y = "Fitted values",
```

```
      title = "Separate regression lines") +
  theme_minimal() -> plt_9_9_4
```

Each of the plots are saved with a name. All plots have the same x axis *sugar*, y axis being one of the columns with fitted values. The geom used is `geom_line`, which is also provided the argument `group` and `linetype`. This will result in different lines being drawn for different values in *country*. The `scale_y_continuous` and `ylim` are used so that all the four plots have the same y axis limit and ticks instead of the default decided by `ggplot`.

The package `ggplot` doesn't have a function to stitch together multiple graphs. So, we use the package `gridExtra`.

```
library(gridExtra)
grid.arrange(plt_9_9_1, plt_9_9_2, plt_9_9_3, plt_9_9_4,
             nrow= 2 )
```

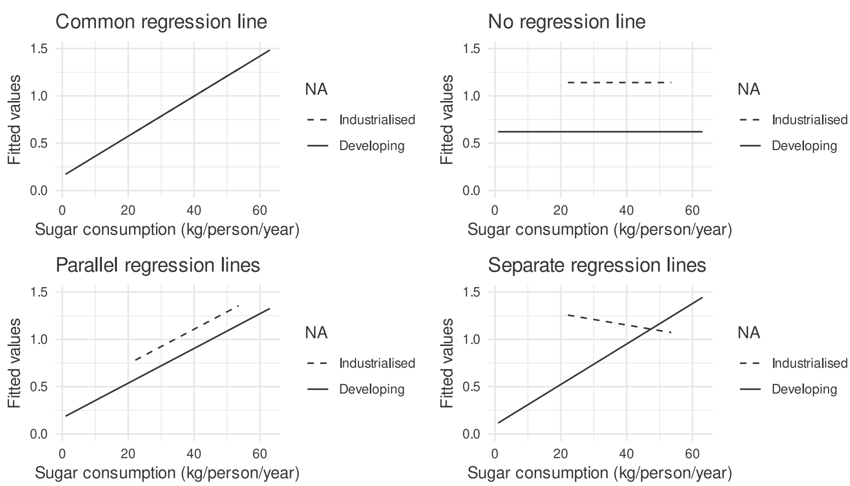


FIGURE 9.5
Replication of figure 9.9

The function `grid.arrange` accepts as many plots as we provide it, arrange them as we specify and return them as a single graph. In our example, we specify `nrow = 2`. So, we get the four graphs arranged in two rows.

Figure 9.10 (page 377)

The code to prepare the graph in figure 9.10 is

```
ggplot(tbl_9_18,
      aes( x = sugar, group = country)) +
  geom_point(aes(y = dmft, shape = country), show.legend = FALSE) +
  geom_line(aes(y = exp(mdl_4), linetype = country), show.legend = FALSE) +
```

```
  labs(x = "Sugar consumption (kg/person/year)",
       y = "DMFT")
```

All the concepts used in building this graph were discussed earlier.

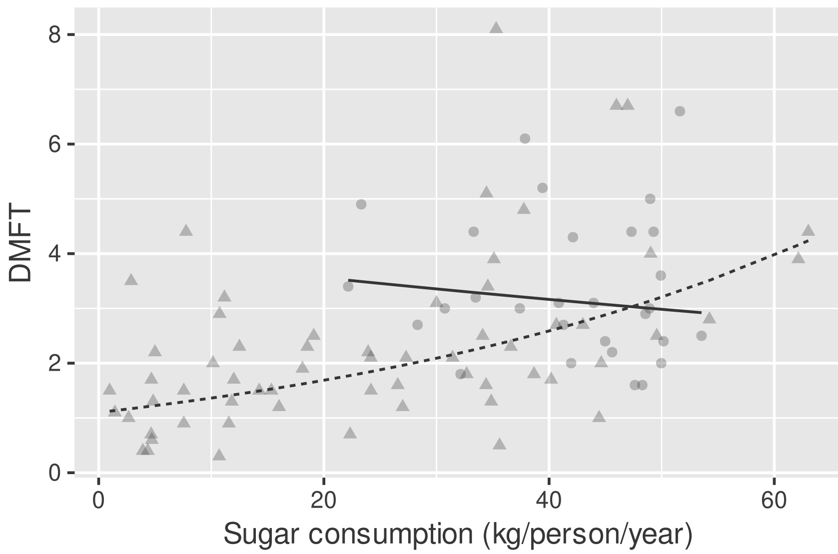


FIGURE 9.6
Replication of figure 9.10

9.6 Several explanatory variables

Table 9.19 (page 379)

To rework the example 9.12, we need to import the data.

```
read_table("K11828 supplements/Datasets/Example 9.12.DAT",
           col_names = c("hdl", "age", "alcohol","chol", "fibre"),
           col_types = cols(hdl = col_double(),
                            age = col_double(),
                            alcohol = col_double(),
                            chol = col_double(),
                            fibre = col_double())) -> tbl_9_12
```

To prepare table 9.19 of the textbook, we use the function `cor`.

```
cor(tbl_9_12)
```

The function `cor` computes the correlation between the columns of its argument if the first argument is a dataframe. So, `cor` will calculate the correlation between the columns of *tbl_9_12*.

TABLE 9.3
Replication of table 9.19

	HDL	Age	Alcohol	Cholesterol	Fibre
hdl	1	−0.006052	0.3281	0.05753	−0.0414
age	−0.006052	1	−0.1244	−0.03287	−0.01706
alcohol	0.328074	−0.124365	1	0.14576	−0.13285
chol	0.057526	−0.032869	0.1458	1	0.1023
fibre	−0.041398	−0.017055	−0.1328	0.1023	1

Table 9.20 (page 380)

There are no functions to directly prepare table 9.20 of the textbook. We will prepare it in a few steps. We will use the library `broom`. Remember to install it using `install.packages` as discussed in chapter 1.

```
library(broom)
mdl_summary <- function(mdl) {
  bind_cols(
    data.frame(var_names = paste(attr(mdl$terms, "term.labels"),
                              collapse = ", "),
               var_num  = length(attr(mdl$terms, "term.labels")),
               r.square = summary(mdl)$r.squared * 100,
               ems = anova(mdl)["Residuals", "Mean Sq"]),
    pivot_wider(select(tidy(mdl), term,estimate),
                names_from = term,
                values_from = estimate))}
```

Here we define a new function, which we call *mdl_summary*. The **keyword** `function` tells R that what follows is a function. The parenthesis following the word `function` determines the arguments that the function can accept and their names inside the function. Here, it will accept only one argument under the name *mdl*. We want the function to accept a fitted model and generate a row containing the details as in table 9.20. The code inside the body of the function, i.e., the portion between the braces are intended to achieve this.

We do this by binding together two data frames column wise. The function that does this is `bind_cols`. It is provided with two single row data frames to bind together.

The first one is built from scratch using `data.frame`, inside which the names of the columns and their values are specified. We specify three columns.

The first, we name as *var_names*. We calculate its value as `paste(attr(mdl$terms, "term.labels"), collapse = ", ")`. We extract the `term.labels` **attribute** of the `terms` component of *mdl* and then collapses its values into a comma-separated string using `paste`. Attributes are meta data about an R object. For example, data frames have the attribute `names` which stores the name of the columns. We can find out what attributes an R object has and their values using the function `attr`. If we want just the value of a specific attribute, we pass the name of that attribute too. Here, we want the attribute of the component `terms` of the fitted model. The component `terms` stores the details of the terms used in model fitting. Its `term.labels` attribute carries the names of the terms as a character vector. Thus we get a string that contains the name of the variables used for specifying the model.

The second column *var_num* makes use of the same attribute `term.labels`, but ascertain its length using `length`. Thus its length will give us the number of variables used for specifying the model.

The third column *r.square* stores the `r.squared` component of `summary`'s return value, after multiplying with 100. The function `anova` returns a dataframe. We subset it to obtain the row with the name `Residuals` and the column with the name `Mean Sq`. Thus we get the error mean square for the model, which we save with the name *ems*.

The second data frame we ask `bind_cols` to join together is provided by `pivot_wider`. It accepts a dataframe, which in our case is provided by `tidy` from the package `broom`. The function presents the coefficients from fitted model as a neat data frame with their standard error and p value. By means of `select`, we select only the columns `term` and `estimate` from the function's return value. Thus, we provide a two column data frame to `pivot_wider`. The function `pivot_wider` reshapes the data frame to a wide format. It takes the values from the column specified as `names_from` argument and makes them new column headings. The values for each of these new columns is filled from the original column specified as the argument `values_from`. Thus, we get as many terms are there in the fitted model with their estimates as the value of the single row of the data frame.

The result returned by our custom function will be a single row data frame containing all the info present in one row of table 9.20 for the model we pass to it.

However, the table is filled with data from all possible models described in example 9.12.

```
lm(hdl ~ age, data = tbl_9_12) -> lm_9_12_age
lm(hdl ~ alcohol, data = tbl_9_12) -> lm_9_12_alc
lm(hdl ~ chol, data = tbl_9_12) -> lm_9_12_chol
lm(hdl ~ fibre, data = tbl_9_12) -> lm_9_12_fib
lm(hdl ~ age + alcohol, data = tbl_9_12) -> lm_9_12_age_alc
lm(hdl ~ age + chol, data = tbl_9_12) -> lm_9_12_age_chol
lm(hdl ~ age + fibre, data = tbl_9_12) -> lm_9_12_age_fib
lm(hdl ~ alcohol + chol, data = tbl_9_12) -> lm_9_12_alc_chol
lm(hdl ~ alcohol + fibre, data = tbl_9_12) -> lm_9_12_alc_fib
lm(hdl ~ chol + fibre, data = tbl_9_12) -> lm_9_12_chol_fib
lm(hdl ~ age + alcohol + chol, data = tbl_9_12) -> lm_9_12_age_alc_chol
lm(hdl ~ age + alcohol + fibre,data = tbl_9_12) -> lm_9_12_age_alc_fib
lm(hdl ~ age + chol + fibre,data = tbl_9_12) -> lm_9_12_age_chol_fib
lm(hdl ~ alcohol + chol + fibre,data = tbl_9_12) -> lm_9_12_alc_chol_fib
lm(hdl ~ ., data = tbl_9_12) -> lm_9_12_all

list(lm_9_12_age, lm_9_12_alc, lm_9_12_chol, lm_9_12_fib,lm_9_12_age_alc,
     lm_9_12_age_chol, lm_9_12_age_fib,lm_9_12_alc_chol, lm_9_12_alc_fib,
     lm_9_12_chol_fib,lm_9_12_age_alc_chol, lm_9_12_age_alc_fib,
     lm_9_12_age_chol_fib, lm_9_12_alc_chol_fib,lm_9_12_all) -> list_mdls
```

We specify all the models and save them in a suitable named object. The formula used in all the models except the last should be clear. The formula for the last model `hdl ~ .`, means that the model terms should include all variables except *hdl*, the outcome variable on the left-hand side of the formula.

We collect all the models in a group using `list`. A **list** is a collection of R objects, similar or not, usually dissimilar.

```
map_dfr(list_mdls, mdl_summary)
```

The function `map_dfr` belongs to a family of map functions from `tidyverse`. These functions accept an argument – a list or a vector and calls the second argument, a function, repeatedly with the values in the first argument. Thus, `map_dfr(list_mdls, mdl_summary)` will call `mdl_summary` with each of the model objects in the *list_mdls*. The result returned by each call is row joined by `map_dfr` for the final result. Thus we will get our table 9.20.

9.6.1 Information criteria

Example 9.13 (page 382)

The function used to calculate AIC, is `AIC`.

```
AIC(lm_9_12_age_alc)
```

```
[1] 3524.6
```

The function `AIC` accepts one or more models for which it calculates the AIC. The function `BIC` calculates BIC. The value returned by both functions is slightly different from the textbook values. However, the models with the minimum AIC or BIC are the same as in the textbook. Preparing table 9.22 can be done in a way similar to how we built table 9.20. We need to modify our custom function to add columns for AIC and BIC and remove those that are not required. We will however, make use of another function `glance` from `broom`.

Table 9.22 (page 382)

```
mdl_compare <- function(mdl) {
  bind_cols(
    data.frame(var_names = paste(attr(mdl$terms, "term.labels"),
                                 collapse = ", "),
               var_num  = length(attr(mdl$terms, "term.labels"))),
    glance(mdl))}

map_dfr(c(list(lm(hdl ~ 1, data =tbl_9_12)), list_mdls),
        mdl_compare) |>
  mutate( age = ifelse(str_detect(var_names, "age"),"X","-"),
          alcohol = ifelse(str_detect(var_names, "alcohol"),"X","-"),
          cholesterol = ifelse(str_detect(var_names, "chol"),"X","-"),
```

TABLE 9.4
Replication of table 9.20

Variable names	Number of x variables	R^2 (%)	Error mean square	Intercept	Estimates			
					Age	Alcohol	Cholesterol	Fibre
age	1	0.00366	0.135	1.38	−0.000386			
alcohol	1	10.76325	0.12	1.26		0.00622		
chol	1	0.33092	0.134	1.31			0.0001369	
fibre	1	0.17138	0.135	1.41				−0.002059
age, alcohol	2	10.88589	0.12	1.15	0.002248	0.0063		
age, chol	2	0.33266	0.134	1.32	−0.000265		0.0001365	
age, fibre	2	0.17595	0.135	1.43	−0.000431			−0.0020647
alcohol, chol	2	10.77287	0.12	1.25		0.00619	0.0000236	
alcohol, fibre	2	10.76373	0.12	1.26		0.00623		0.0001107
chol, fibre	2	0.55686	0.134	1.36			0.0001485	−0.0023765
age, alcohol, chol	3	10.89657	0.12	1.14	0.002258	0.00628	0.0000249	
age, alcohol, fibre	3	10.88705	0.12	1.14	0.002256	0.00631		0.0001709
age, chol, fibre	3	0.55918	0.134	1.37	−0.000307		0.0001481	−0.0023798
alcohol, chol, fibre	3	10.77297	0.12	1.25		0.0062	0.0000233	0.0000504
age, alcohol, chol, fibre	4	10.89703	0.12	1.14	0.002262	0.00628	0.0000242	0.0001083

```
        fibre = ifelse(str_detect(var_names, "fibre"),"X","-")) |>
select(age, alcohol, cholesterol, fibre,var_num, logLik, AIC, BIC)
```

The function `glance` which accepts a model object returns a single row data frame with multiple statistics about that model. The information returned includes AIC, BIC, r squared, log likelihood etc. Our new custom function *mdl_compare* joins these columns with *var_names* and *var_num* we saw in our previous custom function. We pass this custom function along with the list of models to `map_dfr` which will join together row wise the data frames returned for each model. The list we pass is made by adding the null model to the existing list. The null model formula is `hdl ~ 1`, the right-hand side with just `1`. Note how the joint list is made – we provide the lists to be joined as arguments to `c`. Though the model object is a list, we need to place the null model object inside `list` before passing on to `c` so that it is appended properly.

We pass this combined dataframe as the first argument of `mutate` by piping it using `|>`. We create four new columns using `mutate` to mark with an "X", the use of that particular variable as a modelling term in that model. We use `str_detect` to check if the name of that variable is present in the value of *var_names*. If `str_detect` returns `TRUE`, `ifelse` assigns the value "X" to that column, otherwise it assigns "-". The function `ifelse` accepts a construction that returns a logical value, either `TRUE` or `FALSE`. Instead of functions like `str_detect`, we may use expressions constructed using **comparison operators** `>`, `<`, `==`, `<=`, `>=`, `!=` according to our needs. Finally, the result of `mutate` is piped to `select` to select only the columns that we want printed.

TABLE 9.5
Replication of table 9.22

Terms included in the model							
age	alcohol	cholesterol	fibre	Number of x terms	$log_e\hat{L}$	AIC	BIC
-	-	-	-	0	−2040.5	4085.0	4098.0
X	-	-	-	1	−2040.4	4086.8	4106.3
-	X	-	-	1	−1761.7	3529.4	3548.8
-	-	X	-	1	−2032.4	4070.8	4090.3
-	-	-	X	1	−2036.3	4078.6	4098.1
X	X	-	-	2	−1758.3	3524.6	3550.6
X	-	X	-	2	−2032.3	4072.7	4098.7
X	-	-	X	2	−2036.2	4080.4	4106.4
-	X	X	-	2	−1761.4	3530.8	3556.8
-	X	-	X	2	−1761.7	3531.3	3557.3
-	-	X	X	2	−2026.8	4061.7	4087.6
X	X	X	-	3	−1758.0	3526.0	3558.5
X	X	-	X	3	−1758.3	3526.6	3559.0
X	-	X	X	3	−2026.8	4063.5	4096.0
-	X	X	X	3	−1761.4	3532.8	3565.3
X	X	X	X	4	−1758.0	3528.0	3567.0

While we have reworked the examples in the textbook, in the usual work flow we will probably be using a few more functions. The function `update` will update and refit a model.

It requires a model object and an update formula. For example, an alternate way to build some of the models we built earlier is

```
lm(hdl ~ 1, data = tbl_9_12) -> lm_9_12_null
update(lm_9_12_null, . ~ . + age) -> lm_9_12_age
update(lm_9_12_age, . ~ . - age + alcohol) -> lm_9_12_alc
```

We are asking `update` to drop or add one or more terms to the model we supply to it. The update formula is what makes clear our requirement. In the update formula, a period stands for the terms that were used in the original model, a minus `-` for dropping a term and a plus `+` for adding a new term. Thus, in the *lm_9_12_alc*, we are asking `update` to regress whatever term was on the left-hand side of the formula of *lm_9_12_age* against all terms originally on the right-hand side, but dropping *age* and adding *alcohol*.

Another function that may be useful is `step`, which does step wise selection. It accepts an initial model, a `scope` argument and a `direction` argument. To select the appropriate model, we may use, for example

```
step(lm(hdl ~ 1, data = tbl_9_12),
     hdl ~ age + alcohol +chol +fibre,
     direction = "both")
step(lm(hdl ~ ., data = tbl_9_12),
     direction = "both")
```

In the first example, we start with the null model, provide a `scope`, which is taken as the upper model. In the second example, the model object we provide regresses *hdl* on all variables. As we haven't provided a `scope` argument, the starting model object itself is treated as the upper model. Instead of `direction = "both"`, we may specify `"backward"` or `"forward"` if that is what we want. We also have the option to turn off the print that happens during the selection process using `trace = 0` or provide a positive number to print the details.

9.7 Model checking

Figure 9.11, 9.12 (page 385)

To check model fit graphically, we need to pass the model object to `plot`.

```
plot(lm_9_5, which =1)
plot(lm_9_5, which =2)
```

The command `plot` will print the diagnostic plots if it is provided a model object as its argument. By default, when we plot diagnostic plots in base R, 4 plots are generated. The argument `which` is used to restrict the plot to our choice from among the six that R provides.

It accepts a vector containing any combination of values from 1 to 6. Here we select the residual plot and normal plots individually.

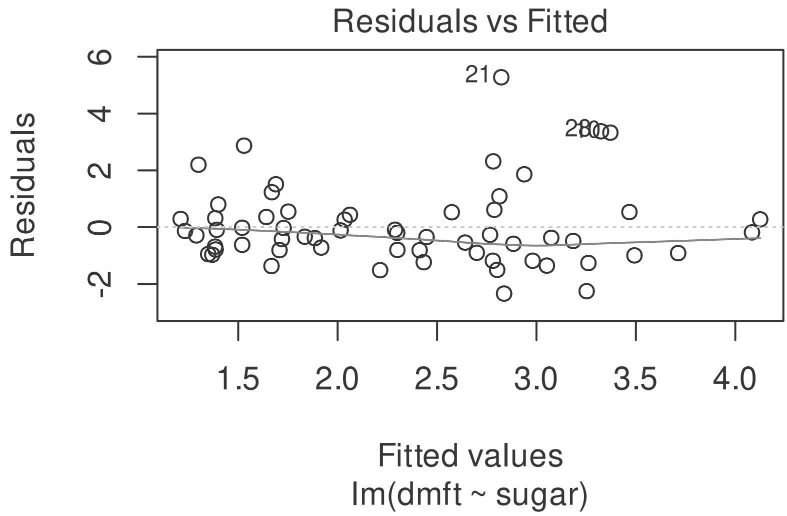


FIGURE 9.7
Replication of figure 9.11

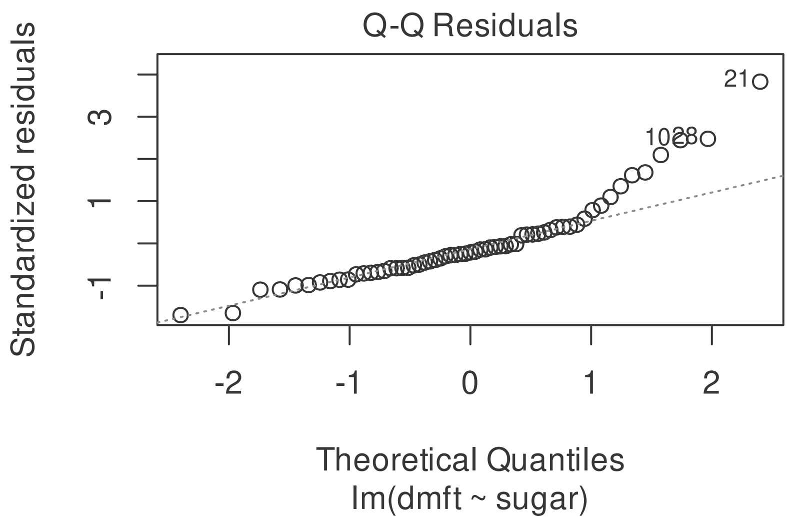


FIGURE 9.8
Replication of figure 9.12

The diagnostic plots that are printed include scale location plot and residuals versus leverage plot in addition to residual plot and normal plot discussed in the textbook.

Figures 9.13, 9.14 (page 386)

Next, we print the residual plot and normal plot for the log model.

```
plot(lm_9_6, which = 1)
plot(lm_9_6, which = 2)
```

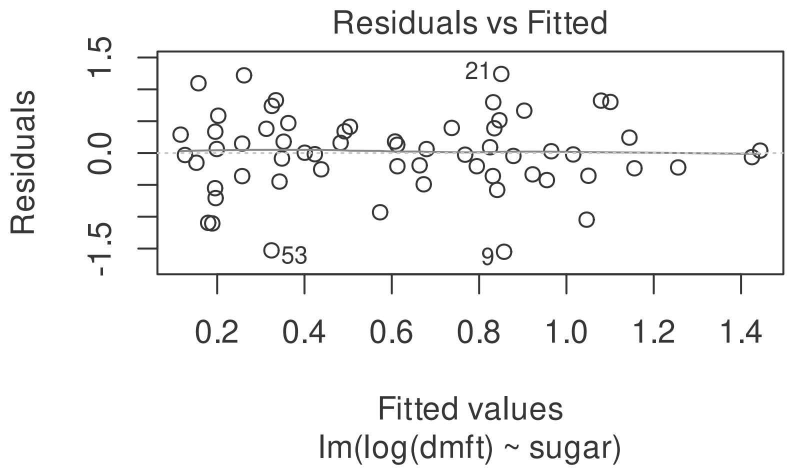


FIGURE 9.9
Replication of figure 9.13

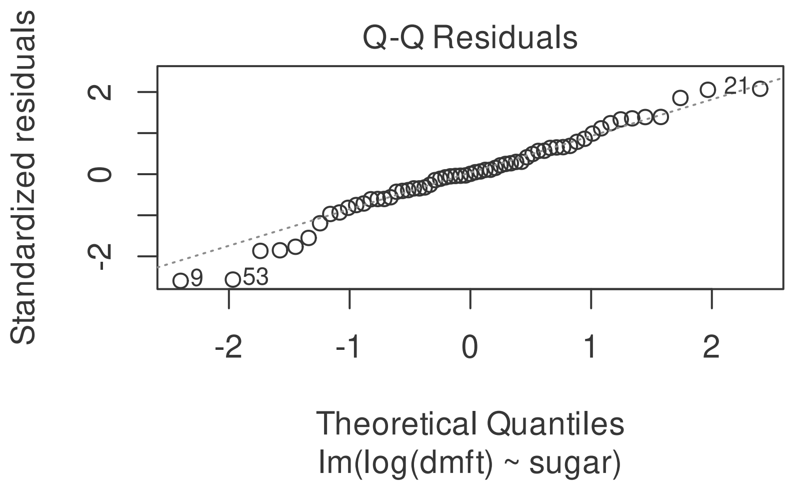


FIGURE 9.10
Replication of figure 9.14

Example 9.14 (page 384)

We can access the residuals directly from a model object using the helper function `residuals` or using `$`. We may, pass the residuals directly to `plot`. We may use `qqnorm` to plot a normal plot of the residuals, passing to it the residuals. The function `cooks.distance` calculates

Cook's distance for each observation used for model building. The `plot` function will generate a graph showing Cook's distance if `which` has the value `4`. The function `influence.measures` will generate some more measures of influence in addition to Cook's distance. The function `rstandard` produces standardised residuals. All these functions accepts a model object as its parameter. We may subset the values returned by these functions if we are interested in some particular value. For example, for Algeria given in example 9.14,

```
fitted(lm_9_5)[1]
residuals(lm_9_5)[1]
rstandard(lm_9_5)[1]
```

will give the fitted value, residual and standardised residual.

```
    1
2.884
       1
-0.58401
       1
-0.42427
```

9.8 Confounding

Example 9.15 (page 387)

To rework the example 9.15, we need to first prepare the data.

```
c(126,131,118,128,128,130,123,137,113,131,123,122,
  125,127,117,138,135,136,148,123,130,125,131,124,
  133,123,135,119,140,128,132,126,128,130,130,135) -> sbp
factor(c(rep(c("chilli", "none"), each = 2),
         rep("chilli",4), rep("none",8), rep("chilli",4),
         rep("none",16)),
       levels = c( "none", "chilli")) -> sauce
factor(rep(c("mex", "white", "afam"),
           times = c(4,12,20))) -> ethnic

bind_cols(sbp = sbp,
          sauce = sauce,
          ethnic = ethnic) -> tbl_9_15
```

We have prepared three vectors and joined them into a dataframe using `bind_cols`. The vectors with factor data was prepared using `factor` to which character vectors made using multiple calls to `rep` to reflect the data given in table 9.24 was fed.

Table 9.23 presents data of the example, categorised according to the value of one column. It also contains a summary statistics – mean, for the two groups of data. Here, we use `aggregate` to show how to obtain summary statistics for subsets of data.

```
aggregate(sbp ~ sauce, data = tbl_9_15, FUN = mean) |>
  pivot_wider(names_from = "sauce",
              values_from = "sbp") -> tbl_9_23
```

The function `aggregate` accepts a formula and a `data` argument from which to obtain the variables in the formula. The variable on the left-hand side of the formula is used for calculating the summary statistic, categorised according to the variables on the right-hand side of the formula. The function name which is used for calculating the summary statistic is passed on to the argument `FUN`. In our example, we categorise *sbp* according to the different values of *sauce*, both from the *tbl_9_24* dataframe, and calculate the mean of *sbp* using the function `mean` for each of the subcategories of *sauce*. We pipe the result, a data frame, using `|>` to `pivot_wider` and the final result is stored as *tbl_9_23*. We will see why we need `pivot_wider` in a moment.

TABLE 9.6
Replication of table 9.23

No chilli sauce	Chilli sauce
127.35	131.7

Table 9.24 (page 388)

Table 9.24 is rather complicated with many summary measures.

```
tbl_9_15 |>
  group_by(sauce, ethnic) |>
  summarise( mean = mean(sbp),
             count = n(),
             values = paste(sbp, collapse = ",")) |>
  pivot_wider(names_from = c("sauce"),
              values_from = c("mean", "count", "values"),
              names_sep = "\n",
              id_cols = "ethnic") |>
  mutate(perc_chili = count_chilli * 100 /(count_chilli + count_none),
         diff = mean_chilli - mean_none,
         mean = (mean_chilli * count_chilli + mean_none * count_none)/
           (count_none + count_chilli)) |>
  select(ethnic, values_chilli, values_none, perc_chili, mean,
         mean_chilli, mean_none, diff) -> tbl_9_24
```

First, we pipe *tbl_9_15* to `group_by`, which produces a grouped dataframe. We are in effect saying that we want separate measures for the cross categories decided by the different values of *sauce* and *ethnic*.

We pipe the grouped database to `summary`, which will produce new columns according to our instructions. We ask `summarise` to make a column *mean*, which will contain the value returned by the function `mean` to which *sbp* is passed as the argument. The second column we ask `summarise` to make is *count*, which will carry the result given by `n`, which is the number of rows. The third column *values* will contain the values in the column *sbp* joined together using ",". As we have passed a grouped dataframe to `summarise`, the result, a data frame, will contain as many cross categories as are there in the grouped data frame. The values of the variables used for categorisation will also be included in separate columns named appropriately.

We pipe the result of `summarise` to `pivot_wider` which changes the dataframe to a wide column format. It takes the values in the column specified as `names_from` and creates new columns, one set for each value in that column. What values, this column will contain is decided by the `values_from` column. We specify three columns as the value of `values_from`. Thus three sets of columns, the names of which will be derived from the values in *sauce* combined with the column names we specified from `values_from`. We get a three row seven column data frame from this `pivot_wider` command.

We pass the result of `pivot_wider` to `mutate` to make new columns. We make three new columns, *perc_chilli* for the percentage of chilli users, *diff* for the difference in means between the chilli users and non users and *mean* for the mean bp without regards to chilli use.

In the next step, we use `select` to select only those columns we want to display. Finally, we may use `t` to transpose the dataframe so that columns become rows and rows columns. However, we won't do it now, as we need the data frame in this orientation for the graph of figure 9.15.

Figure 9.15 (page 388)

We will supply the tables we prepared in the previous steps to prepare the graph in figure 9.15.

```
ggplot() +
  geom_point(aes(x = sauce,
                 y = sbp,
                 shape = ethnic),
             data = tbl_9_15) +
  geom_segment(aes(x = factor("none"),
                   y = none,
                   xend = factor("chilli"),
                   yend = chilli),
               data = tbl_9_23,
               colour = "blue") +
  geom_segment(aes(x = factor("none"),
                   y = mean_none,
                   xend = factor("chilli"),
                   yend = mean_chilli,
                   linetype = ethnic,
                   colour = ethnic),
               data = tbl_9_24) +
  labs(x = "Chilli sauce use?",
```

```
    y = "Systolic blood pressure (mmHg)") +
scale_colour_manual(values = c( "#111111", "#004B73","#713430" ))
```

There are three geoms, each using a different data frame. So, we don't give `ggplot` any arguments. The first `geom_point` uses the raw table. In addition to specifying the `x` and `y` values, we pass *ethnic* as the value for `shape` arguments. Thus, we will get different shapes for the points plotted based on the ethnicity of the user. The second `geom_segment` uses the *tbl_9_23* to display using a line segment, the unadjusted means for the two categories of sauce users. The arguments, `y` and `yend` needs to be in different columns of the data frame used. We used `pivot_wider` while preparing the table for this reason. As there is only one line segment to be drawn, we specify `x` and `xend` directly. We use `factor` so that the values will have the same interpretation as in the previous geom. The last `geom_segment` uses the *tbl_9_24* to draw three line segments, one for each category of *ethnic* as we pass *ethnic* as the value for `colour` and `line_type` arguments. The `y` and `yend` are derived from different columns of the data frame, the reason for not transposing the data frame; while `x` and `xend` are specified `directly`.

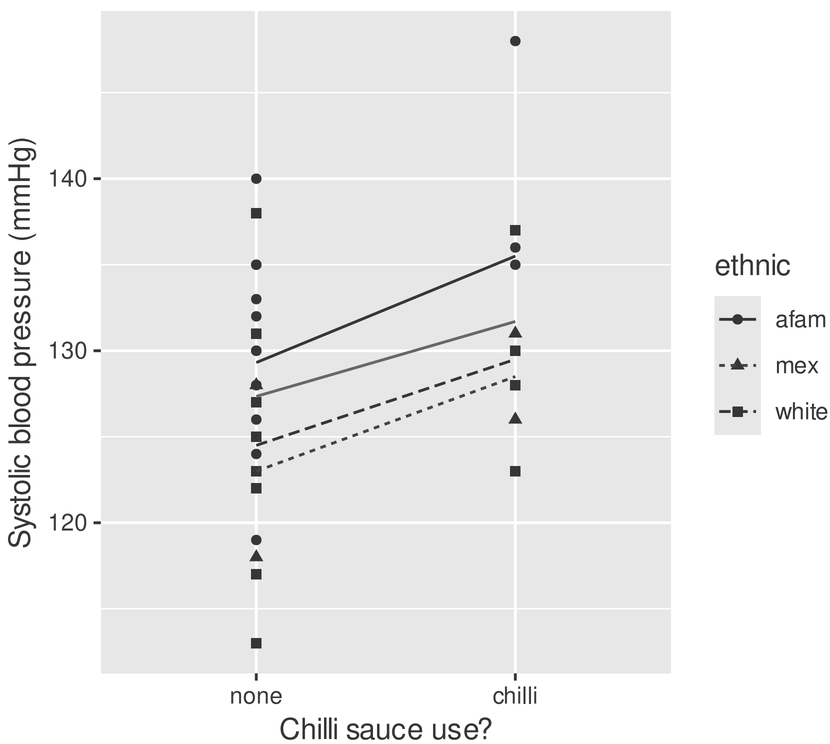


FIGURE 9.11
Replication of figure 9.15

We may now transpose and pretty the *tbl_9_24* to make it look similar to that in the textbook.

```
library(sjmisc)
rotate_df(tbl_9_24, cn = TRUE, rn=" ")
```

We may use `rotate_df` of `sjmisc` to rotate the table. The `cn = TRUE` argument asks it to use the first row of the original dataframe as the column heading of the rotated dataframe. The `rn=" "` asks the function to include the row headings as the first column, with the heading of that column being an empty string.

TABLE 9.7
Replication of table 9.24

	African Americans	Mexican Americans	Whites
values_chilli	135,136,148,123	126,131	128,130,123,137
value_none	130,125,131,124, 133,123,135,119, 140,128,132,126, 128,130,130,135	118,128	113,131,123,122, 125,127,117,138
perc_chili	20.000	50.000	33.333
mean	130.55	125.75	126.17
mean_chilli	135.5	128.5	129.5
mean_none	129.31	123.00	124.50
diff	6.1875	5.5000	5.0000

Note: s.e = standard error

Table 9.25 (page 389)

To rework the example 9.16, we import the data first.

```
read_table("K11828 supplements/Datasets/Example 9.16.DAT",
           col_names = c("fibrinogen", "age", "hpylori"),
           col_types = cols(fibrinogen =  col_double(),
                            age = col_double(),
                            hpylori = col_factor())) -> tbl_9_16

levels(tbl_9_16$hpylori) <- c("Negative", "Positive")
```

After importing the data, we change the labels attached with the factor levels using `levels`, which accepts the factor vector. Rather than receiving its result, we assign the character vector containing the new labels to it using `<-`. We do this so that we get decent labels while printing.

```
tbl_9_16 |>
  group_by(hpylori) |>
  summarise( n = n(),
             mean = mean(fibrinogen),
```

```
              `(s.e.)` = sd(fibrinogen)/sqrt(n),
              Q1 = quantile(fibrinogen, 0.25),
              Q2 = quantile(fibrinogen, 0.5),
              Q3 = quantile(fibrinogen, 0.75))
```

Preparing table 9.25 uses `group_by` and `summarise` we learned earlier. One thing to note is how we specified the name of the *(s.e.)* column. The name contains parentheses, which is not allowed in names unless the name is put inside a pair of back ticks. The names, properly called **identifiers** should only contain letters, digits, underscore and period. Names that adhere to this rule are called **syntactic** names. We should strive to use only syntactic names. However, if you must use non-syntactic names, you must **quote** them within back ticks.

TABLE 9.8
Replication of table 9.25

H. pylori status	n	Mean	s.e	Q_1	Q_2	Q_3
Negative	149	2.7604	0.058563	2.33	2.55	3.07
Positive	361	2.9289	0.037925	2.43	2.84	3.35

Table 9.26 can be easily constructed using the above code, substituting *age* for *fibrinogen*. Table 9.27 requires some extra steps.

Table 9.27 (page 390)

```
mutate(tbl_9_16,
       age_grp = cut(age,
                     breaks = c(25,35,45,55,65,75),
                     right = FALSE)) -> tbl_9_16
```

At first, we add a new column named *age_group* to *tbl_9_16* using `mutate`. For this, we use the function `cut`, which produces a factor vector from a numerical variable, based on which interval the values fall with respect to the `breaks` specified. The option `right` determines which side of the intervals is closed.

Next, we prepare two new data frames from *tbl_9_16*.

```
tbl_9_16 |>
  group_by(age_grp, hpylori) |>
  summarise(n = n(),
            mean = mean(log(fibrinogen)),
            s.e = sd(log(fibrinogen)) / sqrt(n)) |>
  pivot_wider(names_from = "hpylori",
              values_from = c("n", "mean","s.e")) -> tbl_9_27_1
```

Here, we group the data by *age_grp* we created in the previous step and by *hpylori* and then calculate the summary measures in table 9.27. We then change the data frame into a wider format as we want different columns for different values of *hpylori*.

We haven't calculated p values for the t tests comparing the log of fibrinogen levels between the two levels of *hpylori*. The reason is that `t.test` is done on a group of rows, not for each row of the dataframe.

```
tbl_9_16 |>
  group_by(age_grp) |>
  nest() |>
  mutate(p.value = map_dbl(data,
                           function(df) {t.test(log(fibrinogen) ~ hpylori,
                                                data = df,
                                                var.equal = TRUE)$p.value}),
         n = map_dbl(data, nrow)) -> tbl_9_27_2
```

We group by only *age_grp* and then pipe the result to `nest`. The function `nest` creates **nested data frames**. For each value of *age_grp*, `nest` will collect all the rows of the supplied dataframe and create a new dataframe, which will be contained in a cell of the data frame returned as result. By default, the column that contains the nested data frames is called `data`. In our case, the data frame returned by `nest` will have two columns – one containing the unique values from the column *age_grp* and another with the nested data frames.

We pipe the result of `nest` to `mutate` to create two columns. The column *p.value* is calculated by `map_dbl` to which the nested data frames are passed as the first argument. The second argument to `map_dbl` is an anonymous function, which calls `t.test` with arguments appropriate to test the difference between the two levels of *hpylori* in each of the nested data frames. The `p.value` component of the `t.test` result is returned by `map_dbl`. Similarly, the column *n* is constructed by `map_dbl`, this time calling `nrow` with the nested dataframe to obtain the number of rows in each of the nested data frames.

We now have two data frames which we need to join together to get table 9.27 of the textbook.

```
left_join(tbl_9_27_1, tbl_9_27_2) |>
  select(c(age_grp, n, mean_Positive, s.e_Positive,
           mean_Negative, s.e_Negative, p.value))
```

The function `left_join` joins two data frames keeping all rows of the first dataframe and values in the columns of the second data frame for all rows with the same value in the matching columns. We have not specified a matching column because the column *age_grp* is named the same in both data frames and so, will be used as the default matching column. The result from `left_join` is piped to `select` to choose the columns we want to print.

TABLE 9.9
Replication of table 9.27

		H. pylori status				
Age group (years)	n	Positive	s.ep	Negative	s.en	p value
[25,35)	65	0.85633	0.040663	0.86621	0.032932	0.857642
[35,45)	100	0.98980	0.031695	0.96507	0.029907	0.598363
[45,55)	107	1.00137	0.023543	1.00084	0.038512	0.990083
[55,65)	122	1.09739	0.022300	1.10496	0.051917	0.885753
[65,75)	116	1.13323	0.024809	1.03190	0.058855	0.086528

Example 9.16 (page 389)

Fitting the linear model and obtaining the least square means for the data in example 9.16 is not different from the previous examples we saw.

```
emmeans(lm(log(fibrinogen) ~ age + hpylori,data = tbl_9_16),
        specs = "hpylori")
```

```
 hpylori  emmean     SE  df lower.CL upper.CL
 Negative   1.01 0.0189 507    0.976     1.05
 Positive   1.03 0.0121 507    1.011     1.06

Results are given on the log (not the response) scale.
Confidence level used: 0.95
```

9.9 Splines

Example 9.17 (392)

To fit linear splines we can use `mutate` to re-express *sugar* the way explained in the textbook and fit them. Instead, we follow a slightly different, but equivalent path in our first example.

```
lm(dmft ~ sugar + I((sugar-12) * (sugar <=12)) + I((sugar-34) * (sugar <=34)),
   data = tbl_9_8) -> ls_9.17
summary(ls_9.17)
```

While we use the `lm` command for fitting linear splines, the difference is in how we specify the right-hand side of the formula argument. On the right-hand side, we use three terms. The first is the unchanged *sugar* variable. The second and third term uses `I`. The function `I` is used to insulate the expression given inside it from being interpreted according to the special meanings attached to mathematical operators in the context of formulas. Thus, expressions given inside `I` are evaluated in the usual mathematical sense and the resulting value is used for model fitting. The mathematical expression given inside `I` consists of two parts multiplied together. The first part deducts the value of one of the knots from the observed *sugar* value. The second part returns `TRUE` if the observed *sugar* value is less than the value of knot and `FALSE` otherwise, which gets **coerced** to 1 and 0 respectively when used in numerical operations. Thus, we will get a zero if the observed *sugar* value is below the knot considered and difference from the knot value if it is greater.

```
Call:
lm(formula = dmft ~ sugar + I((sugar - 12) * (sugar <= 12)) +
    I((sugar - 34) * (sugar <= 34)), data = tbl_9_8)

Residuals:
   Min     1Q Median     3Q    Max
-2.318 -0.868 -0.296  0.479  5.298

Coefficients:
                                Estimate Std. Error t value Pr(>|t|)
(Intercept)                      0.95373    1.36036    0.70     0.49
sugar                            0.05237    0.03249    1.61     0.11
I((sugar - 12) * (sugar <= 12)) -0.03048    0.09085   -0.34     0.74
I((sugar - 34) * (sugar <= 34)) -0.00182    0.05119   -0.04     0.97

Residual standard error: 1.42 on 57 degrees of freedom
Multiple R-squared:  0.244, Adjusted R-squared:  0.205
F-statistic: 6.15 on 3 and 57 DF,  p-value: 0.00108
```

As we are using `lm`, we may use any of the helper functions like `summary` to extract required information from the model object as we did in our previous examples. While the coefficients are different from the text value, the model we fitted results in the same fitted values as in the text.

Output 9.17 (page 393)

For the second example, we will use a new library, `lspline`. Remember to install it using `install.packages` as discussed in chapter 1.

```
library(lspline)
lm(dmft ~ lspline(sugar, knots = c(12,34), marginal = TRUE),
   data = tbl_9_8)
```

The right-hand side of formula is supplied by `lspline`, which accepts the variable that has to be split into pieces and the value of knots where split should occur. Instead of `knots`, we may specify `n` which specifies the number of equally spaced intervals or `q` which specifies the number of equal frequency intervals. To obtain coefficients for changes in slope rather than the actual slopes, we need to change the default value of `marginal` argument of `lspline`.

```
Call:
lm(formula = dmft ~ lspline(sugar, knots = c(12, 34), marginal = TRUE),
    data = tbl_9_8)

Coefficients:
                                        (Intercept)
                                            1.38143
lspline(sugar, knots = c(12, 34), marginal = TRUE)1
```

```
                                                    0.02007
lspline(sugar, knots = c(12, 34), marginal = TRUE)2
                                                    0.03048
lspline(sugar, knots = c(12, 34), marginal = TRUE)3
                                                    0.00182
```

Figure 9.16 (page 394)

We will now plot the graph in figure 9.16.

```
ggplot(tbl_9_8, aes(x = sugar, y = dmft)) +
  geom_point() +
  geom_smooth(method = "lm",
              formula = y ~ x + I((x-12) * (x<=12)) + I((x-34) * (x<=34))) +
  labs(x = "Sugar consumption (kg/person/year)",
       y = "DMFT")
```

We use `ggplot`, with two geoms. The first `geom_point` is used to plot the individual data values as points. The second `geom_smooth` plots smoothed conditional means calculated from the fitted model objects. In addition to `x` and `y` aesthetics, `geom_smooth` requires a `method` and `formula`. The `method` we use is `lm`, the function we want it to produce the model object. The `formula` is similar to what we used earlier. The difference is that we use `x` and `y` instead of the variable's name.

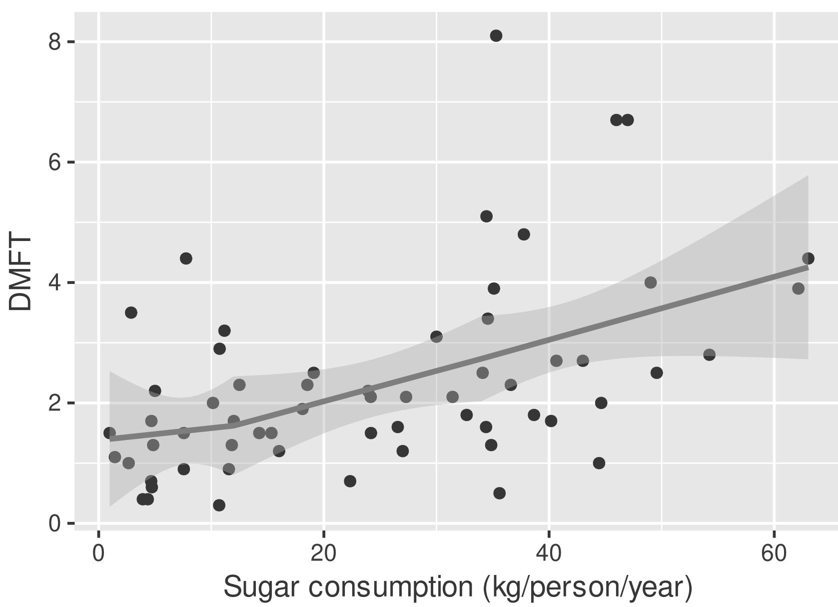


FIGURE 9.12
Replication of figure 9.16

The graph generated is similar to that in figure 9.16 though we have specified the linear splines differently from the textbook.

9.9.1 Other types of splines

Figure 9.17 (page 396)

First, we will reproduce figure 9.17. We use `ggeffects` package here. Remember to install it using `install.packages` as discussed in chapter 1.

```
lm(hdl ~ age + lspline(alcohol, knots = c(2,10,24)) ,
   data = tbl_9_12) -> ls_9_12
library(ggeffects)
plot(ggpredict(ls_9_12, terms = c( "alcohol"))) +
     labs( x = "Alcohol consumption (units/week)",
           y = "HDL-cholesterol (mmol/l)")
```

We fit a piece wise linear spline regression of *hdl* on *alcohol* and *age*. We use `lspline` to split *alcohol* according to the `knots` provided. The `ggpredict` function of `ggeffects` package returns predicted values adjusted for the `terms` specified. Here, we get predicted values of *hdl* adjusted for *age* for the alcohol use values of 0, 2, 10, 24 and 155. These values are the extremes of the *alcohol* values and the `knots` we supplied. The result returned by `ggpredict` is graphed by `plot`. The object returned by `plot` when it is passed the result of `ggpredict` is a ggplott2 object. So, we may add layers as we do to a ggplot object. Here, we add `labs` for axis labels.

Figure 9.18 (page 396)

The library `splines` is used for fitting splines. For natural splines we use `ns` from this package.

```
library(splines)
lm(hdl ~ age + ns(alcohol, knots = c(10),Boundary.knots = c(2,24)),
   data = tbl_9_12) -> ns_9_12
plot(ggpredict(ns_9_12, terms = c( "alcohol"))) +
       labs(x = "Alcohol consumption (units/week)",
            y = "HDL-cholesterol (mmol/l)")
```

The difference from our previous example is that instead of `lspline`, we use `ns`. The arguments that `ns` require are similar to `lspline` – the variables name and the `knots`. The additional argument required is `Boundary.knots`, which along with `knots` determine which segments of the regression are linear. In our example, up to the value 2 and beyond the value 24, it will be linear; 2 to 10 and 10 to 24 will be smooth.

The `splines` package also provides `bs` for B splines. The `loess` function from `stats` performs local polynomial regression fitting.

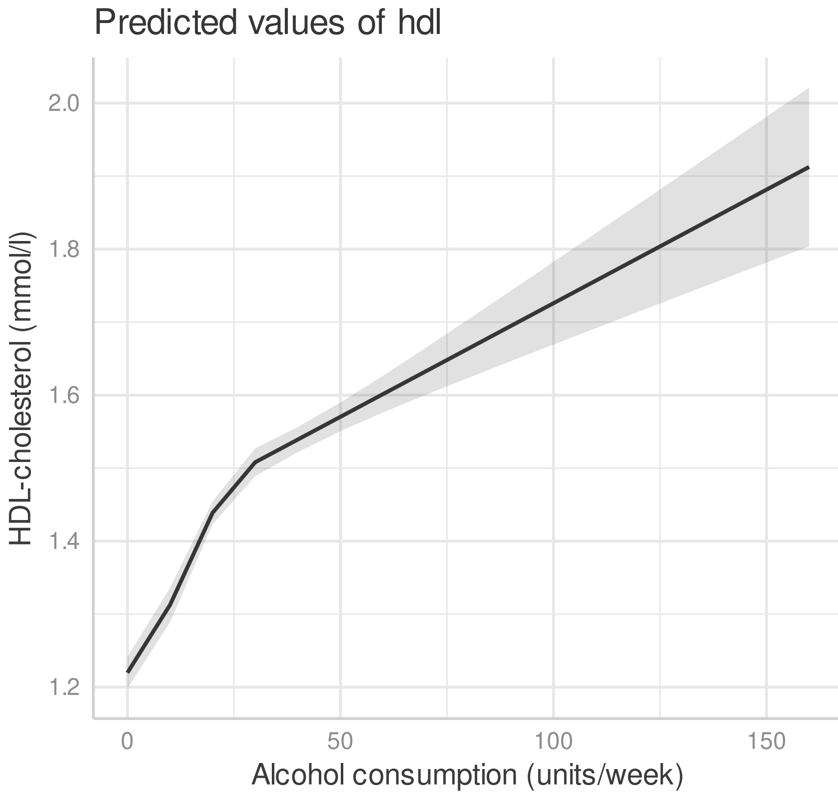


FIGURE 9.13
Replication of figure 9.17

9.10 Panel data

Example 9.19 (page 399)

There are different packages to perform generalised estimating equations. We will be using `geepack`. Remember to install it using `install.packages` as discussed in chapter 1. First, we need to prepare the data.

```
c(3.795,6.225,5.210,7.040,7.550,7.715,6.555,5.360,5.285,6.230,
  6.475,5.680,5.490,9.865,4.625,7.480,4.970,6.710,4.765,6.695,
  4.025,5.510,5.495,5.435,5.350,5.905,6.895,4.350,5.950,5.855,
  5.410,5.220,4.700,4.215,5.395,7.475,4.925,7.115,7.020,5.365,
  3.665,6.130,4.895,7.000) -> first

c(3.250,6.935,4.750,5.080,8.685,7.775,6.005,4.940,5.620,5.870,
  6.620,5.635,5.080,9.465,4.120,6.955,5.100,7.480,4.530,6.160,
  4.160,6.010,5.010,5.975,4.705,5.465,6.925,4.260,5.325,5.505,
  5.280,5.175,4.815,3.610,5.705,6.580,5.190,6.150,6.395,5.805,
  3.710,5.160,5.145,7.425) -> second
```

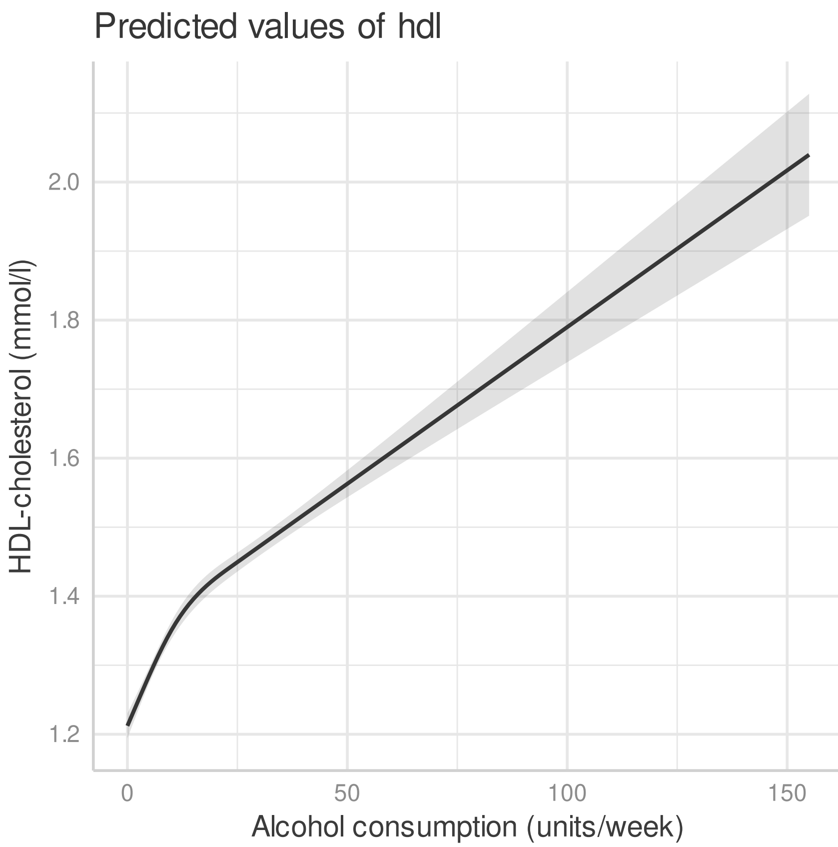


FIGURE 9.14
Replication of figure 9.18

```
bind_cols(id = c(1:44,1:44),
          time = c(rep(1,44), rep(2,44)),
          chol = c(first, second)) -> tbl_2_15
```

We use `bind_cols` to prepare the data as mentioned in example 9.19.

```
library(geepack)
geeglm(chol ~ time,
       id = id,
       data = arrange(tbl_2_15, id, time),
       corstr = "exchangeable") -> gee_9_19
summary(gee_9_19)
```

The arguments required by `geeglm` are similar to `lm`. The first argument required is a formula. The second argument required is `id`, which should carry the cluster identifier. The data frame should be sorted by this variable so that the observations on a cluster are contiguous. We have sorted the table according to *id* variable and on *time* within each *id* value using arrange which is supplied to the data argument, the data frame from which to obtain the variables of formula and id. The corstr is a string that determines the correlation structure that will be used in the calculations. The result of geeglm supports the summary and coefficient helper methods. Also, there is an anova method for geeglm objects. The function QIC calculates QIC, QICu and other measures when a geeglm object is supplied.

```
Call:
geeglm(formula = chol ~ time, data = arrange(tbl_2_15, id, time),
    id = id, corstr = "exchangeable")

 Coefficients:
            Estimate Std.err   Wald Pr(>|W|)
(Intercept)   6.0114  0.2092 825.96   <2e-16 ***
time         -0.1700  0.0833   4.17    0.041 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Correlation structure = exchangeable
Estimated Scale Parameters:

            Estimate Std.err
(Intercept)     1.52   0.382
  Link = identity

Estimated Correlation Parameters:
      Estimate Std.err
alpha      0.9   0.036
Number of clusters:   44  Maximum cluster size: 2
```

The coefficients returned by geeglm are similar to that in output 9.19. The associated p values are also same.

Example 9.20 (page 400)

We rework the example 9.20 similarly.

```
read_table("./K11828 supplements/Datasets/Example 9.20.DAT",
            col_names=c("id", "period", "group","med", "score"),
            col_types = cols(id = col_number(),
                             period = col_factor(),
                             group = col_factor(),
                             med = col_factor(),
                             score = col_number())) -> data_7_5

geeglm(score ~  period * med, id = id,
       data = arrange(data_7_5, id, period),
       corstr = "exchangeable") -> gee_7_5_pm

geeglm(score ~  med, id = id,
       data = arrange(data_7_5, id, period),
       corstr = "exchangeable") -> gee_7_5_m
```

```
coef(summary(gee_7_5_pm))
coef(summary(gee_7_5_m))
```

```
             Estimate Std.err     Wald Pr(>|W|)
(Intercept)   2.90327   0.167 3.00e+02    0.000
period2       0.00952   0.261 1.33e-03    0.971
med2          0.01102   0.258 1.82e-03    0.966
period2:med2 -0.30240   0.502 3.63e-01    0.547
            Estimate Std.err  Wald Pr(>|W|)
(Intercept)    2.908  0.1296 503.6    0.000
med2          -0.145  0.0978   2.2    0.138
```

Though the calculated coefficients are different from the textbook examples, p value of the relevant term is essentially the same as in the textbook.

Example 9.21 (page 400)

We rework the example 9.21 in a similar fashion. We import the data first.

```
read_table("K11828 supplements/Datasets/Example 9.21.dat",
           col_names = c("id","week","group","accept","flavour","change"),
           col_types = cols(id = col_factor(),
                            week = col_double(),
                            group = col_factor(),
                            accept =  col_double(),
                            flavour = col_double(),
                            change = col_logical()),
           na = ".") -> tbl_9_21
```

We have used `read_table` to import the data as in our previous examples. We have used one argument `na` that we haven't used earlier. When the data contain specific string to mark values that are not available, that string should be passed on to `na`. Here, we are telling R that a period is used to mark values that are not available.

```
relevel(tbl_9_21$group, ref = "R") -> tbl_9_21$group
geeglm(accept ~ week * group,
       id = id,
       data = arrange(tbl_9_21, id, week),
       corstr = "exchangeable") -> gee_9_21
summary(gee_9_21)
```

We change the reference level of the factor variable *group* from the default and build the model object as we did in our previous examples.

```
Call:
```

```
geeglm(formula = accept ~ week * group, data = arrange(tbl_9_21,
    id, week), id = id, corstr = "exchangeable")

 Coefficients:
            Estimate Std.err   Wald Pr(>|W|)
(Intercept)   51.727   2.714 363.18   <2e-16 ***
week           0.308   0.637   0.23     0.63
groupC         2.811   3.651   0.59     0.44
week:groupC    0.052   0.852   0.00     0.95
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Correlation structure = exchangeable
Estimated Scale Parameters:

            Estimate Std.err
(Intercept)      474    38.8
  Link = identity

Estimated Correlation Parameters:
      Estimate Std.err
alpha    0.334   0.067
Number of clusters:   109  Maximum cluster size: 6
```

Output 9.20 (page 401)

To obtain the output 9.20, we need to find the mean acceptability by week and treatment groups.

```
tbl_9_21 |>
  group_by(week, group) |>
  summarise(meanAccept = mean(accept, na.rm = TRUE)) -> tbl_9_21_grouped
```

We group the data frame *tbl_9_21* by the variables of interest using `group_by` and then use `summarise` to calculate the mean. Note that we have supplied `na.rm = TRUE` to `mean` to instruct it to remove the `NA` values before calculating the mean. We now use the summarised data frame to build the model object.

```
lm(meanAccept ~ group * week, data = tbl_9_21_grouped)
```

```
Call:
lm(formula = meanAccept ~ group * week, data = tbl_9_21_grouped)

Coefficients:
(Intercept)       groupC         week  groupC:week
   5.20e+01     2.48e+00     2.67e-01     9.04e-04
```

Base R provides `interaction.plot` that can be used to plot spaghetti plots. Here, we will use `ggplot`.

```
ggplot() +
geom_line(aes(x = week, y = accept, group = id),
          colour = "grey",
          data = tbl_9_21) +
labs(x = "Week",
     y = "Acceptability") +
geom_line( aes(x = week, y = meanAccept, colour = group),
           size = 1,
           data = tbl_9_21_grouped) +
scale_colour_manual(labels = c("Normal", "Reduced"),
                    values = c("black", "blue"),
                    name = "Salt")
```

We need `geom_line` to plot lines. We specify the aesthetics, `y` being the value of *accept* and `x` being the value of `week`. The crucial aesthetic is `group`, for which we specify the value *id*. Thus, `ggplot` will plot a different line for each set of rows with the same *id*. Note, that we have specified `colour` as an argument to `geom_line`, but outside `aes`. This means that we want the lines to be drawn with the specified colour, common for all lines. Contrast this with the second `geom_line`, where `colour` is part of `aes`. This results in the lines being drawn with different colours for different values of *group*. Similarly, `size` is specified outside `aes` in the second `geom_line`. This results in both lines having similar thickness, but wider than the lines plotted by the first `geom_line`.

We have used all the rows of *tbl_9_21* in the first `geom_line`. If we want to restrict the number of lines, we should use `slice_head`, `slice_tail` or `slice_sample` to select the rows we desire to be displayed.

9.11 Non-normal alternatives

Example 9.22 (page 402)

The function to perform Kruskal Wallis test is `kruskal.test`. For the example 9.22, the command is

```
kruskal.test(chol ~ diet, data = tbl_9_1)
```

The arguments accepted by `kruskal.test` is the same as `anova` or `lm`, viz. `formula` and `data`.

```

	Kruskal-Wallis rank sum test

data:  chol by diet
```

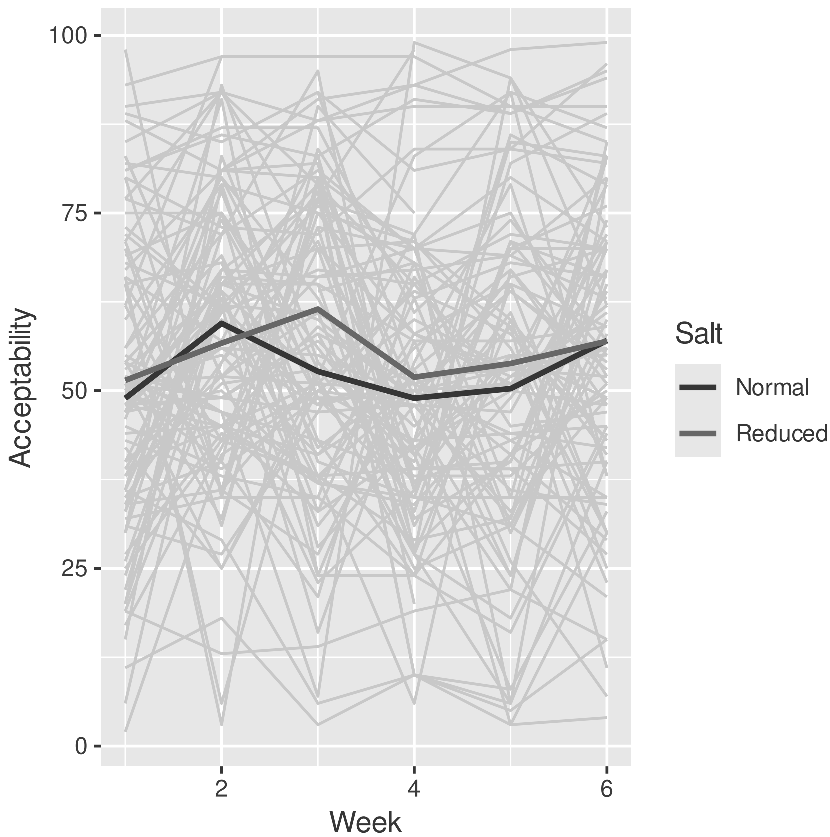


FIGURE 9.15
Spaghetti plot for example 9.21

```
Kruskal-Wallis chi-squared = 13, df = 2, p-value = 0.002
```

The calculated value of the Kruskal Wallis rank sum statistic and its p value are printed. The p value we have is different from that in the textbook. I suspect that the book value is an error.

We end this long chapter with a recap.

9.12 Recap

9.12.1 Concepts discussed in this chapter

- keyword
- attribute
- list
- piping
- comparison operators
- syntactic names
- quote
- identifiers
- nested data frames
- coercion

9.12.2 Commands introduced in this chapter

- readr::col_double
- stats::lm
- stats::anova
- stats::oneway.test
- stats::aov
- stats::confint
- stats::relevel
- stats::TukeyHSD
- dplyr::filter
- ggplot2::theme_bw
- ggplot2::geom_smooth
- stats::predict
- stats::cor
- stats::cor.test
- base::exp
- car::Anova
- base::options
- emmeans::emmeans
- emmeans::pwpm
- emmeans::pairs
- stats::fitted
- ggplot2::ylim
- ggplot2::theme_minimal
- gridExtra::grid.arrange
- base::paste
- base::attr
- broom::tidy
- purrr::map_dfr
- stats::AIC
- broom::glance
- base::ifelse
- stringr::str_detect
- stats::update
- stats::step
- stats::residuals
- stats::rstandard
- sjmisc::rotate_df
- stats::aggregate
- dplyr::n
- tidyr::nest
- purrr::map_dbl
- dplyr::left_join
- base::I
- lspline::lspline
- ggeffects::ggpredict
- splines::ns
- splines::bs
- stats::loess
- geepack::geeglm
- dplyr::arrange
- stats::coef
- readr::col_logical
- stats::kruskal.test

10

Modelling binary outcome data

The main function that we use in this chapter is `glm`, which requires arguments similar to `lm`. In this chapter we use addon packages `ggplot2`, `dplyr`, `readr`, `purrr` all part of `tidyverse`, `Epi`, `broom`, `emmeans` and `gridExtra`. We will start with example 10.4.

10.1 Interpretation of logistic regression coefficients

Example 10.4 (page 415)

The function to perform logistic regression is `glm`. First we will enter the data.

```
factor(c("yes", "no")) -> smoke
c(31 /(31 + 1386), 15 / (15+1883)) -> deaths
glm(deaths ~ smoke,
    family = binomial(link ="logit")) -> lg_10_4
```

The explanatory variable, *smoke* is input as a `factor` and the response variable *deaths* is a numerical vector carrying the proportion of success. We call `glm` with a formula, the left-hand side of which is the response variable and right-hand side is the explanatory variable(s). We need to specify a second parameter `family`, which should be one from a select group of functions. Here we use `binomial`. The function specified as the argument for `family` should be supplied an argument `link`. Though the default link for `binomial` is `logit`, here, we explicitly state it for the sake of clarity. The argument specified as `family` is what makes this call to `glm` a logistic regression. We will see other functions and `link` that can be supplied to `family` later.

The regression model returned by `glm` is stored in *lg_10_4* and inspected with suitable helper functions as we saw earlier. To obtain the odds ratio of smokers versus non smokers, we may use

```
exp(coef(lg_10_4)[2])
```

The function `exp` was used to exponentiate the value returned by `coef`.

```
smokeyes
  2.8077
```

DOI: 10.1201/9781003589563-10

However, if we use `confint`, we will see that R is unable to calculate a confidence interval for the coefficients. This can be solved by changing the way the response variable is supplied.

```
matrix(c(31,1386,15,1883), byrow = TRUE, ncol = 2) -> deaths
glm(deaths ~ smoke, family = binomial(link ="logit")) -> lg_10_4
```

Now, we have constructed the response variable as a matrix with two columns, the first column carrying the number of success and the second carrying the number of failures. We call `glm` with this response matrix. Now, R is able to calculate confidence intervals, which we can use directly or exponentiate to show confidence intervals of odds ratio.

```
exp(confint(lg_10_4)[2,])
```

```
 2.5 % 97.5 %
1.5357 5.3641
```

We may use the coefficients to calculate risks as elaborated in our textbook. For example, `exp(sum(-(coef(lg_10_4))) ^ -1` calculates the risk for smokers. Though we will not calculate the standard error of odds, we should be aware of `vcov` function which returns the variance – covariance matrix of a regression model.

```
vcov(lg_10_4)
```

```
            (Intercept)  smokeyes
(Intercept)    0.067198 -0.067198
smokeyes      -0.067198  0.100177
```

Example 10.5 (page 417)

Fitting logistic regression model for example 10.5 is similar to the previous example.

```
matrix(c(1, 251-1,12, 317-12,13,309-13,6,285-6,10,236-10,8,
         254-8,10,277-10,12,278-12,10,285-10,14,276-14,15,
         274-15,14,296-14,19,305-19,36,341-36,26,305-26,21,
         276-21,28,325-28,41,302-41,38,260-38,49,302-49),
       byrow = TRUE,
       ncol =2) -> deaths
40:59 -> age
glm(deaths ~ age, family = binomial()) -> lg_10_5
```

We prepare the response variable as a two column matrix *deaths* and the explanatory variable *age* as a numerical vector. We call `glm` using these variables in its formula argument while specifying the `family` argument as `binomial` with the default value for `link`.

We can calculate the odds ratio for men aged 59 relative to men aged 40 using

```
exp(coef(lg_10_5)[2] * (59-40))
```

```
   age
8.4874
```

Similarly, the confidence interval for this odds ratio can be calculated using

```
exp(confint(lg_10_5)[2,] * (59-40))
```

```
  2.5 %  97.5 %
 5.7884 12.5733
```

Figure 10.4 (page 418)

To prepare the graphs shown in figure 10.4, we need to prepare a dataframe to pass on to `ggplot`.

```
log(deaths[,1] / deaths[,2]) -> logit.obs
predict(lg_10_5) -> logit.fit
(deaths[,1] / (deaths[,1] + deaths[,2])) * 100 -> prcnt.obs
(( 1+ exp(-coef(lg_10_5)[1] -
            (coef(lg_10_5)[2] * (40:59)))) ^ -1) *   100 -> prcnt.fit
data.frame(age = 40:59, logit.obs,logit.fit,
           prcnt.obs, prcnt.fit) -> data_10_5
```

The observed logit given the name *logit.obs* is calculated as the log of odds using the data entered in *deaths*. The fitted logits is returned by `predict` when we call it with the regression model which we store in *logit.fit*. The observed percentages is calculated from the data in *deaths* and stored as *prcnt.obs*. The fitted percentages are calculated following the details given in the textbook and stored in *prcnt.fit*. We join together these vectors and a vector of ages into a data frame which we call as *data_10_5*. Now, we are in a position to plot the graphs.

```
ggplot(data_10_5) +
  geom_point(aes(x = age, y = logit.obs), shape = 16) +
  geom_point(aes(x = age, y = logit.fit), shape = 4)
```

```
ggplot(data_10_5) +
  geom_point(aes(x = age, y = prcnt.obs), shape = 16) +
  geom_point(aes(x = age, y = prcnt.fit), shape = 4)
```

Both the graphs are made similarly. We use `ggplot`, pass it *data_10_5* as its `data` argument and add `geom_point` twice – once for observed values and once for fitted values. We specify `shape` outside `aes` so that all points in each geom are plotted with the shape of our choice.

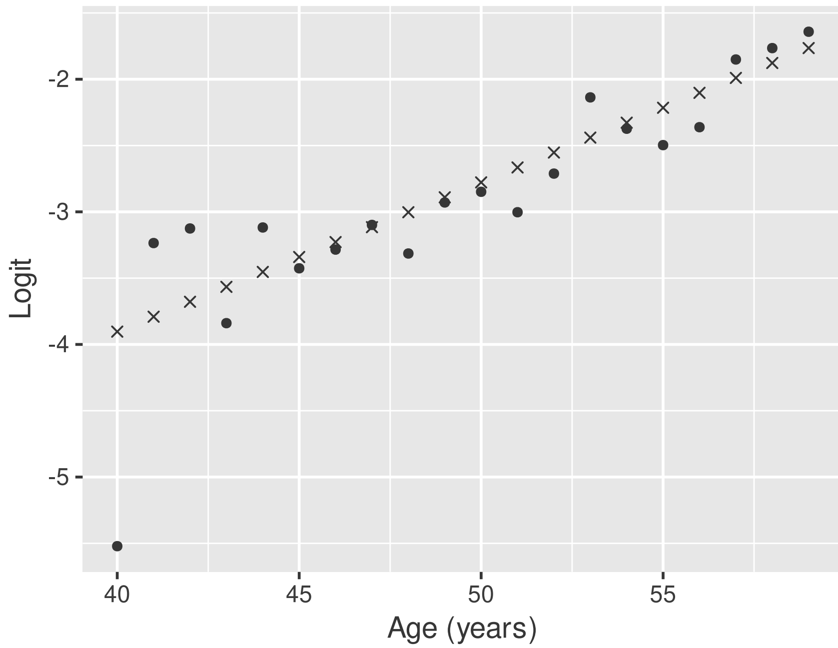


FIGURE 10.1
Replication of figure 10.4(a)

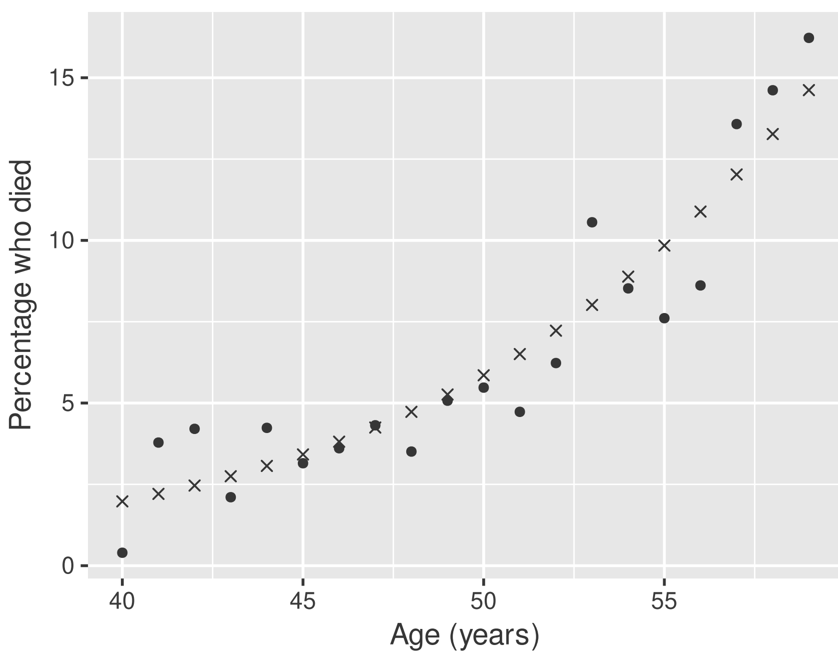


FIGURE 10.2
Replication of figure 10.4(b)

Example 10.6 (page 420)

Reworking example 10.6 is similar to the previous example.

```
matrix(c(10,38-10,40,86-40,36,57-36,226,
         300-226,83,108-83,60,73-60),
       byrow = TRUE,
       ncol =2) -> hpylori
factor(c("I", "II", "IIIn", "IIIm", "IV", "V"),
       levels = c("I", "II", "IIIn", "IIIm", "IV", "V")) -> s.class
glm(hpylori ~ s.class, family = binomial()) -> lg_10_6

summary(lg_10_6)
```

R fixes the first level of factors as zero. Hence, we get the result in table 10.11. We saw `relevel` in the previous chapter which allows us to redefine the base level of a factor. We will not demonstrate it again here. We specify the argument `levels` while constructing *s.class* so that the display order of the factor level is as per our wish. Otherwise, R will arrange factor levels alphabetically by sorting their labels.

```
Call:
glm(formula = hpylori ~ s.class, family = binomial())

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   -1.030      0.368   -2.79  0.00519 **
s.classII      0.890      0.427    2.08  0.03723 *
s.classIIIn    1.569      0.459    3.41  0.00064 ***
s.classIIIm    2.146      0.392    5.47  4.4e-08 ***
s.classIV      2.230      0.433    5.15  2.7e-07 ***
s.classV       2.559      0.479    5.34  9.1e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 6.4435e+01  on 5  degrees of freedom
Residual deviance: 5.7732e-14  on 0  degrees of freedom
AIC: 40.07

Number of Fisher Scoring iterations: 3
```

Figure 10.5 (page 423)

To prepare the graph in figure 10.5, we need to prepare the data.

```
as.data.frame(exp(cbind( lg_10_6$coefficients[-1],
                         confint(lg_10_6, level = 0.95)[-1,]))) -> data_10_6
as.data.frame(rbind(c(or =1, ll =NA, ul = NA),
                    rename(data_10_6,
                           ll = `2.5 %`,
                           ul = `97.5 %`,
                           or = `V1`))) -> data_10_6
ggplot(data_10_6) +
  geom_pointrange(aes(x = s.class, y = or, ymin = ll, ymax = ul)) +
  labs(x = "Social class",
       y = "Odds ratio")
```

We use `cbind` to bind together the coefficients component of the model object and the result returned by `confint`. We remove the values corresponding to the intercept by means of negative subsetting before joining the two. We exponentiate the resultant values which is then converted to a data frame using `as.data.frame`. Thus we get a dataframe with three columns corresponding to the odds ratio and its confidence intervals. In the next step, we add a row with only the value of *or* assigned to 1 and that of *ul* and *ll* assigned as `NA` to *data_10_6* obtained in the previous step. Before adding the new row, the awkward names of the columns obtained by `cbind` are renamed. We then call `ggplot` with this dataframe. We use `geom_pointrange` to draw the confidence interval.

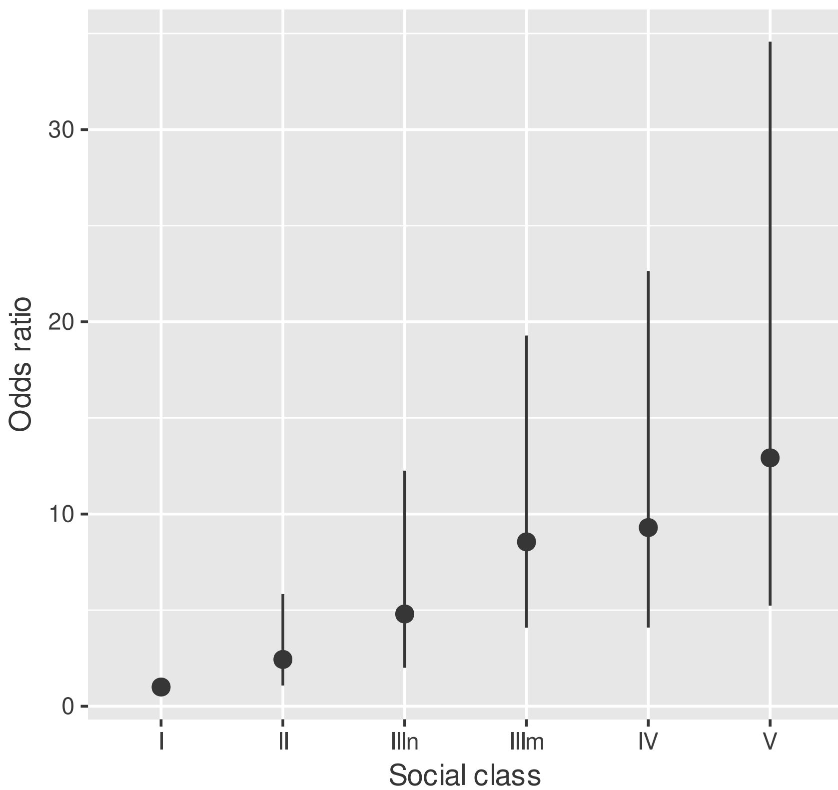


FIGURE 10.3
Replication of figure 10.5

Figure 10.6 (page 425)

To fit against the rank of social classes, we use a numerical vector as the explanatory variable.

```
s.rank <- 1:6
glm(hpylori ~ s.rank, family = binomial()) -> lgo_10_6
```

To fix the odds of the social class rank 1 as unity, we need to adjust the coefficients returned by the regression model so that the coefficient of rank 1 is zero. We can then use the adjusted values to prepare the graphs in figure 10.6

```
as.data.frame(cbind(lodd.adj = predict(lgo_10_6) +
                      (-1 * predict(lgo_10_6)[1]),
                    rank = 1:6)) -> data_10_6f

ggplot(data_10_6f) +
  geom_line(aes(x = rank, y =lodd.adj)) +
  labs(x = "Social class rank",
       y = "Log odds ratio")

ggplot() +
  xlim(1,6) +
  geom_function(fun = function(x){exp(lgo_10_6$coefficients[2] * (x -1))}) +
    labs(x = "Social class rank",
       y = "Odds ratio")
```

We use `predict` to obtain the fitted values for all cases. We make a data frame by binding together these values with the ranks. The resultant data frame is used to draw the graph using `ggplot` and `geom_line`. For the second graph, we use `geom_function` and supply as its argument `fun`, an anonymous function. This function will accept a value x corresponding to the x axis values and return a corresponding value for the y axis. A smooth line joins the points thus calculated. Our anonymous function uses the ordinal model coefficient, multiplies its slope parameter with x minus 1. We deduct one from the rank as the reference for calculating odds ratio is rank 1. The function `xlim` determines the range of the x axis of the graph.

10.1.1 Floating absolute risks

The package `Epi` has a function `float` that implements floating absolute risks. Remember to install it using `install.packages` as discussed in chapter 1.

```
library(Epi)
float(lg_10_6) -> far_10_6
est <- exp(far_10_6$coef)
ul <- exp(far_10_6$coef +  (qnorm(1-0.05/2) * sqrt(far_10_6$var)))
ll <- exp(far_10_6$coef -  (qnorm(1-0.05/2) *sqrt(far_10_6$var)))
```

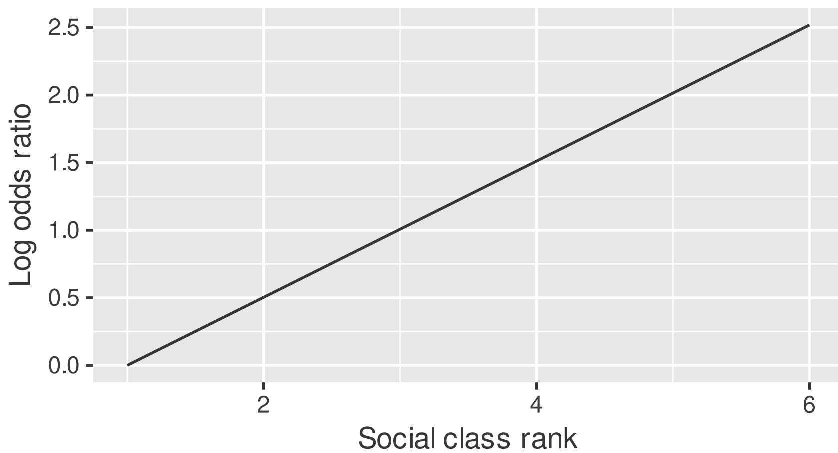


FIGURE 10.4
Replication of figure 10.6 (a)

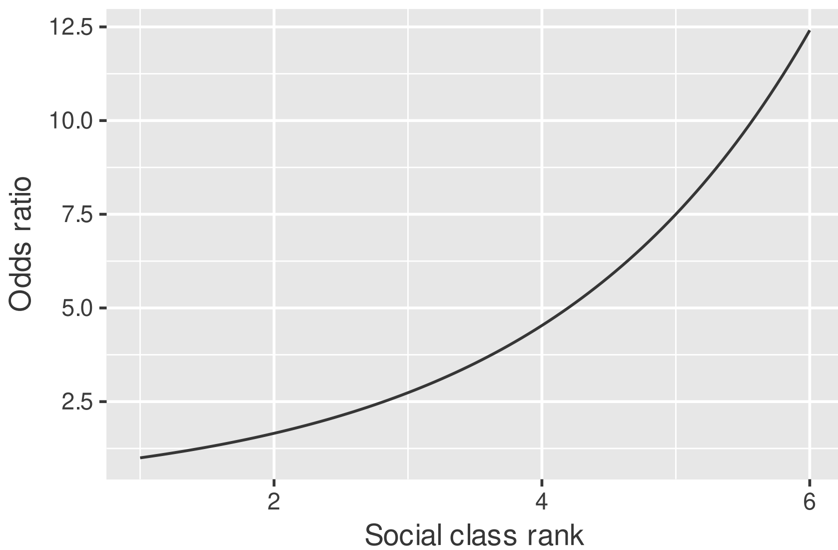


FIGURE 10.5
Replication of figure 10.6 (b)

```
ggplot(cbind.data.frame(est,ul,ll)) +
  geom_pointrange(aes(x = s.class, y = est,ymin = ll, ymax = ul)) +
  labs(x = "Social class",
       y = "Odds ratio")
```

The argument that we provide to `float` is the fitted model. If there are multiple explanatory variables in the model, we need to specify `factor`, the variable for which floating absolute risks need to be calculated. The result returned by `float` is a list, of which we use `coef` and `var` to calculate the FAR odds ratios and their confidence interval by adding and subtracting the product of standard error and appropriate values returned by `qnorm` from the FAR estimates. All these values are exponentiated to convert to odds ratio. These are then bound together using `cbind.data.frame` while calling `ggplot` to use `geom_pointrange` to plot the data.

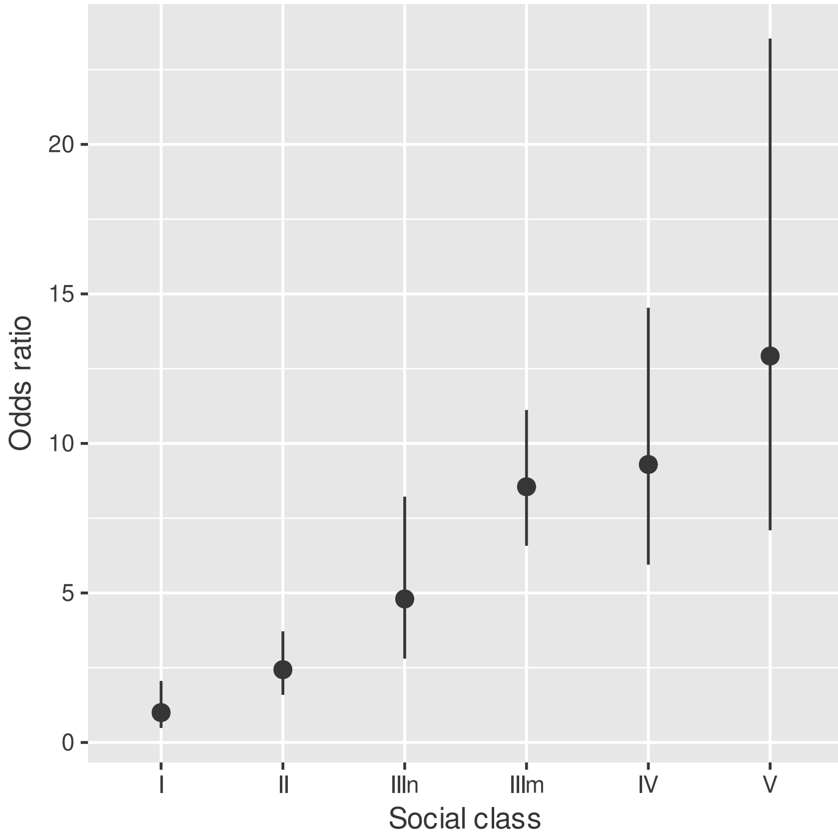


FIGURE 10.6
Replication of figure 10.7

10.2 Generic data

Example 10.10 (page 427)

To fit the generic data of example 10.10, we will import the data first.

```
read_table("K11828 supplements/Datasets/Example 10.10.DAT",
           col_names = c("age", "death"),
           col_types = cols(death = col_factor())) ->tbl_10_10
glm(death ~ age, tbl_10_10, family = binomial()) -> lg_10_10
```

The call to `glm` is no different when data is given in generic form.

```
summary(lg_10_10)
```

We can see that R uses a standard normal test when we pass the model object to `summary`.

```
Call:
glm(formula = death ~ age, family = binomial(), data = tbl_10_10)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -8.4056     0.5507   -15.3   <2e-16 ***
age           0.1126     0.0104    10.8   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 2815.5  on 5753  degrees of freedom
Residual deviance: 2683.7  on 5752  degrees of freedom
AIC: 2688

Number of Fisher Scoring iterations: 6
```

10.3 Multiple logistic regression models

Example 10.11 (page 428)

To rework the example 10.11, we need the data.

```
matrix(c(1,190-1,0,183,4,178-4,8,157-8,4,132-4,2,203-2,2,175-2,
         6,167-6,10,166-10,11,137-11,5,173-5,9,176-9,9,181-9,
         8,167-8,11,164-11,5,139-5,3,156-3,10,154-10,13,174-13,
         16,174-16,5,123-5,8,123-8,12,144-12,13,179-13,23,180-23),
       byrow = TRUE,
       ncol = 2) -> chd
gl(5,1,25,
   labels = c("<=5.41","5.42-6.01","6.02-6.56","6.57-7.31",">7.31")) -> chol

gl(5,5,25,
   labels = c("<=118", "119-127","128-136","137-148",">148")) -> sbp
glm(chd ~ chol + sbp, family = binomial()) -> lg_10_11
```

As the data is grouped, we type in the CHD data of table 10.14 as a matrix with two columns. The corresponding values for cholesterol group and systolic BP group are built as factors using `gl`. The first argument that `gl` requires is the number of levels that the factor has. In our case, both *chol* and *sbp* have five levels. The next argument specifies the number of replications, the number of times each level has to be replicated. In the case of *chol*, the adjacent values in *chd* correspond to different cholesterol groups. So, we specify the number of replications as one. In the case of *sbp*, five adjacent values of *chd* belong to the same systolic blood pressure group. So, the number of replications required to build *sbp* is five. The third argument is the length of the vector required. For both *chol* and *sbp*, we need 25. For *sbp*, the `length` calculated from the first two arguments is what we require and so there really is no need to specify `length`. The argument `labels` is optional. We supply a character vector containing the display label for each level of the factor. Building the model object using `glm` is no different from the previous examples.

Instead of calculating the logit, odds and relative risks for specific examples, we will calculate them for all combinations of the explanatory variables.

```
data.frame(chol, sbp,
           logit = predict(lg_10_11,
                          newdata = data.frame(chol,sbp))) -> pred_10_11
```

We build a data frame using `data.frame`, which joins together its arguments into one data frame. We provide *chol*, *sbp* and another vector returned by `predict` to be named as *logit*. As we saw earlier, `predict` will return predicted values for the values of explanatory variables we supply based on its first argument, a model object. The values of explanatory variables need to be provided as a dataframe with the same column names as in the model object. Here, we join together *chol* and *sbp*. Thus, the data frame supplied to `predict` contains all combinations of both explanatory variables. We store the data frame obtained by joining the values returned by `predict` and *sbp* and *chol* with the name *pred_10_11*.

We may now use `mutate` to calculate odds and risk ratios from the logit values.

```
pred_10_11 |>
  mutate(odds = exp(logit),
         risk = (1 + exp(abs(logit))) ^ -1 ) -> pred_10_11
pred_10_11
```

```
        chol     sbp   logit     odds      risk
1      <=5.41   <=118 -4.5995 0.010057 0.0099569
2   5.42-6.01   <=118 -4.3906 0.012394 0.0122421
3   6.02-6.56   <=118 -3.7766 0.022901 0.0223879
4   6.57-7.31   <=118 -3.5929 0.027518 0.0267811
5       >7.31   <=118 -3.3038 0.036743 0.0354409
6      <=5.41 119-127 -3.9903 0.018494 0.0181578
7   5.42-6.01 119-127 -3.7814 0.022791 0.0222829
8   6.02-6.56 119-127 -3.1674 0.042111 0.0404095
9   6.57-7.31 119-127 -2.9838 0.050602 0.0481650
10      >7.31 119-127 -2.6947 0.067566 0.0632897
```

```
11    <=5.41 128-136 -3.7297 0.023999 0.0234367
12 5.42-6.01 128-136 -3.5208 0.029576 0.0287260
13 6.02-6.56 128-136 -2.9068 0.054648 0.0518161
14 6.57-7.31 128-136 -2.7232 0.065667 0.0616201
15     >7.31 128-136 -2.4341 0.087680 0.0806122
16    <=5.41 137-148 -3.5698 0.028161 0.0273898
17 5.42-6.01 137-148 -3.3609 0.034705 0.0335406
18 6.02-6.56 137-148 -2.7469 0.064125 0.0602606
19 6.57-7.31 137-148 -2.5632 0.077054 0.0715418
20     >7.31 137-148 -2.2741 0.102886 0.0932879
21    <=5.41    >148 -3.2570 0.038504 0.0370767
22 5.42-6.01    >148 -3.0481 0.047451 0.0453015
23 6.02-6.56    >148 -2.4341 0.087677 0.0806093
24 6.57-7.31    >148 -2.2504 0.105355 0.0953136
25     >7.31    >148 -1.9613 0.140674 0.1233256
```

Now, getting the logit, odds or risk of any combination is easy.

```
subset(pred_10_11, chol==">7.31" & sbp==">148")
```

The function `subset` returns a subset of its first argument that satisfies the logical condition given as the second argument. We joined the two parts of our logical condition using `&`. `subset` supports an argument `select` which can be used to restrict the columns printed, if required.

```
    chol  sbp   logit    odds    risk
25 >7.31 >148 -1.9613 0.14067 0.12333
```

Calculating odds ratios and risk ratios are also easy.

```
subset(pred_10_11, chol==">7.31" & sbp==">148")[["odds"]] /
  subset(pred_10_11, chol=="<=5.41" & sbp=="<=118")[["odds"]]
```

```
subset(pred_10_11, chol==">7.31" & sbp==">148")[["risk"]] /
  subset(pred_10_11, chol=="<=5.41" & sbp=="<=118")[["risk"]]
```

We use `subset` to obtain the row of our interest. Then we use `[[` and `]]` to get the numerical value inside the column of our interest. If we use the `select` argument of `subset`, a dataframe with the numerical value is returned, which we cannot use in division. Similarly, if we use single square bracket `[` and `]`, we get a dataframe. Subsetting using double square brackets `[[ ]]` is essential when we want to get the value inside a component of lists, dataframe included.

```
[1] 13.988
```

```
[1] 12.386
```

We will not discuss confidence intervals now.

Example 10.12 (page 431)

We import the relevant data to rework the example 10.12.

```
read_table("K11828 supplements/Datasets/Example 10.12.DAT",
           col_names = c("age", "chol", "bmi", "sbp",
                         "smoke", "active", "chd", "nmbr"),
           col_types = cols( smoke = col_factor(levels = c("1","2","3")),
                             active = col_factor(levels = c("1","2","3")),
                             chd = col_factor(levels = c("0","1")),
                             nmbr = col_skip())) -> data_10_12
glm(chd ~ age + chol + bmi + sbp + smoke + active,
    data = data_10_12,
    family = binomial()) -> lg_10_12
```

We have asked R to import the data treating *smoke*, *active* and *chd* as factors. We have specified the levels of the factor within `col_factor`. As we don't want the final column, we use `col_skip` to avoid importing it.

```
coef(lg_10_12)
```

We use `coef` to print the parameter estimates to confirm that they are the same as in the textbook.

```
(Intercept)         age        chol         bmi         sbp      smoke2
 -10.107560    0.017105    0.307075    0.041656    0.020390    0.322548
     smoke3     active2     active3
   0.729598   -0.190420   -0.101058
```

To calculate the logit for different values of the explanatory variables, we can multiply the vector of parameter estimates by a vector with the values of explanatory variables given in the same sequence and add them up.

```
sum(coef(lg_10_12) * c(1,50,6,25,125,1,0,0,0))
```

Note that we multiply the value of intercept with one. Only that level of a factor which applies is multiplied with one and others with zero. The logit for a 50 year old active ex-smoker with cholesterol 6.0 units and systolic BP 125 units agree with the textbook value except for rounding error.

```
[1] -3.4972
```

10.4 Tests of hypothesis

Section 10.7.1 (page 433)

The deviance of a model is obtained using `deviance`.

```
deviance(lg_10_12)
```

```
[1] 1481.3
```

The model deviance degrees of freedom is obtained using `df.residual`.

```
df.residual(lg_10_12)
```

```
[1] 4040
```

The p value of the goodness of fit chi square test can be obtained using the values returned by these functions.

```
pchisq(deviance(lg_10_5),
       df.residual(lg_10_5),
       lower.tail = FALSE)
```

```
[1] 0.16062
```

The Hosmer-Lemeshaw goodness of fit test for generic data is available through `HosmerLemeshowTest` function of `DescTools`.

Example 10.13 (page 436)

To rework the example 10.13, we need to fit the null model.

```
glm(hpylori ~ 1, family = binomial()) -> lg_10_6_null
glm(hpylori ~ s.class, family = binomial()) -> lg_10_6_full

deviance(lg_10_6_null) - deviance(lg_10_6_full) -> ddev_f_n
df.residual(lg_10_6_null) - df.residual(lg_10_6_full) ->   ddf_f_n
```

We calculate the difference in deviance and difference in degrees of freedom between the two models and use those values to calculate the p value of chi square test.

```
pchisq(ddev_f_n, ddf_f_n, lower.tail = FALSE)
```

```
[1] 1.4679e-12
```

It is easier to obtain the p value of chi square goodness of fit test using `anova`.

```
anova(lg_10_6_null,lg_10_6_full, test = "Chisq")
```

```
Analysis of Deviance Table

Model 1: hpylori ~ 1
Model 2: hpylori ~ s.class
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1         5       64.4
2         0        0.0  5     64.4  1.5e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Example 10.14 (page 437)

To rework the example 10.14, we build the models to be considered.

```
glm(chd ~ 1, family = binomial()) -> lg_10_11_null
glm(chd ~ sbp, family = binomial()) -> lg_10_11_sbp
glm(chd ~ chol, family = binomial()) -> lg_10_11_chol
glm(chd ~ chol + sbp, family = binomial()) -> lg_10_11_full
anova(lg_10_11_null, lg_10_11_sbp,lg_10_11_chol, lg_10_11_full,
      test = "Chisq")
```

The analysis of deviance table is constructed by `anova` from all the model objects passed to it. Note that, models are compared only with the preceding. As the third model is not nested in the second, they are not compared.

```
Analysis of Deviance Table

Model 1: chd ~ 1
Model 2: chd ~ sbp
Model 3: chd ~ chol
Model 4: chd ~ chol + sbp
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1        24       94.6
2        20       56.7  4     37.8  1.2e-07 ***
3        20       49.5  0      7.3
4        16       18.9  4     30.6  3.7e-06 ***
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Further analysis as given in table 10.21 needs to be done separately for each comparison. For example,

```
pchisq(deviance(lg_10_11_null) - deviance(lg_10_11_sbp),
       df.residual(lg_10_11_null) - df.residual(lg_10_11_sbp),
       lower.tail = FALSE)
```

Here we calculated the p value of the goodness of fit chi square test using the difference in deviance of the model with systolic blood pressure from the null model, corresponding to the first row of table 10.21.

```
[1] 1.2052e-07
```

Probably, it is easier to specify two `anova` commands.

```
anova(lg_10_11_null, lg_10_11_sbp, lg_10_11_full, test = "Chisq")
anova(lg_10_11_null, lg_10_11_chol, lg_10_11_full, test = "Chisq")
```

```
Analysis of Deviance Table

Model 1: chd ~ 1
Model 2: chd ~ sbp
Model 3: chd ~ chol + sbp
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1        24       94.6
2        20       56.7  4     37.8  1.2e-07 ***
3        16       18.9  4     37.9  1.2e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Analysis of Deviance Table

Model 1: chd ~ 1
Model 2: chd ~ chol
Model 3: chd ~ chol + sbp
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1        24       94.6
2        20       49.5  4     45.1  3.8e-09 ***
3        16       18.9  4     30.6  3.7e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We will not rework the example 10.15 as the steps involved are as in the previous examples. The functions `AIC`, `BIC` and `step` that we saw in the previous chapter can be used to calculate the information criteria and for automatic model selection.

Example 10.17 (page 441)

We now turn to example 10.17. We will calculate the p value for the test of non linearity and for linear trend.

```
glm(hpylori ~ 1, family = binomial()) -> lg_10_6_null
glm(hpylori ~ s.class, family = binomial()) -> lg_10_6_full
glm(hpylori ~ s.rank, family = binomial()) -> lg_10_6_ord

anova(lg_10_6_null, lg_10_6_ord, lg_10_6_full, test = "Chisq")
anova(lg_10_6_null,lg_10_6_full, test = "Chisq" )
```

The calculation is no different from the previous examples and yield the values in table 10.25.

```
Analysis of Deviance Table

Model 1: hpylori ~ 1
Model 2: hpylori ~ s.rank
Model 3: hpylori ~ s.class
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1         5       64.4
2         4        6.5  1     58.0  2.7e-14 ***
3         0        0.0  4      6.5     0.17
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Analysis of Deviance Table

Model 1: hpylori ~ 1
Model 2: hpylori ~ s.class
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1         5       64.4
2         0        0.0  5     64.4  1.5e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Example 10.18 (page 441)

To rework the example 10.18, we need to build model objects with polynomial effects.

```
glm(hpylori ~ poly(s.rank, degree = 2), family = binomial()) -> lg_10_6_p2
glm(hpylori ~ poly(s.rank, degree = 3), family = binomial()) -> lg_10_6_p3
glm(hpylori ~ poly(s.rank, degree = 4), family = binomial()) -> lg_10_6_p4
glm(hpylori ~ poly(s.rank, degree = 5), family = binomial()) -> lg_10_6_p5
```

We use the function `poly` as the right-hand side of the formula in all our model specifications above. The function `poly` expands the variable to power terms to the degree specified. Thus `poly(s.rank, degree = 2)` is equivalent to $\text{s.rank}^1 + \text{s.rank}^2$.

```
anova(lg_10_6_null,lg_10_6_ord, lg_10_6_p2,
      lg_10_6_p3, lg_10_6_p4, lg_10_6_p5,
      test = "Chisq")
```

The anova table prepared using the polynomial models give results similar to that in table 10.26.

```
Analysis of Deviance Table

Model 1: hpylori ~ 1
Model 2: hpylori ~ s.rank
Model 3: hpylori ~ poly(s.rank, degree = 2)
Model 4: hpylori ~ poly(s.rank, degree = 3)
Model 5: hpylori ~ poly(s.rank, degree = 4)
Model 6: hpylori ~ poly(s.rank, degree = 5)
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1         5       64.4
2         4        6.5  1     58.0  2.7e-14 ***
3         3        0.7  1      5.8    0.016 *
4         2        0.7  1      0.0    0.835
5         1        0.2  1      0.4    0.510
6         0        0.0  1      0.2    0.638
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Output 10.2 (page 443)

To make the table in output 10.2, we will use the `broom` library.

```
library(broom)
tidy(lg_10_12, conf.int = TRUE) |>
  mutate(chisqr = (estimate / std.error) ^ 2,
         `Pr > ChiSq` = pchisq(chisqr,1, lower.tail = FALSE))
```

We use `tidy` to get a neat dataframe of the data returned by `summary` when a model object is passed to it. The argument `conf.int` is specified as `TRUE` so that we get the confidence intervals. We add two columns to the data frame returned by `tidy` to calculate the value of chi square and its probability according to the formula given by the textbook. Note that the p value returned by the chi square test and the normal test done by default are the same.

```
# A tibble: 9 x 8
  term   estimate statistic  p.value conf.low conf.high  chisqr `Pr > ChiSq`
  <chr>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>   <dbl>        <dbl>
```

```
1 (Inte~ -10.1      -10.1   3.81e-24 -1.21e+1   -8.17   103.        3.81e-24
2 age      0.0171     1.25  2.10e- 1 -9.56e-3    0.0439   1.57      2.10e- 1
3 chol     0.307      5.14  2.68e- 7  1.89e-1    0.424   26.5       2.68e- 7
4 bmi      0.0417     1.95  5.13e- 2 -7.94e-4    0.0830   3.80      5.13e- 2
5 sbp      0.0204     5.33  9.81e- 8  1.28e-2    0.0278  28.4       9.81e- 8
6 smoke2   0.323      1.29  1.98e- 1 -1.62e-1    0.825    1.66      1.98e- 1
7 smoke3   0.730      3.33  8.72e- 4  3.16e-1    1.18    11.1       8.72e- 4
8 activ~  -0.190     -1.06  2.90e- 1 -5.37e-1    0.170    1.12      2.90e- 1
9 activ~  -0.101     -0.433 6.65e- 1 -5.65e-1    0.353    0.187     6.65e- 1
```

Table 10.27 (page 445)

To prepare table 10.27, we need to exponentiate the coefficients of the relevant model.

```
cbind.data.frame(unadjusted = exp(coef(lg_10_11_chol))[2:5],
                 adjusted = exp(coef(lg_10_11_full))[2:5])
```

We have used `coef` to obtain the coefficients, passed it on to `exp` to exponentiate the coefficients and subset the relevant values. We build a data frame by using `cbind.data.frame` and passing to it the exponentiated coefficients from the two models we are comparing.

TABLE 10.1
Replication of table 10.27

	Odds ratio	
Serum total cholesterol fifth	Unadjusted	Adjusted
chol5.42-6.01	1.2516	1.2324
chol6.02-6.56	2.3563	2.2771
chol6.57-7.31	2.9583	2.7362
chol>7.31	4.0512	3.6535

10.5 Interaction

Example 10.22 (page 446)

Building a model with interaction terms is similar to previous examples. First, the data.

```
matrix(c(57,1022,56,918,39,927,46,1022,
         30,915,20,1081,10, 1066,10,938),
       byrow= TRUE,
       ncol = 2) -> chd
factor(rep(c("<=144", "145-169", "170-194", ">=194"),2)) -> bortner
factor(c(rep("male",4), rep("female",4))) -> sex
glm(chd ~ 1, family = binomial()) -> lg_10_22_null
glm(chd ~ bortner, family = binomial()) -> lg_10_22_bort
```

```
glm(chd ~ sex , family = binomial()) -> lg_10_22_sex
glm(chd ~ bortner + sex, family = binomial()) -> lg_10_22_bs
glm(chd ~ bortner * sex, family = binomial()) -> lg_10_22_bsi

anova(lg_10_22_null, lg_10_22_bort,lg_10_22_bs, lg_10_22_bsi,
      test = "Chisq")
```

We use a two column matrix as our response variable *chd*. The explanatory variables are factors with length as much as the number of rows in the explanatory variable. We call `glm` to build the appropriate model objects. The formula used to specify interaction uses `*` between the variables of interest. There is no need to build the dummy variables ourselves. Using `anova`, we build the anova table. We haven't included the "Bortner given sex" row from table 10.32. We should use a separate `anova` command if we wish to calculate that too. Note that though we call `anova` with four models, it is sufficient to call `anova` with just the last model.

```
Analysis of Deviance Table

Model 1: chd ~ 1
Model 2: chd ~ bortner
Model 3: chd ~ bortner + sex
Model 4: chd ~ bortner * sex
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1         7       86.6
2         4       72.5  3     14.1   0.0027 **
3         3        8.1  1     64.3    1e-15 ***
4         0        0.0  3      8.1   0.0431 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Table 10.34 (page 448)

We will use `predict` to build table 10.34.

```
data.frame(bortner, sex,
           odds = exp(predict(lg_10_22_bsi,
                              newdata = data.frame(bortner,
                                                   sex)))) -> pred_10_22
cbind.data.frame(Bortner = c("<=144", "145-169", "170-194", ">=194"),
                 Males = subset(pred_10_22,sex == "male")[["odds"]] /
                   subset(pred_10_22,
                          bortner =="<=144" &sex == "male")[["odds"]],
                 Females = subset(pred_10_22,
                                  sex == "female")[["odds"]] /
                   subset(pred_10_22,
                          bortner =="<=144" &sex == "female")[["odds"]])
```

We build a data frame using the exponentiated values returned by `predict` when it is passed a data frame containing all combinations of *bortner* and *sex*. To calculate odds ratio, we subset the two sexes separately and then divide them with the reference values. We collect the result in an unnamed data frame for printing.

TABLE 10.2
Replication of table 10.34

	Sex	
Bortner quarter	Male	Female
<=144	1.00000	1.00000
145-169	1.09376	0.56429
170-194	0.75433	0.28612
>=194	0.80702	0.32516

We may use `tidy(lg_10_22_bsi, conf.int = 0.95)` to get an output similar to output 10.3 except that rows with estimate zero won't be printed.

Example 10.23 (page 449)

We now turn to example 10.23. First, we need the data.

```
read_table("K11828 supplements/Datasets/Example 10.23.DAT",
           col_names = c("sex", "bscore", "bqrtr", "chd", "survive"),
           col_types = cols(sex = col_factor(levels = c("1","2")),
                            bqrtr = col_factor(levels = c("1","2","3","4")),
                            chd = col_factor(levels = c("0","1")))
           ) -> data_10_23
glm(chd ~ bscore * sex,
    data = data_10_23,
    family = binomial()) -> lg_10_23
anova(lg_10_23,  test = "Chisq")
```

When we import the data, we need to specify which of the variables are factors. We fit only the full model as in the previous examples. We pass that model to `anova` to build the anova table. If we need the missing rows, we need to build the model with the predictors in the alternate order and pass it to `anova`.

```
Analysis of Deviance Table

Model: binomial, link: logit

Response: chd

Terms added sequentially (first to last)

           Df Deviance Resid. Df Resid. Dev Pr(>Chi)
NULL                        8156       2358
bscore      1     10.3      8155       2348   0.0013 **
```

```
sex         1     64.5      8154       2283  9.5e-16 ***
bscore:sex  1      8.5      8153       2275   0.0036 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Output 10.4 (page 451)

We may use `summary` or `tidy` to build the output 10.4

```
tidy(lg_10_23, conf.int = TRUE)
```

```
# A tibble: 4 x 7
  term        estimate std.error statistic  p.value conf.low conf.high
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
1 (Intercept) -2.59      0.301       -8.61 7.59e-18 -3.19     -2.01
2 bscore      -0.00231   0.00177     -1.31 1.90e- 1 -0.00577   0.00115
3 sex2         0.627     0.582        1.08 2.82e- 1 -0.533     1.75
4 bscore:sex2 -0.0106    0.00364     -2.92 3.47e- 3 -0.0177   -0.00349
```

Though the coefficients have values different from output 10.4, the calculated logits are similar to output 10.4. The difference is due to the different reference level used for *sex*.

Changing the reference level of a factor is easy as we saw in an earlier example. We will rebuild the model after changing the reference level of *sex*.

```
relevel(data_10_23$sex, ref = 2) -> data_10_23$sex
tidy(glm(chd ~   bscore * sex,
         data = data_10_23,
         family = binomial()),
     conf.int = TRUE)
```

```
# A tibble: 4 x 7
  term        estimate std.error statistic   p.value conf.low conf.high
  <chr>          <dbl>     <dbl>     <dbl>     <dbl>    <dbl>     <dbl>
1 (Intercept)  -1.96     0.498       -3.94 0.0000806 -2.97     -1.02
2 bscore       -0.0129   0.00318     -4.07 0.0000470 -0.0192   -0.00669
3 sex1         -0.627    0.582       -1.08 0.282     -1.75      0.533
4 bscore:sex1   0.0106   0.00364      2.92 0.00347    0.00349   0.0177
```

When the reference level is 2, the code for females, the coefficients for sex equals 2 is zero. Hence, the coefficient for the interaction term involving sex equals 2 is also zero. Thus the only coefficient contributing towards logit is *bscore* and the intercept. The intercept's value gets cancelled and thus only *bscore* becomes relevant to calculating the logit and its confidence interval. Thus, we need to use the confidence interval of *bscore* alone as the confidence interval of logits for the *sex* that was used as the reference level.

10.6 Dealing with a quantitative explanatory variable

Example 10.24 (page 454)

To categorise continuous variables, we need `cut`. In addition to the vector carrying the continuous variable, it accepts a `break` argument, which is a vector containing the boundary values for the categories. There are additional logical arguments `right` which determines whether the intervals are closed on the right or not and `include.lowest` which determines whether the value that equals the open boundary should be included or not.

```
cut(data_10_12$sbp,
    quantile(data_10_12$sbp, probs = seq(0, 1, 0.25)),
    include.lowest = TRUE) |> table()
```

We provide to `cut`, the vector containing the systolic blood pressure. We calculate the quantiles of the same vector using `quantile` and provide them as the `breaks` argument of `cut`. To calculate the values of the quantiles, `quantile` needs an argument `probs`, which we supply as the sequence from 0 to 1 at steps of 0.25. As, the vector of values returned by `quantile` includes the minimum and because `cut` will not include the lowest value in a category by default, we say `include.lowest=TRUE`. Finally, the vector thus cut into categories is piped into `table` to confirm that the results we get are consistent with that in table 10.36.

```
 [82,121] (121,131] (131,143] (143,233]
     1083       960       995      1011
```

To categorise the vector according to the quarters defined by events, all we need is change the `breakpoints`. We can obtain the quartiles if we filter the data to include only those rows with the value 1 in the *chd* column.

```
quantile(filter(data_10_12,chd == 1)$sbp,
         probs = seq(0, 1, 0.25))
```

```
  0%  25%  50%  75% 100%
  97  128  138  151  210
```

However, the minimum and the maximum returned by `quantile` is affected by the restriction caused by filtering of the rows. Thus, the minimum and maximum returned doesn't apply to the whole data set. So, we need to specify the `breaks` manually incorporating the values returned by `quantile` changing the minimum and maximum to reflect that of the entire dataset.

```
cut(data_10_12$sbp,
```

```
    c(82,128,138,151,233),
    include.lowest = TRUE) |> table()
```

The result we get is different from that in table 10.37. I suspect that there has been a mistake in the textbook.

```
 [82,128] (128,138] (138,151] (151,233]
     1765       934       757       593
```

10.6.1 Linear spline model

To fit the linear spline model, we will use `lspline`. Remember to install it using `install.packages` as discussed in chapter 1.

Example 10.25 (page 455)

```
glm(chd ~  lspline(sbp , knots = c(121,131,143)),
    family = binomial(),
    data = data_10_12) -> ls_10_25
glm(chd ~  lspline(sbp , knots = c(121,131,143), marginal = TRUE),
    family = binomial(),
    data = data_10_12) -> ls_10_25_m
tidy(ls_10_25)
tidy(ls_10_25_m)
```

Similar to what we saw in the previous chapter, we use `lspline` as the right-hand side of the model formula. We provide `lspline` with the `knots` we want; as a numeric vector. We build two models, the first one where the coefficients represent the slope of each segment of the linear spline. In the second model, which is specified with `marginal=TRUE`, the intercept as well as the first slope coefficient are the same as in the first model. However, the subsequent coefficients are change in slope compared with the previous. If we desire a joint table like 10.38, we can combine together the coefficients of the two models using `cbind`, either directly or after `tidy`ing it.

```
# A tibble: 5 x 5
  term                                    estimate std.error statistic p.value
  <chr>                                      <dbl>     <dbl>     <dbl>   <dbl>
1 (Intercept)                             -1.03e+1   3.89       -2.64  0.00830
2 lspline(sbp, knots = c(121, 131, 143~  5.69e-2   0.0333      1.71  0.0876
3 lspline(sbp, knots = c(121, 131, 143~  4.76e-2   0.0325      1.46  0.143
4 lspline(sbp, knots = c(121, 131, 143~  8.21e-3   0.0218      0.376 0.707
5 lspline(sbp, knots = c(121, 131, 143~  2.03e-2   0.00704     2.88  0.00392
# A tibble: 5 x 5
  term                                    estimate std.error statistic p.value
  <chr>                                      <dbl>     <dbl>     <dbl>   <dbl>
1 (Intercept)                             -1.03e+1    3.89      -2.64  0.00830
```

```
2 lspline(sbp, knots = c(121, 131, 143~  5.69e-2    0.0333     1.71  0.0876
3 lspline(sbp, knots = c(121, 131, 143~ -9.29e-3    0.0574    -0.162 0.871
4 lspline(sbp, knots = c(121, 131, 143~ -3.94e-2    0.0490    -0.803 0.422
5 lspline(sbp, knots = c(121, 131, 143~  1.21e-2    0.0261     0.464 0.643
```

Figure 10.10 (page 456)

To prepare the graph in figure 10.10, we need to prepare the data first. The dataframes that we use to build the model objects can't be used directly for preparing the graphs as they don't have a column of logit. Thus, our first step is to add the logit and its upper and lower bounds to the dataframes.

```
cbind.data.frame(data_10_12,
                 predict(glm(chd ~ sbp,
                             data = data_10_12,
                             family = binomial()),
                         se.fit = TRUE)) |>
  mutate (uci = fit + (1.96 * se.fit),
          lci = fit - (1.96 * se.fit)) -> pred_lg_10_25sbp


cbind.data.frame(data_10_12,
                 predict(glm(chd ~ cut(data_10_12$sbp,
                                       breaks = c(50,121,143,250)),
                             data = data_10_12,
                             family = binomial()),
                         se.fit = TRUE)) |>
  mutate (uci = fit + (1.96 * se.fit),
          lci = fit - (1.96 * se.fit)) -> pred_lg_10_25cut

cbind.data.frame(data_10_12,
                 predict(glm(chd ~  lspline(sbp,knots = c(100,120,140,160)),
                             family = binomial(),
                             data = data_10_12),
                         se.fit = TRUE)) |>
  mutate (uci = fit + (1.96 * se.fit),
          lci = fit - (1.96 * se.fit)) -> pred_ls_10_25cli

cbind.data.frame(data_10_12,
                 predict(ls_10_25, se.fit = TRUE)) |>
  mutate (uci = fit + (1.96 * se.fit),
          lci = fit - (1.96 * se.fit)) -> pred_ls_10_25qrt
```

In the first four commands, we use `predict` to get the calculated logit for each row of the dataframe. The function `predict` requires a model object as its argument. Where we haven't saved a model object, we build the model object inside `predict` by specifying the call to `glm` as its model object argument. We also specify `se.fit=TRUE` so that `predict` returns the standard error as well. We use `cbind.data.frame` to bind together the original dataframe with the data returned by `predict`. Then, we use `mutate` to build two new columns to store the value of the upper and lower confidence interval of the predicted logit. We do this by adding and subtracting 1.96 times the standard error to /from the predicted logit. We store the dataframes thus prepared with suitable names so that we may use them in calls to `ggplot`.

```
ggplot(pred_ls_10_25qrt, aes(y = fit, x = sbp)) +
  geom_ribbon(aes(ymin =lci, ymax = uci),
              fill = "blue",
              alpha = 0.25) +
  geom_line() +
  labs(x = "Systolic blood pressure (mmHg)",
       y = "Logit",
       title = "Spline with knots at quartiles") -> plot_ls_10_25qrt


ggplot(pred_lg_10_25sbp, aes(y = fit, x = sbp)) +
  geom_ribbon(aes(ymin =lci, ymax = uci),
              fill = "blue",
              alpha = 0.25) +
  geom_line() +
  labs(x = "Systolic blood pressure (mmHg)",
       y = "Logit",
       title =  "Linear") -> plot_ls_10_25sbp


ggplot(pred_lg_10_25cut, aes(y = fit, x = sbp)) +
  geom_ribbon(aes(ymin =lci, ymax = uci),
              fill = "blue",
              alpha = 0.25) +
  geom_line()+
  ylim(-8,0) +
  labs(x = "Systolic blood pressure (mmHg)",
       y = "Logit",
       title = "Categorical by quarter")-> plot_ls_10_25cut


ggplot(pred_ls_10_25cli, aes(y = fit, x = sbp)) +
  geom_ribbon(aes(ymin =lci, ymax = uci),
              fill = "blue",
              alpha = 0.25) +
  geom_line()+
  ylim(-8,0) +
  labs(x = "Systolic blood pressure (mmHg)",
       y = "Logit",
       title = "Spline with chosen knots" ) -> plot_ls_10_25cli

library(gridExtra)
grid.arrange(plot_ls_10_25sbp, plot_ls_10_25cut,
             plot_ls_10_25cli, plot_ls_10_25qrt)
```

We call `ggplot` with the data frames we prepared. Each `ggplot` call asks for two geoms – `geom_ribbon` to create the confidence interval and `geom_line` to create the estimate. We provide `alpha=0.25` to `geom_ribbon` so that it is drawn with a translucent colour. We use `ylim` to ensure that all the four graphs have the same vertical extent. The horizontal range is the same because the same variable *sbp* is used for all the graphs. Finally, we use `grid.arrange` to make a composite graph from the individual graphs we prepared.

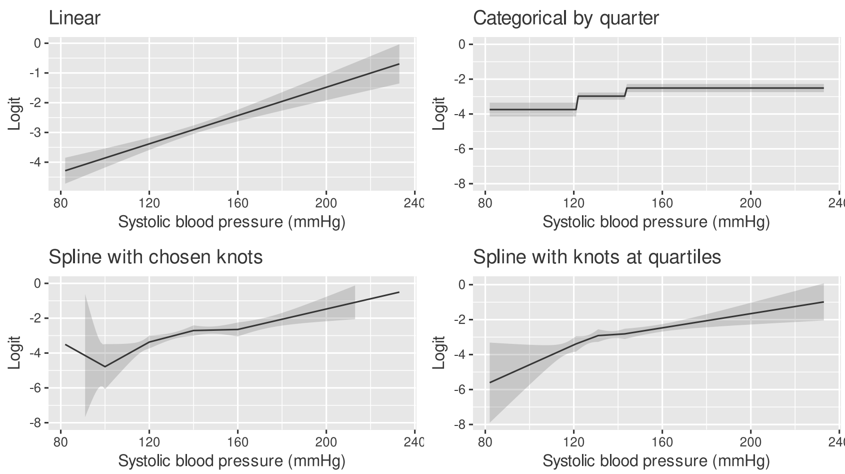


FIGURE 10.7
Replication of figure 10.10

Figure 10.11 (page 458)

Figure 10.11 is made in a similar way except that one model is laid over the other.

```
read_table("K11828 supplements/Datasets/Example 10.26.dat",
           col_names = c("egfr", "stroke"),
           col_types = cols(`stroke` = col_factor(levels = c("0", "1")))
           ) -> data_10_26
glm(stroke ~ egfr,
    data =  data_10_26,
    family = binomial()) -> lg_10_26

glm(stroke ~ lspline(egfr, knots = quantile(data_10_26$egfr, c(1/3, 2/3))),
    data =  data_10_26,
    family = binomial()) -> ls_10_26
glm(stroke ~ lspline(egfr,
                     knots = quantile(data_10_26$egfr,c(1/3, 2/3)),
                     marginal =TRUE),
    data =  data_10_26,
    family = binomial()) -> ls_10_26m

cbind.data.frame(data_10_26,
                 predict(ls_10_26, se.fit = TRUE)) |>
  mutate(ucils = fit + (1.96 * se.fit),
```

```
          lcils = fit - (1.96 *se.fit)) -> pred_10_26ls

cbind.data.frame(data_10_26,
                 predict(lg_10_26, se.fit = TRUE)) |>
  mutate(ucilg = fit + (1.96 * se.fit),
         lcilg = fit - (1.96 *se.fit)) -> pred_10_26lg

ggplot(pred_10_26ls) +
  geom_ribbon(aes(x=egfr, ymin = lcils, ymax = ucils),
              alpha = 0.25) +
  geom_line(aes(x=egfr, y = fit)) +
  geom_line(aes(x=egfr, y = fit),
            data =  pred_10_26lg,
            colour = "#004B73")+
  geom_line(aes(x=egfr, y = ucilg),
            data =  pred_10_26lg,
            colour = "#004B73",
            linetype = 3) +
  geom_line(aes(x=egfr, y = lcilg),
            data =  pred_10_26lg,
            colour = "#004B73",
            linetype = 3) +
  labs(x = "eGFR (ml/min/1.73m^2",
       y = "Logit")
```

We import data and build the model objects for example 10.26 as we did in the previous examples. We prepare two dataframes with the predicted values from the two models. While we could have combined the two, there will be conflict in the columns returned by `predict`. When we call `ggplot`, we first draw `geom_ribbon`. Otherwise, it will draw over the other lines and obscure them. When we call `geom_line` to draw the linear model, we provide it with a `data` argument to override that given in the `ggplot` call. We also specify line types and colours for the lines.

10.7 Model checking

Example 10.27 (page 460)

R provides helper functions to obtain the various residuals.

```
hatvalues(lg_10_5)
residuals(lg_10_5)
influence(lg_10_5)$dev.res
rstandard(lg_10_5)
```

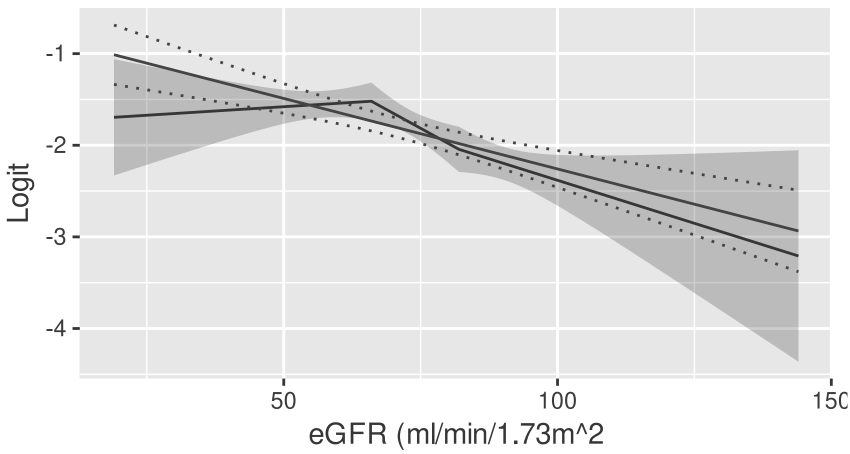


FIGURE 10.8
Replication of figure 10.11

The function `hatvalues` returns the leverage for each observation. The function `residuals` returns the raw residuals, while `rstandard` returns the standardised residuals. The deviance residual is available as the component `dev.res` returned by `influence`. The function `summary` when provided with the result of `influence`, will give a list of outliers according any of these measures.

```
       1        2        3        4        5        6        7        8
0.098315 0.120284 0.112579 0.098758 0.077008 0.077258 0.077785 0.071506
       9       10       11       12       13       14       15       16
0.066880 0.059292 0.054744 0.056877 0.059482 0.072335 0.075584 0.084760
      17       18       19       20
0.128586 0.156968 0.178435 0.272564
```

```
        1         2         3         4         5         6         7
-2.187774  1.737357  1.797810 -0.693693  0.986487 -0.240677 -0.177415
        8         9        10        11        12        13        14
 0.057026 -1.015599 -0.142785 -0.270138 -1.298515 -0.686425  1.654159
       15        16        17        18        19        20
-0.223551 -1.291742 -1.358424  0.813221  0.630932  0.778480
```

```
        1         2         3         4         5         6         7
-2.187774  1.737357  1.797810 -0.693693  0.986487 -0.240677 -0.177415
        8         9        10        11        12        13        14
 0.057026 -1.015599 -0.142785 -0.270138 -1.298515 -0.686425  1.654159
       15        16        17        18        19        20
-0.223551 -1.291742 -1.358424  0.813221  0.630932  0.778480
```

```
        1         2         3         4         5         6         7
-2.303961  1.852327  1.908442 -0.730713  1.026815 -0.250550 -0.184746
        8         9        10        11        12        13        14
```

```
 0.059181 -1.051364 -0.147216 -0.277850 -1.337097 -0.707799  1.717440
       15         16         17         18         19         20
-0.232511 -1.350232 -1.455201  0.885699  0.696084  0.912747
```

Figure 10.12 (page 462)

We may choose to plot any of these residuals. To prepare the graph in figure 10.12, we need the command

```
plot(rstandard(lg_10_5) ~ age,
     xlab = "Age (years)",
     ylab = "Standardised deviance residuals")
```

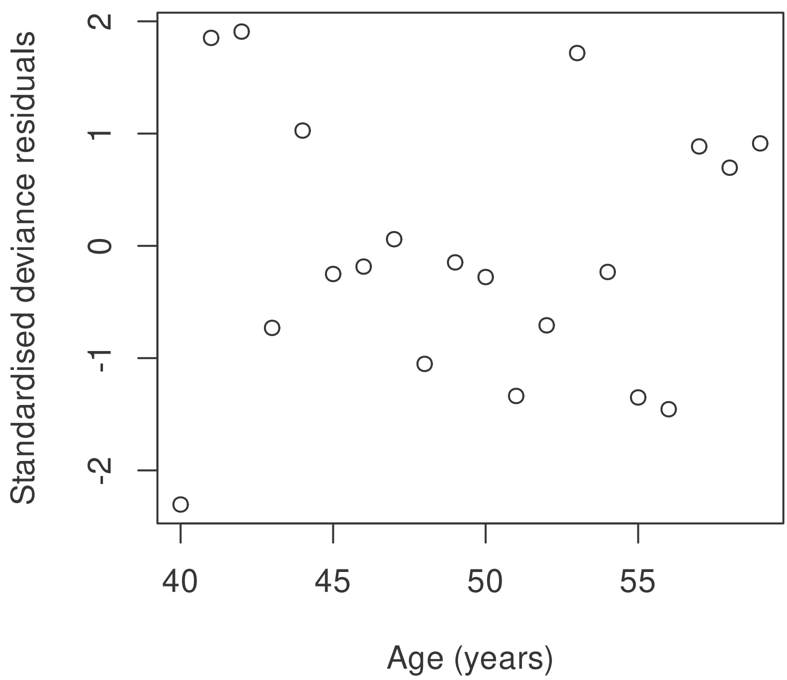


FIGURE 10.9
Replication of figure 10.12

Figure 10.13 (page 463)

Similarly, to plot the graphs in figure 10.13, we may use

```
qplot(s.class, rstandard(lg_10_6_ord),
     xlab = "Social class",
     ylab = "Standardised deviance residuals",
     main = "Linear model")
qplot(s.class, rstandard(lg_10_6_p2),
      xlab = "Social class",
```

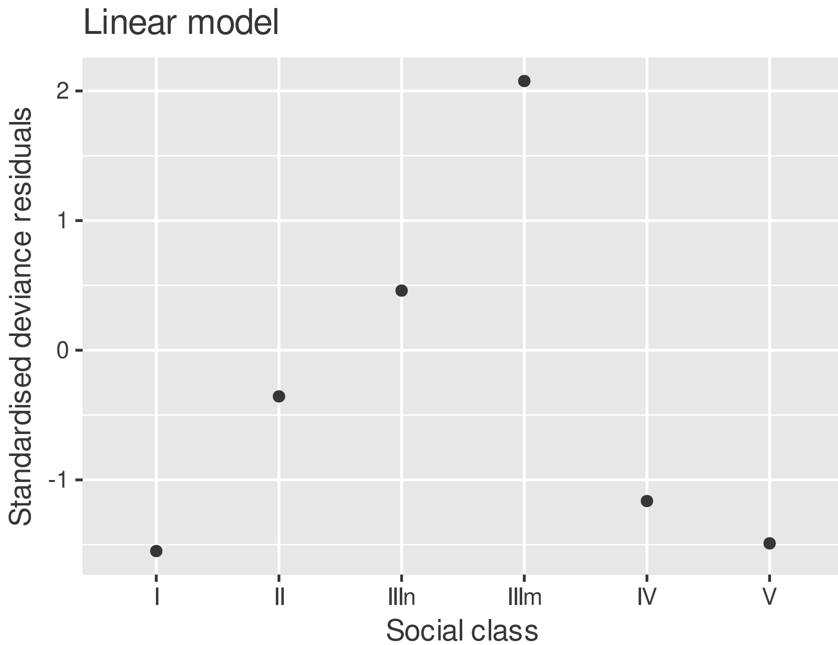


FIGURE 10.10
Replication of figure 10.13 (a)

```
      ylab = "Standardised deviance residuals",
     main = "Quadratic model")
```

Note that we have used `qplot` from `ggplot` package, which has a syntax similar to `plot` with some differences. The order of the variables are different in `qplot` and `plot`. Formulas are acceptable in `plot`, but not in `qplot`.

10.8 Case control studies

Example 10.32 (page 469)

As we don't have the data of table 6.11 in generic form, we will skip example 10.31. To rework the example 10.32, we need to import the data. We will use the package `survival` for this example.

```
read_table("K11828 supplements/Datasets/Table 10.42.DAT",
           col_names = c("id", "cc_status", "ddimer", "sbp"),
           col_types = cols(id  = col_integer(),
                            ddimer = col_factor(levels = c("0", "1")))
           ) -> tbl_10_42
library(survival)
```

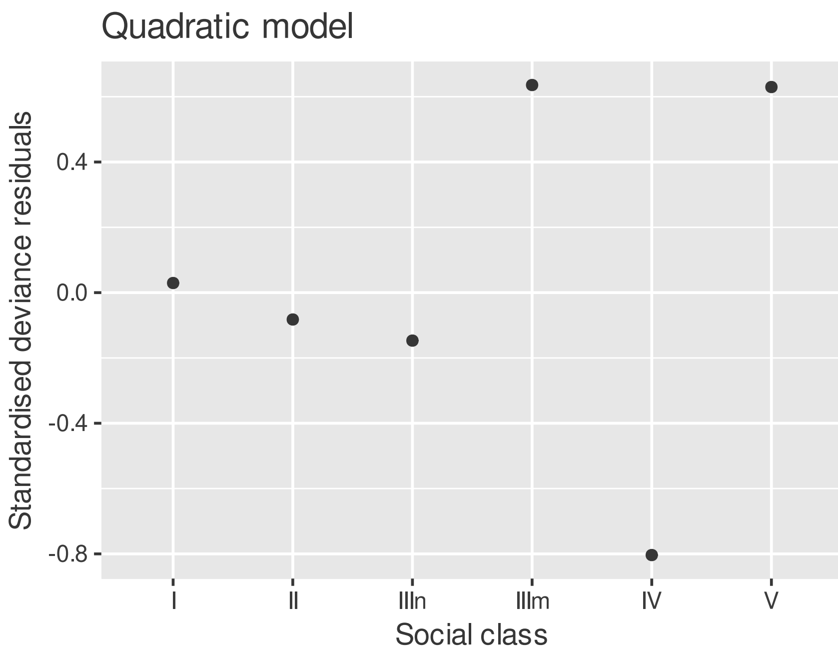


FIGURE 10.11
Replication of figure 10.13 (b)

```
clogit(cc_status ~ ddimer + strata(id),
       data =  tbl_10_42) -> cl_10_42_dd
clogit(cc_status ~ sbp + strata(id),
       data =  tbl_10_42) -> cl_10_42_bp
clogit(cc_status ~ ddimer + sbp + strata(id),
       data =  tbl_10_42) -> cl_10_42_db
```

Note that when we import data, the outcome variable *cc_status* is not defined as a factor, but left as numeric, the default. This is what `clogit` from `survival` which we use to perform conditional logistic regression analysis, expects. Apart from the name of the function, the formula that we provide for conditional logistic regression is different in that it has an additional term `strata()`, which accepts the name of variable which identifies the case control groups, in our case *id*.

Though we may use helper functions to extract the relevant information from the model objects, the default print method itself displays most details. We may use `anova` to compare the models.

```
cl_10_42_db
anova(cl_10_42_db)
```

```
Call:
clogit(cc_status ~ ddimer + sbp + strata(id), data = tbl_10_42)
```

```
         coef exp(coef) se(coef)   z    p
ddimer1 0.194     1.215    0.451 0.4 0.67
sbp     0.022     1.022    0.011 2.0 0.04

Likelihood ratio test=4.5  on 2 df, p=0.1
n= 135, number of events= 28


Analysis of Deviance Table
 Cox model: response is Surv(rep(1, 135L), cc_status)
Terms added sequentially (first to last)

       loglik Chisq Df Pr(>|Chi|)
NULL    -48.6
ddimer  -48.5  0.27  1      0.601
sbp     -46.4  4.25  1      0.039 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Here, we have used the model with both risk factors to call the default print method. R reports the likelihood ratio test, its df and its probability. The coefficients are reported for both explanatory variables along with its exponentiated value, standard error as well as Wald's test and its p value. We don't get a confidence interval, for which we may use `confint`. Another helper function of importance is `logLik`, which returns the log likelihood of the model. `anova` too reports log likelihood rather than -2 log likelihood.

10.9 Outcomes with several levels

Example 10.33 (page 472)

We will use `polr` from `MASS` to fit the proportional odds model. First, the data.

```
data.frame(hstry = factor(rep(c("yes", "no"), each = 4)),
           chd = factor(rep(c("mi", "ang2", "ang1", "none"),2),
                        levels = c("mi", "ang2", "ang1", "none")),
           num = c(104,17,45,830,192,30,122,3376)) -> tbl_10_44
library(MASS)
polr(chd ~ hstry,
     weights = num,
     data = tbl_10_44,
     method = "logistic") -> po_10_44
```

We build a dataframe to represent the data in table 10.44. For this, we use `rep` and `factor`. The order of the levels of the response factor *chd* needs to be specified as the default alphabetic arrangement does not reflect the actual order. For this, we provide the `levels` argument to `factor` while building *chd*.

We call `polr` in a way similar to previous examples. Thus, we have a formula that specifies the response and predictors and `data` argument to specify the data frame in which to look for the variables mentioned in the formula. As we have grouped data, we need to ask R to weight the fitting using *num*. Though `logistic` is the default `method`, we specify it for clarity. We may use `summary` to confirm that the model we built is the same as given in output 10.7.

```
summary(po_10_44)
```

```
Call:
polr(formula = chd ~ hstry, data = tbl_10_44, weights = num,
    method = "logistic")

Coefficients:
          Value Std. Error t value
hstryyes -0.684      0.102   -6.72

Intercepts:
          Value   Std. Error t value
mi|ang2    -2.884   0.068    -42.245
ang2|ang1  -2.725   0.065    -42.113
ang1|none  -2.286   0.057    -40.406

Residual Deviance: 4108.43
AIC: 4116.43
```

Multinomial regression can be done using `multinom` function of `nnet` package.

10.10 Longitudinal data

Example 10.35 (page 475)

We will use the package `gee` to rework the example 10.35.

```
read_table("K11828 supplements/Datasets/Example 9.21.dat",
           col_names = c("id","week","group","ascore","fscore","detect"),
           col_types = cols(id = col_integer(),
                            week = col_integer(),
                            group = col_factor(),
                            ascore = col_double(),
                            fscore = col_double(),
                            detect = col_integer()),
           na = ".")  |>
  filter(!is.na(detect)) |>
```

```
  arrange("id", "week") -> data_9_21
relevel(data_9_21$group, ref = "R") -> data_9_21$group
```

The process of importing is similar to our previous examples. An argument that we haven't seen earlier is `na`, which is used to specify the character in the data file that stands for `NA`, the period in our case. We remove those rows with `NA` in the *detect* column using `filter`. We make sure that the imported data is sorted by the id variable. We use `arrange` to achieve this. The first argument for `arrange` is the dataframe that needs to be sorted, which is supplied by the pipe. Rest of the arguments are names of the columns by which the dataframe should be sorted. Note that we have specified the response variable *detect* as numeric, rather than as a factor because that is what `geeglm`, the function that we use to model generalised estimating equations requires. Remember to install `geepack` using `install.packages` as discussed in chapter 1.

```
library(geepack)
geeglm(detect ~ group,
       id = id,
       data = data_9_21,
       family = binomial(),
       corstr = "exchangeable") |>
  summary()
```

We use the package `geepack` to model using generalised estimating equations. The function is named `geeglm`. It requires a formula specifying the model. In addition it requires an `id`, which specifies the column that carries the information to identify the clusters. The arguments `data` is the name of the dataframe in which the supplied variables should be looked for. The `family = binomial()` specifies that we want to use a binomial model. The argument `corstr` is used to specify the correlation structure we want. The `summary` function prints a great amount of information including the value of coefficients, their standard errors, the z value calculated for the standard errors and the probability for the z value.

```
Call:
geeglm(formula = detect ~ group, family = binomial(), data = data_9_21,
    id = id, corstr = "exchangeable")

 Coefficients:
            Estimate Std.err Wald Pr(>|W|)
(Intercept)    0.138   0.166 0.69     0.41
groupC        -0.048   0.239 0.04     0.84

Correlation structure = exchangeable
Estimated Scale Parameters:

            Estimate Std.err
(Intercept)    0.999 0.00593
  Link = identity
```

```
Estimated Correlation Parameters:
      Estimate Std.err
alpha    0.213  0.0464
Number of clusters:   107  Maximum cluster size: 5
```

We can use gee function from the package gee as well to fit generalised estimating equations. However, there is no anova method for the model object it returns.

10.11 Binomial regression

Example 10.36 (page 476)

To fit a binomial regression, we use `glm`, but with the link `log`. We use the variables from example 10.11.

```
glm(chd ~ chol + sbp,
    family = binomial(link = "log")) -> br_10_11
```

The only difference from the fitting of logistic regression is in the `link` argument to `binomial`. We can use `tidy` to get the odds ratio, relative risk and their confidence intervals.

```
dplyr::select(tidy(lg_10_11_full,exponentiate = TRUE,conf.int = TRUE)[-1, ],
              c(estimate,conf.low,conf.high)) |>
  rename(`Odds Ratio` = estimate,
         `OR ll` = conf.low,
         `OR ul` = conf.high) -> or_10_11

dplyr::select( tidy(br_10_11,
                    exponentiate = TRUE,
                    conf.int = TRUE)[-1, ],
                    c(estimate,conf.low,conf.high)) |>
  rename(`Relative Risk` = estimate,
         `RR ll` = conf.low,
         `RR ul` = conf.high) -> rr_10_11
```

We use `tidy`, asking it to exponentiate the coefficients and to provide confidence intervals. We exclude the row corresponding to the intercept by negative subsetting. We use `select` to select only those columns we want in the table. We specify `dplyr` before `select` to make sure that the conflict with a function of the same name in `MASS` that we loaded earlier doesn't affect us. We use `rename` to change the default column names returned by `tidy`. We store the dataframes thus made.

```
cbind.data.frame(predictor = c(paste("sbp",levels(sbp)),
                               paste("chol", levels(chol))),
                 rbind(c(1,NA,NA),
                       or_10_11[5:8,],
                       c(1,NA,NA),
                       or_10_11[1:4,]),
                 rbind(c(1,NA,NA),
                       rr_10_11[5:8,],
                       c(1,NA,NA),
                       rr_10_11[1:4,]))
```

We column bind a vector containing the labels for each row and the two data frames using `cbind.data.frame`. The vector containing the row labels is built by concatenating the `levels` of the explanatory variables we used for model building after adding a tag before them to indicate the variable name. The dataframes we made that contains the exponentiated coefficients and their confidence intervals does not have a row for the reference levels, but the vector containing the row labels has. So, we need to insert a row containing the value one for the reference levels before we can column bind them. We do this by breaking up the dataframes suitably by subsetting it and then using `rbind` to join them back together with the rows for the reference levels.

```
        predictor Odds Ratio OR ll OR ul Relative Risk RR ll RR ul
1        sbp <=118       1.00    NA    NA          1.00    NA    NA
2      sbp 119-127       1.84 1.021  3.43          1.81 1.022  3.31
3      sbp 128-136       2.39 1.369  4.34          2.29 1.343  4.10
4      sbp 137-148       2.80 1.622  5.07          2.68 1.585  4.76
5         sbp >148       3.83 2.260  6.83          3.56 2.150  6.23
6      chol <=5.41       1.00    NA    NA          1.00    NA    NA
7   chol 5.42-6.01       1.23 0.656  2.34          1.23 0.665  2.30
8   chol 6.02-6.56       2.28 1.316  4.10          2.20 1.301  3.90
9   chol 6.57-7.31       2.74 1.614  4.85          2.60 1.568  4.53
10      chol >7.31       3.65 2.188  6.41          3.41 2.092  5.88
```

Table 10.46 (page 477)

To prepare table 10.46, first we import the data.

```
read_table("K11828 supplements/Datasets/Example 9.16.DAT",
           col_names = c("fibrinogen", "age", "hpylori" ),
           col_types = cols(hpylori  =col_factor(levels =c("0","1")))
           ) -> data_9_16

mutate(data_9_16,
       age_hl = factor(ifelse(age >= 50, "old", "young"),
                       levels = c("young", "old")),
       fib_hl = factor(ifelse(fibrinogen >= 2.7, "high", "low"),
                       levels = c("low","high"))) -> data_9_16
```

We then `mutate` the data to prepare two new columns from the existing columns. We use `ifelse` which evaluates the logical statement given as its first argument and returns the second argument if the logical statement is true or the third argument if the logical statement is false. Thus, these new columns will contain the appropriate label to indicate in which of the two groups they fall according to the value of *age* and *fibrinogen*.

```
table(data_9_16$fib_hl,
      data_9_16$age_hl,
      data_9_16$hpylori,
      dnn = c("fibrinogen", "age group", "hpylori")) |>
  data.frame() |>
  pivot_wider(names_from = hpylori,
              names_prefix = "hpylori",
              values_from = Freq) |>
  mutate(prevalence = hpylori1 / (hpylori0 + hpylori1),
         odds = hpylori1 / hpylori0)
```

We then use `table` to cross tabulate the three variables. The `dnn` argument determines the name of the resulting columns. The last column containing the counts is named `Freq`. However, the result we get from `table` is an array, not very suitable for printing. So, we convert it into a dataframe. The resulting dataframe has different rows corresponding to the different values of *hpylori*. But, we want different columns. So, we use `pivot_wider` to make new columns corresponding to the different values of *hpylori* and carrying the corresponding `Freq`. We pipe the resulting data frame into `mutate` to calculate prevalence and odds. Note that we have prepared only the first four rows of table 10.46 corresponding to cross tabulation of age and fibrinogen groups. We need to repeat the steps by cross tabulating only fibrinogen groups and row bind the result with this data frame to get the complete table.

TABLE 10.3
Replication of table 10.46

Fibrinogen	Age group	No H. pylori	H. pylori	Prevalence	Odds
low	young	63	79	0.556	1.25
high	young	27	48	0.640	1.78
low	old	26	73	0.737	2.81
high	old	33	161	0.830	4.88

Table 10.48 (page 478)

To prepare table 10.48, we need to specify all four models.

```
list(unadjlog = glm(hpylori ~ fib_hl,
                    data = data_9_16,
                    family = binomial(link = "logit")),
     unadjbin = glm(hpylori ~ fib_hl,
                    data = data_9_16,
                    family = binomial(link = "log")),
```

```
     adjstlog = glm(hpylori ~ fib_hl + age_hl,
                    data = data_9_16,
                    family = binomial(link = "logit")),
     adjstbin = glm(hpylori ~ fib_hl + age_hl,
                    data = data_9_16,
                    family = binomial(link = "log"))) -> mdls_10_37

map_dfr(mdls_10_37, coef, .id = "model") |>
  replace_na(list(age_hlold = 0)) |>
  mutate(pfib_l = ifelse(str_detect(model, "bin"),
                         exp(`(Intercept)` + age_hlold),
                         (1 + exp(-1 *
                                    (`(Intercept)` +
                                       age_hlold ))) ^ -1),
         pfib_h = ifelse(str_detect(model, "bin"),
                         exp(`(Intercept)` + age_hlold + fib_hlhigh),
                         (1 + exp(-1 *
                                    (`(Intercept)` +
                                       age_hlold +
                                       fib_hlhigh))) ^ -1),
         pratio = ifelse(str_detect(model, "bin"),
                         exp(fib_hlhigh),
                         pfib_h / pfib_l)) -> tbl_10_48
tbl_10_48
```

In the first step we save the four model objects as a list using the command `list`. In the next step, we use `map_dfr` to build a data frame containing the coefficients of one model in one row. The `map_dfr` accepts a list, the list containing the model objects in our case. It passes on each element of the list to the function that is its second argument, `coef` in our case. The result returned by the function is converted to a data frame. If we supply the `.id` argument, `map_dfr` will add a column whose name is the value of `.id`. Its value will be the name of each element of the list we supplied as the first argument.

The data frame returned by `map_dfr` will contain `NA` for *age_hold* column for the unadjusted models. We change those `NA`s to 0 using `replace_na`. The function `replace_na` accepts in addition to the dataframe whose `NA` needs to be replaced, a list containing the variable value pair indicating the name of the column whose `NA` values need change and the value that should replace `NA`s. In our case, we change the `NA`s of *age_hlold* column to 0.

Finally, we use `mutate` to add three new columns to calculate the prevalences corresponding to high and low fibrinogen levels and prevalence ratio. Each specification of `mutate` is an `ifelse` which evaluates whether the name of the model contains “bin”. The function that does this comparison is `str_detect` from the library `stringr`, a part of `tidyverse`. It accepts the variable whose values should be checked and the string whose presence is checked for. Based on whether `str_detect` returns `TRUE` or `FALSE`, we use the calculations appropriate for the model to calculate the value of the new columns.

TABLE 10.4
Replication of table 10.48

Model fitted	Parameters			Prevalence		
	Intercept b_0	Fibrinogen b_1	Age b_2	Low fibrinogen	High fibrinogen	Prevalence ratio
unadjlog	0.535	0.713	0.00	0.631	0.777	1.23
unadjbin	−0.461	0.209	0.00	0.631	0.777	1.23
adjstlog	0.193	0.450	0.90	0.749	0.824	1.10
adjstbin	−0.580	0.124	0.27	0.734	0.831	1.13

Example 10.38 (page 478)

We fit binomial regression model to grouped data, similar to how we fit a logistic model. The difference is in the `link`. Here, we rework example 10.38.

```
factor(c("yes", "no")) -> smoke
matrix(c(31,1386,15,1883),
       nrow = 2,
       byrow = TRUE) -> deaths
glm(deaths ~ smoke,
    family = binomial(link ="log")) -> bi_10_38
```

We can use `predict` to obtain the log risk for each category of the predictor variable, along with the standard error.

```
predict(bi_10_38, se.fit=TRUE) -> prdl_10_38

exp(prdl_10_38$fit[1] + c(-1,1) * (prdl_10_38$se.fit[1] * 1.96))
exp(prdl_10_38$fit[2] + c(-1,1) * (prdl_10_38$se.fit[2] * 1.96))
```

The function `predict` returns a list with the elements `fit` and `se.fit`. We select the one element of `fit`, add to and subtract from it 1.96 times the corresponding element from `se.fit`. We repeat the same for the other element.

```
[1] 0.0154 0.0310
[1] 0.00477 0.01308
```

To get the confidence interval of the risk ratio, we can use `confint`.

```
exp(confint(bi_10_38))["smokeyes",]
exp(confint.default(bi_10_38))["smokeyes",]
```

We use `confint`, exponentiate it and subset the row corresponding to smokers. If we would prefer the confidence interval based on normal approximation as given in the textbook, we may use `confint.default` instead of `confint`.

```
 2.5 % 97.5 %
  1.53   5.25
 2.5 % 97.5 %
  1.50   5.11
```

Example 10.39 (page 480)

To rework example 10.39 to demonstrate calculation of adjusted risks, we need to fit the binomial regression model.

```
rep(c("no", "yes"), each = 2) -> smoke
rep(c("rent", "own"), 2) -> house
matrix(c(33,923,48,1722,52,898,29,678),
       byrow = TRUE,
       ncol = 2) -> chd

glm(chd ~ smoke + house,
    family = binomial(link=log)) -> br_10_39

data.frame(`Housing Tenure` = house,
           `Confounder` = smoke,
           `CHD Events` = chd[,1],
           n = marginSums(chd, 1),
           Risk = chd[,1]/  marginSums(chd, 1),
           `Log risk` = predict(br_10_39),
           Risk = predict(br_10_39, type = "response"))
```

The grouped data is typed in as matrix and binomial regression model is fit using `glm` as we did in our previous examples. To build table 10.51, we build a data frame. The function `marginSums` calculates the sum of a matrix or array along the specified margins. Here, we specify `marginSums` to calculate the totals of each row of *chd*. Thus, we get the total for each combination of housing type and smoking status. We divide the first column of *chd* obtained by subsetting with the total returned by *marginSums* to calculate the risk directly. We use `predict` to get the log risk. When we pass `type="response"`, `predict` returns the predicted values in the scale of the response, i.e. after exponentiation.

```
  Housing.Tenure Confounder CHD.Events    n   Risk Log.risk Risk.1
1           rent         no         33  956 0.0345    -3.35 0.0350
2            own         no         48 1770 0.0271    -3.62 0.0269
3           rent        yes         52  950 0.0547    -2.91 0.0543
4            own        yes         29  707 0.0410    -3.18 0.0417
```

We can calculate the adjusted log risks from the fitted model.

```
sum(coef(br_10_39) * c(1,0.5,1))
sum(coef(br_10_39) * c(1, tapply(marginSums(chd,1), smoke,FUN= sum)["yes"] /
                         marginSums(chd), 1))
```

We multiply the values returned by `coef` with the appropriate values and add them together. For the balanced weighting, we multiply the coefficient for smoking status with 0.5 and that for intercept and housing type with 1. For the observed margins calculations, we multiply the coefficient for smoking with the proportion of smokers, while intercept and coefficient for housing remains one. We don't type the proportion of smokers, but calculate it. We use `tapply` to get the numerator. The function `tapply` applies a function given as the value for the argument `FUN` to each group of values of its first argument, the groups being determined by the second argument `INDEX`. In our example, the first argument is the result of `marginSums(chd,1)`, the row-wise total of *chd*. The variable *smoke* determines the groups. Thus we get an array containing the totals of *chd* for each value of *smoke* viz "yes" and "no". We subset the value for "yes" of this array and divide it with the value returned by `marginSums(chd)`. When no margins are specified, `marginSums` sums across all margins and returns the total cases. Thus, we get the prevalence of smoking for calculating the observed margin adjusted log risk.

```
[1] -3.13
[1] -3.19
```

We may instead use `emmeans` to calculate adjusted values.

```
library(emmeans)
emmeans(br_10_39, c("house"))
emmeans(br_10_39,
        c("house"),
        weights =tapply(marginSums(chd,1),smoke,FUN= sum))
```

The function `emmeans` is passed the model object as its first argument. The second argument is a character vector which specifies the names of the variables for which we want the adjusted risks estimated. In our case, we want the adjusted risks for the various levels of *house*. The function `emmeans` also supports `weights`, which permits vectors, with weight specified for each of the level of variable for which adjustment is required. We use `tapply(marginSums(chd,1), smoke,FUN= sum)` as our weight to adjust according to the observed margins. This works because the way we have specified `tapply` returns the total number of smokers and non smokers. The advantage of `emmeans` is that confidence intervals are returned as well.

```
 house emmean    SE  df asymp.LCL asymp.UCL
 own    -3.40 0.114 Inf     -3.62     -3.18
 rent   -3.13 0.107 Inf     -3.34     -2.92

Results are averaged over the levels of: smoke
```

```
Results are given on the log (not the response) scale.
Confidence level used: 0.95
 house emmean    SE  df asymp.LCL asymp.UCL
 own    -3.45 0.112 Inf     -3.67     -3.23
 rent   -3.19 0.112 Inf     -3.41     -2.97

Results are averaged over the levels of: smoke
Results are given on the log (not the response) scale.
Confidence level used: 0.95
```

10.11.1 Risk differences

Example 10.40 (page 484)

To fit the binomial model for estimating risk differences, we use the data from the previous example.

```
glm(chd ~ house,
    family = binomial(link="identity")) -> br_10_40sv
glm(chd ~ smoke + house,
    family = binomial(link="identity")) -> br_10_40dv
```

The difference in model specification is the value that `link` takes. For the binomial model that assumes additive effects, we use `link="identity"`.

We may use `confint` to get the confidence interval for the coefficients or use `predict` which returns the standard errors.

```
confint.default(br_10_40sv)
predict(br_10_40sv, se.fit = TRUE)
```

```
             2.5 % 97.5 %
(Intercept) 0.0243 0.0379
houserent   0.0020 0.0250
$fit
     1      2      3      4
0.0446 0.0311 0.0446 0.0311

$se.fit
      1       2       3       4
0.00473 0.00349 0.00473 0.00349

$residual.scale
[1] 1
```

Example 10.42 (page 486)

We will use geeglm to model using generalised estimating equation Poisson regression.

```
library(geepack)
read_table("K11828 supplements/Datasets/Example 10.12.DAT",
           col_names = c("age", "chol", "bmi", "sbp",
                         "smoke", "active", "chd", "nmbr"),
           col_types = cols( smoke = col_factor(levels =c("1","2","3")),
                             active = col_factor(levels = c("1","2","3")),
                             chd = col_integer())) -> data_10_12

geeglm(chd ~ age + chol + bmi + sbp + smoke +active,
       data = data_10_12,
       id = nmbr,
       family = poisson(),
       std.err = "san.se") -> ge_10_42
summary(ge_10_42)$coef
```

When we import data, we specify the response variable as numeric, to satisfy the requirement of geeglm. We call geeglm with the arguments we discussed in an earlier example. The difference is in the family argument. Here, we specify poisson() with its default link. We have also specified std.err = "san.se" to say that we want the robust estimation. However, it is the default option used even if we omit the argument. The summary method prints a great deal of information. Here, we select only the coef component, a data frame, returned by summary for printing.

```
             Estimate Std.err    Wald Pr(>|W|)
(Intercept)  -9.6080 0.80531 142.342 0.00e+00
age           0.0162 0.01286   1.581 2.09e-01
chol          0.2793 0.04793  33.962 5.62e-09
bmi           0.0386 0.01866   4.291 3.83e-02
sbp           0.0187 0.00335  31.166 2.37e-08
smoke2        0.3042 0.23616   1.659 1.98e-01
smoke3        0.6854 0.20512  11.166 8.33e-04
active2      -0.1734 0.16723   1.075 3.00e-01
active3      -0.0925 0.21973   0.177 6.74e-01
```

Table 10.54 (page 486)

We can column bind the coef components of the logistic model we fitted earlier and the GEE model to prepare table 10.54.

```
bind_cols(as.data.frame(summary(lg_10_12)$coef)[-1,],
          summary(ge_10_42)$coef[-1,]) |>
  mutate(ORlci = exp(`Estimate...1` - (1.96 * `Std. Error`)),
         ORuci = exp(`Estimate...1` + (1.96 * `Std. Error`)),
         RRlci = exp(`Estimate...5` - (1.96 * `Std.err`)),
         RRuci = exp(`Estimate...5` + (1.96 * `Std.err`)),
         OR = paste(format(exp(`Estimate...1`),trim = FALSE,digits = 2),
```

```
                  "(",
                  format(ORlci, trim = FALSE, digits = 2),
                  ",",
                  format(ORuci, trim = FALSE, digits = 2),
                  ")" ),
        RR = paste(format(exp(`Estimate...5`),trim = FALSE,digits = 2),
                  "(",
                  format(RRlci,digits = 2),
                  ",",
                  format(RRuci, digits = 2),
                  ")"),
        OR_p = format(`Pr(>|z|)`, digits = 2),
        RR_p = format(`Pr(>|W|)`, digits = 2)) |>
  dplyr::select(OR,OR_p, RR, RR_p)
```

The `coef` component of the `summary` method for logistic model is not a dataframe, but a matrix. So, we coerce it to a dataframe using `as.data.frame`. We use `bind_cols` to column bind the two data frames. We calculate the upper and lower confidence intervals for odds ratio and risk ratio from the coefficients. Note that the name of the estimate columns change after column binding to ensure unique column names. We `paste` together the exponentiated coefficient and its lower and upper intervals into a string. The numbers passed to `paste` are prettied using `format`. Its argument `trim=FALSE` decides whether the number is right justified to a common width, padded with zeros on the left. The argument `digits` determines the number of significant figures. Finally, we `select` only those columns that we want to display.

TABLE 10.5
Replication of table 10.54

	Odds ratio		Relative Risk	
Variable	Estimate (95% CI)	p value	Estimate (95% CI)	p value
age	1.02 (0.99 , 1.04)	2.1e-01	1.02 (0.99 , 1.04)	2.1e-01
chol	1.36 (1.21 , 1.53)	2.7e-07	1.32 (1.20 , 1.45)	5.6e-09
bmi	1.04 (1.00 , 1.09)	5.1e-02	1.04 (1.00 , 1.08)	3.8e-02
sbp	1.02 (1.01 , 1.03)	9.8e-08	1.02 (1.01 , 1.03)	2.4e-08
smoke2	1.38 (0.84 , 2.26)	2.0e-01	1.36 (0.85 , 2.15)	2.0e-01
smoke3	2.07 (1.35 , 3.19)	8.7e-04	1.98 (1.33 , 2.97)	8.3e-04
active2	0.83 (0.58 , 1.18)	2.9e-01	0.84 (0.61 , 1.17)	3.0e-01
active3	0.90 (0.57 , 1.43)	6.7e-01	0.91 (0.59 , 1.40)	6.7e-01

We can obtain adjusted risks using `emmeans`.

```
emmeans(ge_10_42, c("smoke"), weights = "proportional")
```

```
 smoke emmean    SE   df lower.CL upper.CL
```

```
1      -3.66 0.189 4040    -4.03    -3.29
2      -3.36 0.156 4040    -3.66    -3.05
3      -2.98 0.102 4040    -3.18    -2.78

Results are averaged over the levels of: active
Covariance estimate used: vbeta
Results are given on the log (not the response) scale.
Confidence level used: 0.95
```

While the adjusted means are the same as in the textbook, the confidence interval is different from that in the textbook.

10.12 Propensity score

Example 10.45 (page 492)

We will skip examples 10.43 and 10.44. We will first import the relevant data to rework example 10_45.

```
read_table("K11828 supplements/Datasets/Example 10.45.dat",
           col_names = c("age", "simd", "smoke","alcohol", "cancer"),
           col_types = cols(smoke = col_logical(),
                            alcohol = col_logical(),
                            cancer= col_logical())) -> data_10_45
glm(alcohol ~ age + simd +smoke ,
    data = data_10_45,
    family = binomial()) -> lg_10_45
data_10_45$ps <- predict(lg_10_45, type = "response")
summary(data_10_45$ps)
```

Importing data and fitting the logistic model are similar to the previous examples. We use `predict` to obtain the fitted values for each record. By specifying `type="response"`, we ensure that logit is exponentiated. We attach the predicted values to the original dataframe as a new column. Finally, we use `summary` to confirm that the averages and extremes are similar to that given in the textbook.

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.226   0.330   0.389   0.398   0.457   0.731
```

We will use the library `MatchIt` to match records based on the propensity score. Remember to install it using `install.packages` as described in chapter 1. I couldn't find an R package that allows digit matching. So, we will use greedy nearest neighbour matching with calipers. Consequently, the results here will differ from the textbook.

```
library(MatchIt)
matchit(alcohol ~ age +simd + smoke,
        data = data_10_45,
        method = "nearest",
        caliper = 0.2) -> ps_10_45
match.data(ps_10_45) -> match_10_45_pair
```

The function we use for matching is `matchit`. It requires a formula, the left-hand side of which is the treatment variable and the right-hand side is the combination of variables that we want to use for calculating propensity score. It accepts a `data` argument denoting the dataframe in which to find the variables mentioned in the formula. We specify the matching method as `method="nearest"`. The function `matchit` allows many more ways of matching, which are probably better methods. The argument `caliper` is used to specify the value of the caliper if we want to use one. Note that `matchit` is very versatile and supports a great many ways of performing matching than we are discussing here. We store the object returned by `matchit`.

The object returned by `matchit` is examined using different helper functions. For example, `summary` will print a detailed analysis of balance before and after matching. Another useful function is `plot`, which will prepare graphical displays for analysing balance. Many different graphs are available. For example, `plot(ps_10_45, type = "density")` will show the density plot for each of the covariates before and after matching.

The function `match.data` when supplied an object returned by `matchit` will return a data frame containing only those rows that were matched. In addition to the columns present initially, additional columns containing the matching strata value, the distance measure calculated etc. are also included.

Figure 10.15 (page 496)

We will use the data frame returned by `match.data` to prepare the graph of figure 10.15.

```
ggplot(match_10_45_pair)+
  geom_density(aes(x = ps,
                   group = alcohol,
                   linetype = alcohol,
                   colour = alcohol)) +
  labs(x = "Propensity score",
       y = "Density") +
  scale_color_manual(labels = c("Heavy Drinkers", "Not"),
                     values = c("#111111","#004B73")) +
  scale_linetype_manual(labels = c("Heavy Drinkers", "Not"),
                        values = c(1,2))
```

We call `ggplot` with the dataframe returned by `match.data`. We use `geom_density` to plot the density of the `ps` column of the dataframe, containing the calculated propensity scores. We ask to plot different lines for each value of *alcohol* with different colours and line types.

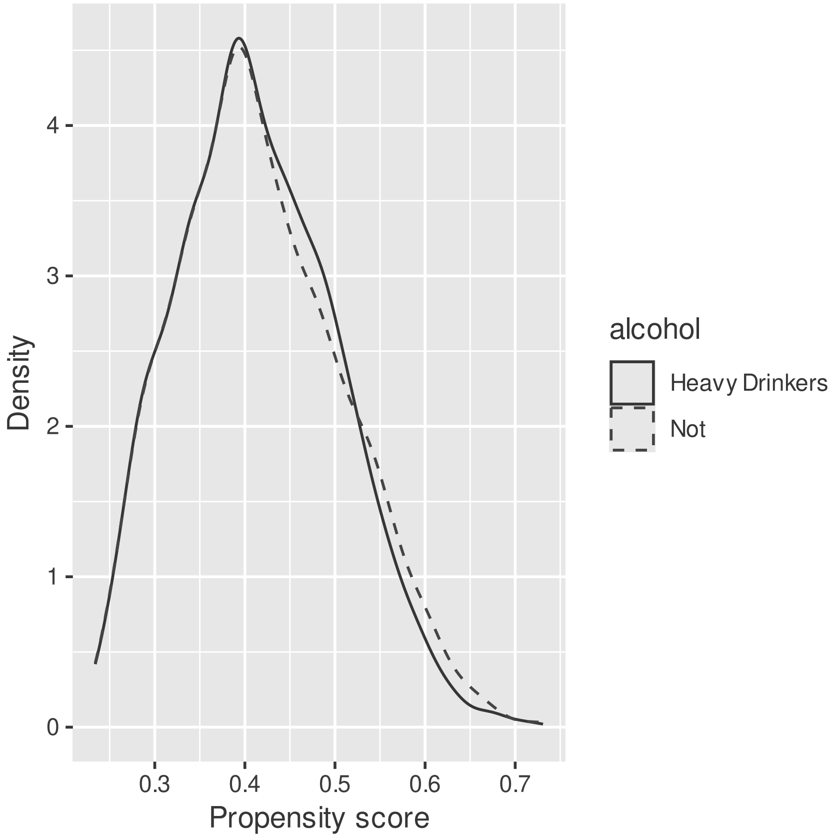


FIGURE 10.12
Replication of figure 10.15

Our graph is different from the textbook because of the different matching method we followed.

Figure 10.16 (page 497)

We will use the original dataset as well as the matched dataset to prepare the graphs of figure 10.16. We need `gridExtra` package to combine the two sets of graphs into one figure.

```
library(gridExtra)
grid.arrange(
ggplot(data_10_45)+
  geom_boxplot(aes(x = simd,
                   y = factor(alcohol, labels = c("Not","Heavy drinker")),
                   colour = alcohol),
               show.legend = FALSE)  +
  labs(x = "Scottish Index of Multiple Deprivation",
       y = NULL) +
  scale_colour_manual(values = c("#111111","#004B73")),
ggplot(match_10_45_pair)+
  geom_boxplot(aes(x = simd,
```

```
                  y = factor(alcohol, labels = c("Not","Heavy drinker")),
                  colour = alcohol),
              show.legend = FALSE) +
  labs(x = "Scottish Index of Multiple Deprivation",
       y = NULL) +
  scale_colour_manual(values = c("#111111","#004B73")))
```

We pass the appropriate data set to `ggplot`. The geom we use is `geom_boxplot`, which draws a boxplot for the `x` variable we specify. We use `y = factor(alcohol, labels = c("Not","Heavy drinker"))` to prepare different boxplots for different values of *alcohol*, with appropriate labels. The `colour` arguments gives different colours to each value of *alcohol* from the ones specified in `scale_colour_manual`. We put both `ggplot` calls inside `grid.arrange` to combine them into one figure.

FIGURE 10.13
Replication of figure 10.16

Table 10.60 (page 496)

To prepare table 10.60, we will use both the original and matched data sets.

```
group_by(data_10_45, alcohol) |>
  summarise(age = paste0(round(mean(age),2)," (", round(sd(age),2), ")"),
            simd = paste0(round(mean(simd),2)," (", round(sd(simd),2), ")"),
            smoke = as.character(round(mean(smoke),4) *100)) |>
```

```
  pivot_longer(cols = c(age, simd, smoke),
               names_to = "Covariates") -> sum_orig
group_by(match_10_45_pair, alcohol) |>
  summarise(age = paste0(round(mean(age),2)," (", round(sd(age),2), ")"),
            simd = paste0(round(mean(simd),2)," (", round(sd(simd),2), ")"),
            smoke = as.character(round(mean(smoke),4) * 100)) |>
  pivot_longer(cols = c(age, simd, smoke),
               names_to = "Covariates") -> sum_match

left_join(sum_orig,
          sum_match,
          by = c("alcohol", "Covariates")) |>
  rename(All = value.x, Matched = value.y) |>
  mutate(Drinking = ifelse(alcohol, "Heavy", "Not")) |>
  arrange(Covariates, Drinking) |>
  dplyr::select(Covariates, Drinking, All, Matched)
```

First, we prepare two dataframes, one each for the original dataset and for the matched dataset. We use `group_by` to group the data by *alcohol* and pipe the result to `summarise`. We use `summarise` to calculate summary measures for the three covariates. For *age* and *simd*, we calculate mean and standard deviation using `mean` and `sd`, round them using `round` and join them together with appropriate parentheses using `paste0`. For *smoke*, we calculate the proportion of smokers using `mean`. This is possible because of **implicit coercion** whereby the logical `TRUE` is converted to 1 and `FALSE` to 0. We need to convert this numerical value to character because we intend to join together all the summary measures into one column and for this to happen all values should be of the same mode. We use `as.character` to make this conversion. Each of the summary measure calculated is in a different column. We use `pivot_longer` to bring them all in one column. The first argument to `pivot_longer` is `cols`, the columns whose values we want to bring together into one column. Here we specify three columns corresponding to the covariates. The argument `names_to` determines the name of the new column.

We use `left_join` to join the two summary data frames. It matches the supplied dataframes using the values given in `by` and joins the appropriate rows.. We could have used `bind_cols` as well for joining the dataframes. An advantage of `left_join` is that the common columns `by` which the dataframes are joined are not repeated unlike `bind_cols`. One column each of the two summary data frames have the same name. When joined by `left_join`, their names are appended with `.x` and `.y` to distinguish between them. We `rename` them appropriately. The result is piped to `mutate`. We use `mutate` to create a new column *Drinking* from *alcohol*. We use `ifelse` to return the value “Heavy” if *alcohol* has the value `TRUE` and “Not” otherwise. Note that we don’t do an explicit logical comparison to check if the value of alcohol is `TRUE`. In other words, we don’t use `alcohol == TRUE`. We do so because the value of *alcohol* itself is a logical value. We sort the result using `arrange` and then use `select` to order and select the columns we need.

As expected, the values calculated for the unmatched data agree with the textbook while that of the matched data is slightly different from the textbook value because of the different matching procedure we use.

TABLE 10.6
Replication of table 10.60

Covariates	Drinking	All	Matched
age	Heavy	49 (5.72)	49.07 (5.71)
age	Not	50.29 (5.79)	49.05 (5.73)
simd	Heavy	24.65 (19.12)	24.18 (18.68)
simd	Not	20.08 (15.94)	23.36 (17.52)
smoke	Heavy	44.43	43.89
smoke	Not	33.85	43.03

Example 10.46 (page 496)

I am not aware of any ready-made function to cross tabulate the matched data and calculate the relative risk. So, we do the calculations by hand.

```
filter(match_10_45_pair, alcohol == TRUE) -> match_exp_10_45
filter(match_10_45_pair, alcohol == FALSE) -> match_nxp_10_45

left_join(match_exp_10_45,
          match_nxp_10_45,
          by = "subclass",
          suffix=c(".exp", ".nxp")) |>
  group_by(cancer.exp, cancer.nxp) |>
  summarise(counts = n()) |>
  arrange(desc(cancer.exp),
          desc(cancer.nxp)) -> sum_match_10_45

matrix(sum_match_10_45$counts,
       byrow = TRUE,
       nrow = 2,
       dimnames = list(`Heavy drinker dies from cancer`= c( "Yes","No"),
                       `Non-heavy drinker dies from cancer` =c( "Yes","No"))
       ) -> mm_10_45
```

First, we split the dataframe into two – one of heavy alcohol users and another of non-heavy alcohol users. We use `filter` to accomplish this. Next, we `left_join` them `by` the `subclass` column which carries the matching id. In effect, we are making a dataframe with one row for a pair of matched exposed and unexposed. We provide `suffix` to `left_join` to differentiate the columns of exposed and unexposed.

We pipe the joined dataframe to `group_by`. We ask `group_by` to group according to the values of *cancer.exp* and *cancer.nxp*, which carries the outcome variables of exposed and unexposed. The grouped data frame is fed to `summarise`, which summaries the number of records in each combination of values of the grouping variables. The summarised dataframe is then sorted in the descending order of *cancer.exp* and *cancer.nxp*. The sorting is needed to replicate the order in table 10.61. Finally, we construct a two-by-two matrix from the *count* column of the summarised dataframe and give appropriate column and row names.

We use the matrix to calculate RR, its variance, confidence interval and Wald statistic according to the formula given in the textbook.

```
rowSums(mm_10_45)[1] / colSums(mm_10_45)[1] -> rr_10_45
sqrt((mm_10_45[1,2] + mm_10_45[2,1]) /
       (rowSums(mm_10_45)[1] * colSums(mm_10_45)[1])) -> se_lRR_10_45
exp(log(rr_10_45) + c(-1.96, 1.96) *  se_lRR_10_45)
pchisq(log(rr_10_45)^2 / se_lRR_10_45^2, 1, lower.tail=FALSE)
```

We use `rowSums`, `colSums` and subsetting of the appropriate cells of the matrix we prepared to calculate the results. Note that as our matching algorithm was different from the textbook, our results differ from that in the textbook.

```
[1] 1.01 1.41
   Yes
0.0335
```

Next we divide the propensity score into its fifths and fit the data separately for these groups. We achieve this by specifying method for matching as `subclass`. In the textbook, the propensity score of the entire dataset is divided into its fifths. By default, `matchit` divides the propensity score of only those treated to derive the subgroups. We need to change the default behaviour of `matchit`.

```
matchit(alcohol ~ age + simd + smoke,
        data = data_10_45,
        method = "subclass",
        subclass =5,
        estimand = "ATE") -> ps_10_45_sc
match.data(ps_10_45_sc) -> match_10_45_sc
```

First, we use `matchit` to prepare a matchit object. The call is similar to our previous example, except that we specify the `method` as `subclass`, the number of `subclass` required as 5 and `estimand` as `"ATE"`, for average treatment effect. We specify `estimand` as `"ATE"` so that quantiles of propensity score to decide the subclass is calculated from the entire data set and not just the treated, which is the default. We use `match.data` to save the matched data as a dataframe.

Table 10.62 (page 497)

In order to build table 10.62, we write a custom function. We do this because we need to repeat essentially the same steps, save the stratum number, to obtain each row of information in the table. A function makes this a bit easier.

```
build_tbl <- function(x){mdl <- glm(cancer ~ alcohol ,
                                    family = binomial(link = "log"),
                                    data = match_10_45_sc,
                                    subset = subclass == x)
list(stratum = x,
     estimate = summary(mdl)$coefficients["alcoholTRUE", 1],
     std.err = summary(mdl)$coefficients["alcoholTRUE", 2],
```

```
    events = nrow(subset(match_10_45_sc,subclass == x & cancer == TRUE)),
    n = nrow(subset(match_10_45_sc, subclass == x)))}
```

We name the function *build_tbl*. We expect the function to accept one argument, the subclass' number, which will be accepted under the name x by the function. Inside the function, a binomial model is fitted using `glm` using the *match_10_45_sc* dataframe we prepared earlier. However, we restrict the data to only that match the specified subclass. This is achieved by specifying the `subset` argument to `glm` as `subclass == x`. Next, a list of five items, corresponding to the first five columns of table 10.62 is built. We use the `coefficients` component of the `summary` returned for the model object to select the estimate and standard error estimated for *alcohol*. We use `nrow` which returns the number of rows in a data frame, to calculate the total number of observations and events in each subclass. The data frame that is supplied to `nrow` is filtered using `subset`, which accepts the name of the dataframe to be filtered and the filtering condition. To obtain the total number of observations, we filter only by the subclass; to obtain the number of events, we filter by *cancer* as well.

We now build the table proper.

```
bind_rows(lapply(1:5, build_tbl)) |>
  mutate(n.logRR = n * estimate, n.se = (n * std.err)^2) -> tbl_10_62

summarise(tbl_10_62,
          across(c(events, n, n.logRR, n.se), sum)) ->  smry_tbl_10_62

bind_rows(tbl_10_62,smry_tbl_10_62)
```

We use `lapply`, supplying it with a vector with values 1 to 5 and the name of the function we built in the previous step. Thus, `lapply` will call the function five times, with the values 1 to 5, the value of the subclass and each time the function will return a list of five items for that particular subclass. As we call `lapply` from within `bind_rows`, the results are row bound into one data frame, which we save with the name *tbl_10_62*, after adding two more columns using `mutate`. These correspond to the last two columns of table 10.62. We build a row corresponding to the bottom row of table 10.62 using `summary`, which we row bind for display.

TABLE 10.7
Replication of table 10.62

Stratum (fifth)	log RR	SE	Events	n	$n \times$ log RR	$(\mathrm{n} \times SE)^2$
1	−0.0231	0.159	165	1009	−23.3	25720
2	0.2714	0.163	135	1009	273.8	26906
3	0.1999	0.176	115	1008	201.5	31580
4	0.2103	0.175	115	1009	212.1	31340
5	0.0506	0.163	132	1009	51.1	26983
			662	5044	715.3	142529

We now use the information in the summary we calculated to substitute on the formulas given in the textbook to calculate the weighted log relative risk & its standard error and the Wald statistics & its significance.

```
smry_tbl_10_62$n.logRR/ smry_tbl_10_62$n -> logRR_10_62
sqrt(smry_tbl_10_62$n.se)/ smry_tbl_10_62$n -> se_lRR_10_62
exp(logRR_10_62 + c(-1.96, 1.96) *  se_lRR_10_62)
pchisq(logRR_10_62^2 / se_lRR_10_62^2, 1, lower.tail=FALSE)
```

```
[1] 0.995 1.334
[1] 0.0581
```

Table 10.63 (page 498)

We will now build table 10.63.

```
glm(cancer ~ alcohol ,
    family = binomial(link = "log"),
    data = data_10_45) -> unadj
glm(cancer ~ alcohol + age + smoke + simd,
    family = binomial(link = "log"),
    data = data_10_45) -> adjst
glm(cancer ~ alcohol + ps ,
    family = binomial(link = "log"),
    data = data_10_45) -> psadjst

mutate(data_10_45,
       weight = ifelse(alcohol,1/ps,1/ (1-ps))) -> data_10_45
glm(cancer ~ alcohol,
    family = binomial(link = "log"),
    weights = weight,data = data_10_45) -> pswt
```

We build the model objects using `glm`. The dataframe that we supply for the unadjusted, covariate adjusted, and ps adjusted models is the original data with the calculated propensity score appended. For the inverse propensity weighted model, we use `mutate` to add a column with the weight calculated using `ifelse` and use it as the value of `weights` argument. This function evaluates a condition and returns the second argument if the condition is true and the third argument if the condition is false. We specify *alcohol* as our condition because its value is itself logical.

In the next step, we specify a custom function to build the rows for models that were built using `glm`.

```
build_tbl_noclstr <- function(mdl){
  tidy(mdl,conf.int = TRUE, exponentiate = TRUE) |>
  mutate(rr = paste0(round(estimate,2),
```

```
                     " (",
                     round(conf.low,2),
                     ", ",
                     round(conf.high,2),
                     ")"),
         `p value` = round(p.value,2)) |>
    filter(term == "alcoholTRUE") |>
    dplyr::select(rr,`p value`)}
```

Our custom function accepts a model object, passes it to `tidy` asking it to exponentiate and to return confidence intervals. This dataframe is passed to `mutate` to produce a column containing a string with the point estimate and confidence interval joined together and another with the p value rounded to three decimal places. We restrict the rows to *alcohol* and columns to the joined point estimate plus confidence interval and p value.

Now, we build the table proper.

```
bind_cols( Method = c("Unadjusted regression",
                     "Adjusted regression",
                     "Adjusted for PS",
                     "Weighted by inverse PS",
                     "Pair-matched by PS",
                     "Stratified by PS"),
           bind_rows(lapply(list(unadj, adjst, psadjst,pswt),
                            build_tbl_noclstr),
                     list(rr = paste0(round(rr_10_45, 2),
                                      " (",
                                      round(exp(log(rr_10_45) -
                                                (1.96 * se_lRR_10_45)),2),
                                      ", ",
                                      round(exp(log(rr_10_45) +
                                                (1.96 * se_lRR_10_45)),2),
                                      ")"),
                          `p value` = round(pchisq(log(rr_10_45)^2 /
                                                   se_lRR_10_45^2,
                                                 1,
                                                 lower.tail=FALSE),3)),
                     list(rr = paste0(round(exp(logRR_10_62), 2),
                                      " (",
                                      round(exp(logRR_10_62 -
                                                (1.96 * se_lRR_10_62)),2),
                                      ", ",
                                      round(exp(logRR_10_62 +
                                                (1.96 * se_lRR_10_62)),2),
                                      ")"),
                          `p value` = round(pchisq(logRR_10_62^2 /
                                                   se_lRR_10_62^2,
                                                 1,
```

```
                                                    lower.tail=FALSE),3))))|>
  rename(`Relative risk (95% confidence interval)` = rr)
```

We use `bind_rows` to row bind the result returned by calling `lapply` with a list of models and our custom function *build_tbl_noclstr*. The confidence interval and p value for the pair matched and stratified analysis calculated in earlier steps are joined inside a `list` each of which are supplied to `bind_cols` along with those returned by `bind_rows`. We join these rows with a column with name of the methods. Finally, the column containing the point estimate plus confidence interval is renamed.

TABLE 10.8
Replication of table 10.63

Method	Relative risk (95% confidence interval)	p value
Unadjusted regression	1.11 (0.96, 1.28)	0.167
Adjusted regression	1.16 (1, 1.33)	0.045
Adjusted for PS	1.15 (0.99, 1.33)	0.061
Weighted by inverse PS	1.15 (1.04, 1.27)	0.007
Pair-matched by PS	1.19 (1.01, 1.41)	0.034
Stratified by PS	1.15 (1, 1.33)	0.058

Note that while the result that we get when we use the original data frame is similar to that given in the textbook, the results are different when we use propensity score matched data. This is expected as different matching algorithms are expected to give different results. Also, as mentioned in the textbook, there is no consensus as to which ps matching algorithm or estimate of causal effect is better. Hence, I have not taken the effort to find out if the methods used in the textbook can be replicated exactly in R. Clearly, there is more to propensity score matching using R than is mentioned here. One package I would suggest you to check out for propensity score analysis is `PSweight`.

10.13 Recap

10.13.1 Concepts

- implicit coercion

10.13.2 Commands introduced in this chapter

- stats::glm
- stats::binomial
- stats::vcov
- ggplot2::geom_pointrange
- dplyr::rename
- ggplot2::geom_function
- ggplot2::xlim
- Epi::float
- readr::col_skip
- stats::deviance
- stats::df.residual
- stats::poly
- ggplot2::geom_ribbon
- stats::hatvalues
- stats::inflence
- ggplot2::qplot
- survival::clogit
- survival::strata
- MASS::polr
- readr::col_integer
- tidyr::replace_na
- base::tapply
- stats::confint.default
- stats::poisson
- base::format
- MatchIt::matchit
- MatchIt::match.data
- ggplot2::geom_density
- ggplot2::geom_boxplot
- base::paste0
- tidyr::pivot_longer
- dplyr::desc
- base::colSums
- base::lapply

11

Modelling follow-up data

In this chapter we will use the package `survival` to model follow up data. Most functions require formulas, dataframes or vectors. In addition to the `tidyverse` packages `readr`, `dplyr`, `stringr`, `tibble` and `ggplot2`, we will also use the add on packages `broom`, `SurvRegCensCov` and `Greg`. We start with example 11.1.

11.1 Estimating the hazard function

Example 11.1 (page 509)

```
library(tidyverse)
library(survival)
library(broom)
c(10,12,13,15,16,20,20,24,24,26,26,27,39,42,
  45,45,48,52,58,60,61,62,73,75,77,104,120) -> time_11_1
rep(TRUE, 27) -> event_11_1
survfit(Surv(time_11_1, event_11_1) ~ 1) -> km_11_1
```

We save the survival times with a suitable name. As all subjects experience death, the variable *events_11.1* has the value `TRUE` repeated 27 times. We fit a Kaplan Meir curve to the data using `survfit`, which accepts a formula with a `Surv` object as its left-hand side. The `Surv` object requires a variable with the survival time and a variable with information on death / alive to be supplied to it. As we are fitting only one KM curve, the right-hand side of the formula is just `1`. We may use `summary` to print a tabular data similar to the one in table 11.1. However, it won't print the column corresponding to hazard. Hence, we will use the variables returned by `survfit` to prepare table 11.1.

```
lead(km_11_1$time) - km_11_1$time -> km_11_1$interval
km_11_1$n.event /(km_11_1$interval  * km_11_1$n.risk) -> km_11_1$hazard
bind_cols(`Time (t)` = km_11_1$time,
          `Survivors (n)` = km_11_1$n.risk,
          `Deaths (e)` = km_11_1$n.event,
          `Interval (u)` = km_11_1$interval,
          `Survival (s)` = km_11_1$surv,
          `Hazard (h)` = km_11_1$hazard) -> tbl_11_1
bind_rows(c(`Time (t)` = 0,
```

DOI: 10.1201/9781003589563-11

```
           `Survivors (n)` = 27,
           `Deaths (e)` = 0,
           `Interval (u)` = 10,
           `Survival (s)` = 1,
           `Hazard (h)` = 0), tbl_11_1) -> tbl_11_1
tbl_11_1
```

The `survfit` object returns many variables in a list. We use the variable `time` to calculate the interval. The *interval* is calculated as the difference between the value of the next `time` and current `time`. The function `lead` supplies us the next value in the provided variable. We use this information to calculate hazard according to the formula given in the textbook. Finally we column bind the required variables to print our table. We add a row to indicate the state at the start of the time period. Note that though `survift` does not provide hazard, it does provide a variable with calculated cumulative hazards.

Time (t)	Survivors (n)	Deaths (e)	Interval (u)	Survival (s)	Hazard (h)
0	27	0	10	1.000000	0.0000000
10	27	1	2	0.962963	0.0185185
12	26	1	1	0.925926	0.0384615
13	25	1	2	0.888889	0.0200000
15	24	1	1	0.851852	0.0416667
16	23	1	4	0.814815	0.0108696
20	22	2	4	0.740741	0.0227273
24	20	2	2	0.666667	0.0500000
26	18	2	1	0.592593	0.1111111
27	16	1	12	0.555556	0.0052083
39	15	1	3	0.518519	0.0222222
42	14	1	3	0.481481	0.0238095
45	13	2	3	0.407407	0.0512821
48	11	1	4	0.370370	0.0227273
52	10	1	6	0.333333	0.0166667
58	9	1	2	0.296296	0.0555556
60	8	1	1	0.259259	0.1250000
61	7	1	1	0.222222	0.1428571
62	6	1	11	0.185185	0.0151515
73	5	1	2	0.148148	0.1000000
75	4	1	2	0.111111	0.1250000
77	3	1	27	0.074074	0.0123457
104	2	1	16	0.037037	0.0312500
120	1	1	NA	0.000000	NA

Figure 11.2 (page 510)

To get a survival plot, all we need to do is pass the `survfit` object to `plot`.

```
plot(km_11_1,
     conf.int= FALSE,
     xlab = "Time (weeks)",
     ylab = "Probability of survival")
```

We use `conf.int= FALSE` to suppress the lines that mark the confidence interval of the estimated survival function.

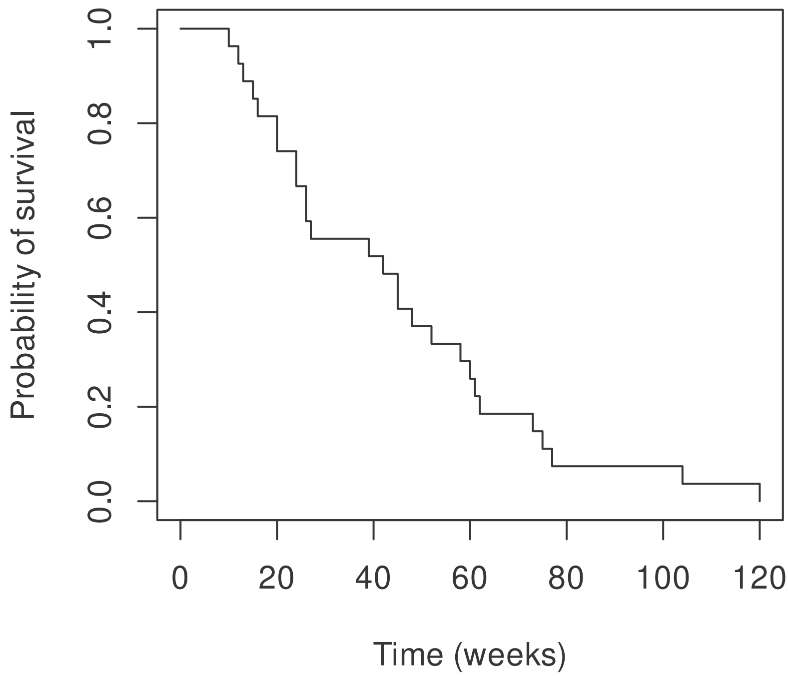


FIGURE 11.1
Replication of figure 11.2

Figure 11.3 (page 510)

To plot the hazard function, we use the hazard variable that we calculated.

```
plot( tbl_11_1$`Time (t)`,
      tbl_11_1$`Hazard (h)`,
      type = "s",
      xlab = "Time (weeks)",
      ylab = "Hazard")
```

We use `plot`, provide it with the x and y values, this time from *tbl_11_1* we prepared and specify `type="s"` to make it plot using steps.

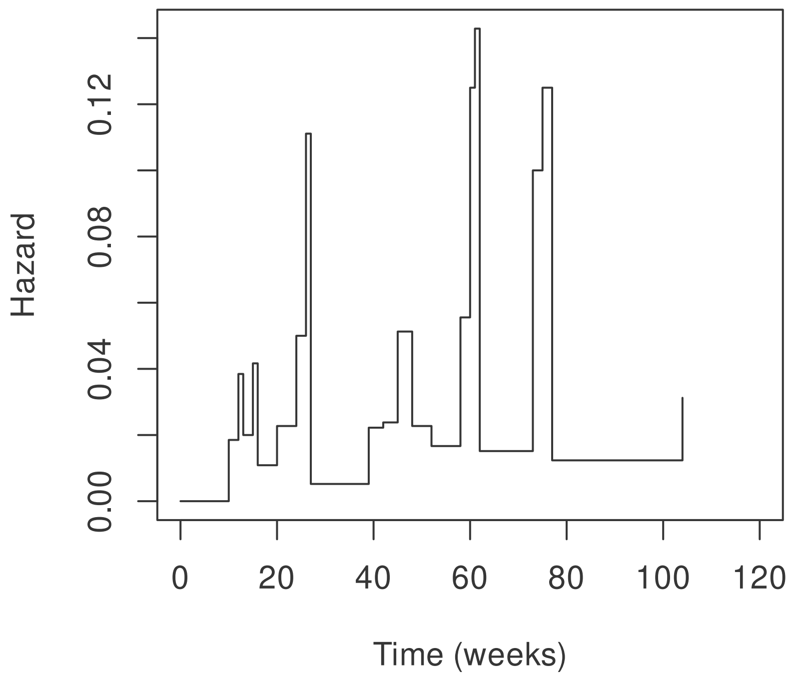


FIGURE 11.2
Replication of figure 11.3

11.1.1 Person-time estimation

Example 11.2 (page 511)

The function to calculate person times is `pyears`, which we use to rework example 11.2.

```
pyears(time_11_1 ~ 1 ,
       subset = time_11_1 >= 10 & time_11_1 < 20 ,
       scale =  1)
sum(time_11_1 >= 20)
```

The functions requires a formula. The left-hand side could be a `Surv` object or just the time variable. We use `subset` to specify the restrictions that we want to impose. The argument `scale = 1` is used to change the default behavior of dividing the time period by 365.25 assuming that the periods are in days. We obtain the number of people surviving up to and beyond 20 weeks by summing the elements of *time_11_1* which satisfies the condition.

```
Call:
pyears(formula = time_11_1 ~ 1, subset = time_11_1 >= 10 & time_11_1 <
    20, scale = 1)

Total number of person-years tabulated: 66
Total number of person-years off table: 0
Observations in the data set: 5
[1] 22
```

The result returned by the function is different from that in the text. The reason is that `pyears` doesn't subtract 10 from the periods for each of the events as done in the textbook.

11.2 Probability models

Example 11.5 (page 519)

Of the probability models, we will first fit the Weibull model of example 11.5. We will use `survreg` from `survival`.

```
survreg(Surv(time_11_1, event_11_1) ~ 1,
        dist = "weibull") -> wb_11_5
1 / wb_11_5$scale -> gamma_11_5
exp(-wb_11_5$coefficients["(Intercept)"]/ wb_11_5$scale) -> lambda_11_5
gamma_11_5
lambda_11_5
```

The first argument to `survreg` is a formula with a `Surv` object as its left-hand side. Our formula is the same as we used for Kaplan Meir estimation. The second argument specifies the distribution that we assume for the parametric regression. We may use `summary` to print the details of the survival regression object. Among the values returned by the function are the intercept and scale with the same interpretation that is required for SAS output as given in our textbook. We convert the two values to gamma and lambda using the formula given in the textbook.

```
[1] 1.6743
(Intercept)
  0.0014412
```

Figure 11.10 (page 520)

We now use the calculated values to plot the graph in figure 11.10. We use `ggplot` for plotting.

```
ggplot(tbl_11_1) +
  geom_step(aes(x= `Time (t)`, y = `Survival (s)`)) +
  geom_function(fun = function(x) exp(-lambda_11_5 * x ^ gamma_11_5),
                linetype = 2,
                colour =  "blue")
```

We supply `ggplot` with *tbl_11_1* and use `geom_step` to draw the observed survival curve. We use `geom_function` to draw the Weibull survival function. `geom_function` requires a function as its argument using which it will calculate the y value for each x value. Here, we supply an anonymous function which calculates the Weibull survival function for each of the time period according to the formula given in the textbook using the values returned by `survreg`. We use a different colour and linetype for the Weibull survival curve by specifying our choice of values for `linetype` and `colour`.

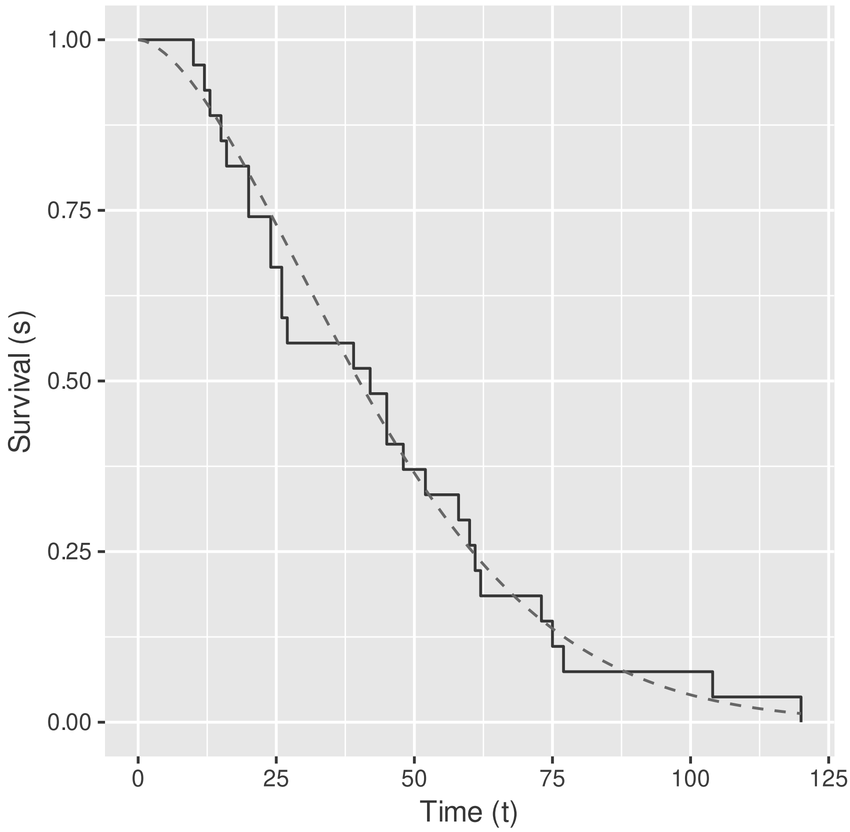


FIGURE 11.3
Replication of figure 11.10

Example 11.4 (page 516)

Fitting the exponential model is similar to the previous example.

```
survreg(Surv(time_11_1, event_11_1) ~ 1,
        dist = "exponential") -> xp_11_4
exp(- (coef(xp_11_4))) -> lambda_11_4
lambda_11_4
```

To fit an exponential model, `survreg` fits a Weibull distribution with the scale fixed at 1. Thus the coefficient returned needs the same transformation as described under Weibull to convert it to lambda of exponential model. We may substitute lambda in any of the formulas given in the textbook. For example, the median survival time calculated is `log(2)/lambda_11_4`.

```
(Intercept)
   0.022613
```

11.3 The Cox proportional hazards model

Example 11.7 (page 526)

To rework the example 11.7, we need the data.

```
c(12,15,16,20,24,26,27,39,42,45,45,58,60,61,
  62,73,77,104,120,10,13,20,24,26,48,52,75) -> time_11_7
factor(c(rep("low", 19), rep("high",8)),
       levels = c("low", "high")) -> cell_11_7
rep(TRUE,27) -> events_11_7
coxph(Surv(time_11_7,events_11_7) ~ cell_11_7,
      ties = "breslow") -> cph_11_7
summary(cph_11_7)
```

The data for the vector for survival period is rearranged so that all values that correspond to one type of cellularity is contiguous. The data on cellularity is stored as a factor with two levels. We specify the argument `levels` for `factor` to set the baseline level. If we don't specify `levels`, the levels are ordered alphabetically and *high* would be treated as the base level.

The function to build Cox's proportional hazards model is `coxph`. It requires a formula, the left-hand side of which is a `Surv` object. The right-hand side are the explanatory variables; we have only *cell_11_7*. The argument `ties` decides how the command will manage ties. Here, we use `breslow` so that we get answers similar to the textbook though the default method used by `coxph` is considered more accurate.

```
Call:
coxph(formula = Surv(time_11_7, events_11_7) ~ cell_11_7, ties = "breslow")

  n= 27, number of events= 27

               coef exp(coef) se(coef)    z Pr(>|z|)
cell_11_7high 0.558     1.747    0.437 1.28      0.2

              exp(coef) exp(-coef) lower .95 upper .95
cell_11_7high      1.75      0.572     0.742      4.11

Concordance= 0.568  (se = 0.052 )
Likelihood ratio test= 1.52  on 1 df,   p=0.2
Wald test            = 1.63  on 1 df,   p=0.2
Score (logrank) test = 1.67  on 1 df,   p=0.2
```

The summary method prints the calculated coefficient, its exponentiation, standard error and confidence interval. The likelihood ratio test, Wald test and Score test statistics and their p values are also printed. We may use `anova` to print an anova table.

Example 11.8 (page 527)

To rework the example 11.8, we import the data.

```
read_table("K11828 supplements/Datasets/Example 11.8.DAT",
           col_names = c("cholfifths", "sbpfifths", "chd", "survive"),
           col_types = cols(cholfifths = col_factor(levels = c("1","2","3",
                                                               "4", "5")),
                            sbpfifths = col_factor(levels = c("1","2","3",
                                                              "4", "5")))
           ) -> data_11_8

coxph(Surv(survive, chd) ~ sbpfifths + cholfifths ,
      data = data_11_8) -> cph_11_8sc
coxph(Surv(survive, chd) ~  cholfifths + sbpfifths ,
      data = data_11_8) -> cph_11_8cs
```

Note that when we import the data, the response variable is left as the default numeric type rather than converting it into a factor to satisfy the requirement of `coxph`. Fitting a Cox proportional model is similar to our previous example except that the right-hand side contains two explanatory variables. We fit two models which differ only in the sequence in which the explanatory variables are introduced.

Table 11.2 (page 528)

We use `anova` to obtain the data in table 11.2.

```
anova(cph_11_8sc) -> anova_11_8sc
anova(cph_11_8cs) -> anova_11_8cs

cbind(model = c("SBP", "Cholesterol",
                "SBP + Cholesterol",
                "Cholesterol + SBP"),
      select(rbind(anova_11_8sc[2,],
                   anova_11_8cs[2,],
                   anova_11_8sc[3,],
                   anova_11_8cs[3,],
                   make.row.names = FALSE),
             `Chisq`, `Df`, `Pr(>|Chi|)`))
```

We subset the relevant rows from the two anova tables and row bind them, select the relevant columns and add a column to indicate the model to prepare the table.

TABLE 11.1
Replication of table 11.2

	Test details		
model	Δ	df	p value
SBP	39.017	4	6.9104e-08
Cholesterol	45.027	4	3.9253e-09
SBP + Cholesterol	37.492	4	1.4264e-07
Cholesterol + SBP	31.482	4	2.4410e-06

Output 11.2 (page 529)

The `summary` method will display the information in output 11.2. Here, we use `tidy` to print a cleaner table

```
library(broom)
tidy(cph_11_8sc, conf.int = TRUE)
```

```
# A tibble: 8 x 7
  term        estimate std.error statistic    p.value conf.low conf.high
  <chr>          <dbl>     <dbl>     <dbl>      <dbl>    <dbl>     <dbl>
1 sbpfifths2     0.602     0.302     2.00  0.0460       0.0108      1.19
2 sbpfifths3     0.850     0.288     2.96  0.00311      0.287       1.41
3 sbpfifths4     1.01      0.283     3.56  0.000375     0.453       1.56
4 sbpfifths5     1.33      0.275     4.83  0.00000136   0.789       1.87
5 cholfifths2    0.203     0.318     0.637 0.524       -0.421       0.826
6 cholfifths3    0.804     0.283     2.84  0.00448      0.250       1.36
7 cholfifths4    0.976     0.274     3.56  0.000369     0.439       1.51
8 cholfifths5    1.26      0.267     4.70  0.00000258   0.733       1.78
```

Note that the statistic calculated is z statistic rather than the Chi square statistics.

Table 11.3 (page 529)

To prepare table 11.3, we use `tidy` again.

```
tidy(cph_11_8sc,  conf.int = TRUE, exponentiate = TRUE) |>
  mutate(value = paste0(round(estimate,2) ,
                        "(",
                        round(conf.low,2),
                        ", ",
                        round(conf.high,2),
                        ")"),
         fifth = str_sub(term, -1,-1),
         variable = ifelse(str_starts(term, "sbp"),
                           "Systolic blood pressure",
                           "Serum total cholesterol")) |>
  select(variable, fifth, value ) |>
  pivot_wider(names_from = variable, values_from = value)
```

We ask `tidy` to exponentiate the coefficients and its confidence interval by specifying `exponentiate=TRUE`. We `mutate` the result returned by `tidy` to create a string consisting of the exponentiated coefficients and their confidence interval appropriately rounded. We create two new columns, one to denote the fifth and one to denote the explanatory variable. To create *fifth*, we use `str_sub` to extract the last character from the `term` column. The first `-1` indicates that the substring begins from the first character from the end of `term` and the second `-1` indicates that the portion till the first character from the end of `term` should be returned. As the start and end are the same, we get just the last character from the end of `term`. The *variable* column created will have the name appropriate for the explanatory variable based on whether `term` starts with "sbp" as determined by `str_starts`. We `select` the columns we created and then use `pivot_wider` to separate the `values_from` *value* to different columns, which will be named according to their `names_from` *variable*.

TABLE 11.2
Replication of table 11.3

Fifth	Systolic blood pressure	Serum total cholesterol
2	1.83(1.01, 3.3)	1.22(0.66, 2.28)
3	2.34(1.33, 4.11)	2.23(1.28, 3.89)
4	2.74(1.57, 4.78)	2.65(1.55, 4.54)
5	3.77(2.2, 6.47)	3.51(2.08, 5.93)

Figure 11.12 (page 530)

We will now reproduce the graph in figure 11.12.

```
plot(survfit(cph_11_8cs,
             newdata = data.frame(cholfifths = factor(1:5),
                                  sbpfifths = factor(3))),
     ylim = c(.9, 1),
     col = c(1,2,3,4,5),
     xlab = " Time years",
     ylab = "Estimated probability of survival") -> plt_cph
text(plt_cph, c("1", "2", "3", "4", "5"),  adj= - 0.25)
```

First, we use `survfit` to prepare survival curves. However, unlike our previous examples, we provide it with the model object returned by `coxph` and with a new dataframe. The `newdata` we provide contains the explanatory variables used for building the Cox model. The column *cholfifths* has one instance each of the unique value of *cholfifths* in the data frame used for building the Cox model. As *cholfifths* was defined as a factor in the original dataframe, here also we use `factor`. As we want the survival probabilities to be adjusted for systolic blood pressure at the level of the middle fifth of the systolic blood pressure, the value of *sbpfifths* is set as `factor(3)` for the `newdata`. Thus `survfit` will give us five survival curves, one for each fifth of cholesterol adjusted at the middle fifth of systolic blood pressure.

We use `plot` to draw the curves returned by `survfit`. In addition to the `survfit` curves, we provide `ylim` to restrict the range of y-axis to between 0.9 and 1 or else, the lines will be squashed to the upper one tenth of the plot. We also provide `col` with five different integers so that each line gets a different colour. We don't print the graph immediately, but save it as we want to add labels to the lines. The function `text` adds the provided labels to the end of each line. The argument `adj` is provided to move the labels a bit away from the end of lines.

Example 11.9 (page 530)

We use the data from the previous example to rework example 11.9.

```
coxph(Surv(survive, chd) ~ as.numeric(sbpfifths),
            data = data_11_8)
```

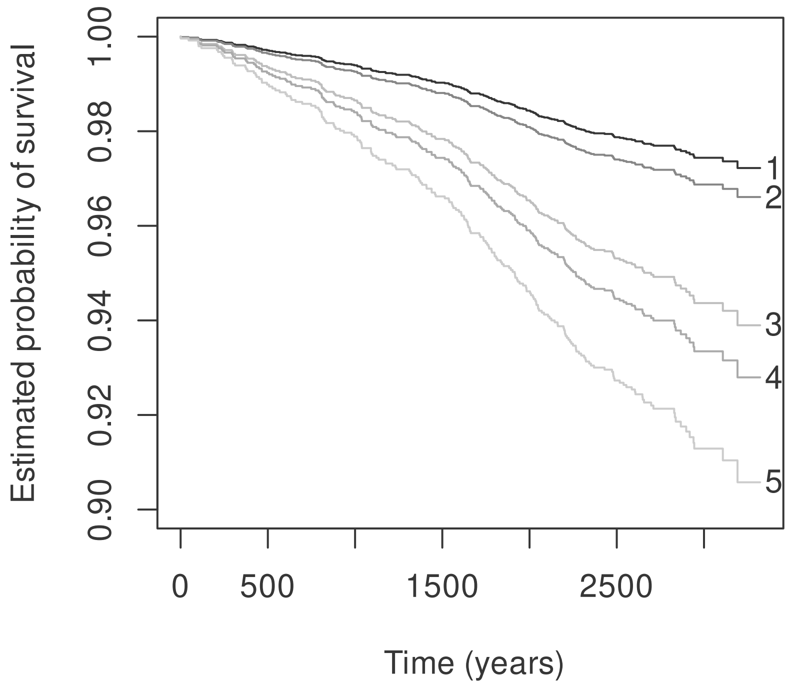


FIGURE 11.4
Replication of figure 11.12

Here, we have converted *sbpfifths*, which was originally defined as a factor, to a number using `as.number`. Otherwise, the procedure is similar to the previous example. We may use `anova` to compare the categorical and linear models. As we haven't saved the relevant models, we will specify them inside `anova`.

```
anova(coxph(Surv(survive, chd) ~ sbpfifths, data = data_11_8),
      coxph(Surv(survive, chd) ~ as.numeric(sbpfifths), data = data_11_8))
```

```
Analysis of Deviance Table
 Cox model: response is  Surv(survive, chd)
 Model 1: ~ sbpfifths
 Model 2: ~ as.numeric(sbpfifths)
  loglik Chisq Df Pr(>|Chi|)
1  -1604
2  -1605  1.23  3       0.75
```

Example 11.10 (page 531)

Reworking example 11.10 is similar to our previous example, except for the data and the explanatory variables we use.

```
read_table("K11828 supplements/Datasets/Example 10.12.DAT",
           col_names = c("age", "chol", "bmi", "sbp",
                         "smoke", "active", "chd", "survive"),
           col_types = cols( smoke = col_factor(levels = c("1","2","3")),
                             active = col_factor(levels =c("1","2","3")))
           )-> data_10_12
coxph(Surv(survive, chd) ~ age + chol + bmi + sbp + smoke +active,
      data = data_10_12)
```

```
Call:
coxph(formula = Surv(survive, chd) ~ age + chol + bmi + sbp +
    smoke + active, data = data_10_12)

           coef exp(coef) se(coef)    z     p
age      0.0180    1.0182   0.0132  1.4 0.171
chol     0.2861    1.3312   0.0550  5.2 2e-07
bmi      0.0381    1.0388   0.0205  1.9 0.063
sbp      0.0200    1.0203   0.0036  5.5 4e-08
smoke2   0.3121    1.3663   0.2441  1.3 0.201
smoke3   0.6999    2.0135   0.2134  3.3 0.001
active2 -0.1886    0.8281   0.1732 -1.1 0.276
active3 -0.1098    0.8961   0.2242 -0.5 0.624

Likelihood ratio test=89  on 8 df, p=7.4e-16
n= 4049, number of events= 196
```

Example 11.11 (page 532)

We will now turn our attention to example 11.11.

```
read_table("K11828 supplements/Datasets/Example 10.23.DAT",
           col_names = c("sex", "bscore", "bqrtr","chd", "survive"),
           col_types = cols(sex = col_factor(levels = c("1","2")),
                            bqrtr = col_factor(levels = c("1","2","3","4")))
           ) -> data_10_23
relevel(data_10_23$sex, ref = "2") -> data_10_23$sex
coxph(Surv(survive, chd) ~ sex *  bqrtr,
      data = data_10_23) -> cph_11_11_fb
relevel(data_10_23$sex, ref = "1") -> data_10_23$sex
coxph(Surv(survive, chd) ~ sex *  bqrtr,
      data = data_10_23) -> cph_11_11_mb
```

After importing data, we use `relevel` to change the reference level for *sex*, so that females would be taken as the baseline as in the textbook. This will give us the values in output 11.5. If we avoid changing the reference level or if we change it back to “1”, we will get the values in output 11.6. Fitting the proportional hazards model in either case is similar to our previous examples. We may use `summary` or `confint` to obtain the confidence interval of the coefficients.

Table 11.5 (page 534)

To prepare table 11.5, we will use `tidy`.

```
bind_rows(tidy(cph_11_11_mb, conf.int = TRUE, exponentiate = TRUE) |>
            filter(str_starts(term, "bqrtr")) |>
            mutate(sex = "Male",
                   quarter =  str_sub(term, -1,-1),
                   result = paste0(round(estimate,2),
                                   " (",
                                   round(conf.low, 2),
                                   ",",
                                   round(conf.high,2),
                                   ")")) |>
            select(quarter, sex, result),
          tidy(cph_11_11_fb, conf.int = TRUE, exponentiate = TRUE) |>
            filter(str_starts(term, "bqrtr")) |>
            mutate(sex = "Female",
                   quarter =  str_sub(term, -1,-1),
                   result = paste0(round(estimate,2),
                                   " (",
                                   round(conf.low, 2),
                                   ",",
                                   round(conf.high,2),
                                   ")")) |>
            select(quarter, sex, result)) |>
  pivot_wider(names_from = "sex", values_from = result)
```

We use `bind_rows` to row bind two sets of dataframes, one for males and one for females, both returned by `tidy`. We ask `tidy` to return confidence intervals in addition to the coefficients and to exponentiate the values. For males, we supply `tidy` with the Cox model in which male sex was specified as the reference level and for females the Cox model in which female sex was the reference level. We use `filter` to select from the data frame returned by `tidy`, only those rows which give the values for the Bortner quarters. We add a new column to indicate sex, another to indicate the Bortner quarter and one with the hazard ratio and its confidence interval. Finally, we select only those columns that we want to show. We use `pivot_wider` to separate out the values of the two sexes into different columns.

TABLE 11.3
Replication of table 11.5

	Sex	
Quarter of Bortner score	Male	Female
2	1.1 (0.76,1.58)	0.56 (0.32,0.99)
3	0.75 (0.5,1.12)	0.29 (0.14,0.59)
4	0.8 (0.54,1.18)	0.33 (0.16,0.67)

Example 11.12 (page 534)

To rework example 11.12, we set the reference level for sex to female.

```
relevel(data_10_23$sex, ref = "2") -> data_10_23$sex
coxph(Surv(survive, chd) ~ sex *  bscore,
      data = data_10_23) -> cph_11_12_fb
cph_11_12_fb
```

Fitting an interaction model is similar to our earlier examples. We use * to say that we want interaction terms in addition to the main effects.

```
Call:
coxph(formula = Surv(survive, chd) ~ sex * bscore, data = data_10_23)

               coef exp(coef) se(coef)    z     p
sex1        -0.6133    0.5416   0.5723 -1.1 0.284
bscore      -0.0129    0.9872   0.0031 -4.1 4e-05
sex1:bscore  0.0105    1.0105   0.0036  2.9 0.003

Likelihood ratio test=84  on 3 df, p=<2e-16
n= 8157, number of events= 268
```

We may obtain the variance covariance matrix using `vcov`.

```
vcov(cph_11_12_fb)
```

```
                  sex1      bscore sex1:bscore
sex1         0.3274821  1.4969e-03 -1.9886e-03
bscore       0.0014969  9.8761e-06 -9.8760e-06
sex1:bscore -0.0019886 -9.8760e-06  1.2851e-05
```

Before we move on to the next section, note that `coxph` can handle time dependent covariates and recurrent events.

11.4 The Weibull proportional hazards model

Example 11.13 (page 538)

To rework example 11.13, we will use the variables from one of our previous examples.

```
data.frame(time = time_11_7,
           events = events_11_7,
           cell = cell_11_7) -> data_11_13
```

```
survreg(Surv(time, events) ~ cell,
        dist='weibull',
        data = data_11_13) -> wbph_11_13

1 / wbph_11_13$scale -> gamma_11.13
exp(-wbph_11_13$coefficients["(Intercept)"]/
      wbph_11_13$scale) -> lambda_11_13
- wbph_11_13$coefficients["cellhigh"]/
  wbph_11_13$scale -> coef_11_13

library(SurvRegCensCov)
ConvertWeibull(wbph_11_13)
```

We use `data.frame` to bind together the relevant variables. We then use `survreg` to fit a Weibull proportional hazards model. The difference from our previous example on Weibull model is that the right-hand side of the formula has the explanatory variable *cell* instead of 1. Following the formula given in the textbook, we calculate the Weibull shape and scale. The coefficient for high cellularity was also calculated from the coefficient of the model. We may use them to estimate the hazard function or survival function as given in the textbook. If we desire standard errors, we may use `vcov` to obtain the variance covariance matrix. We use `ConvertWeibull` from `SurvRegCensCov` to present the result of the survival regression in an easily understandable form. Remember to install it using `install.packages` as discussed in chapter 1.

```
$vars
         Estimate        SE
lambda    0.00094 0.0010585
gamma     1.74069 0.2591691
cellhigh  0.63281 0.4306925

$HR
             HR      LB     UB
cellhigh 1.8829 0.80951 4.3796

$ETR
             ETR      LB     UB
cellhigh 0.69521 0.43252 1.1175
```

The ETR in the result stands for event time ratio and HR for hazard ratio.

Figure 11.13 (page 540)

We will now prepare the graph shown in figure 11.13.

```
survfit(Surv(time, events)  ~ cell,
        data = data_11_13) -> km_11_13
bind_rows(data.frame(survival = 1,
                     strata =  factor("cell=high"),
                     time = 0),
```

```
          data.frame(survival = 1,
                     strata =  factor("cell=low"),
                     time = 0),
          data.frame(survival = summary(km_11_13)$surv,
                     strata = summary(km_11_13)$strata,
                     time = summary(km_11_13)$time)) -> tbl_11_13
```

We use `survfit` to get the Kaplan Meir curves. We bind together the relevant columns of the KM curves to build a data frame that we will supply to `ggplot`. We also add two rows to indicate the survival of 1 at time zero. We need two rows, because there are two strata. Note that in these two rows, the `strata` are given values *cell=high* and *cell=low*, rather than *high* and *low*. This to make the values align with the values returned by `survfit`.

```
ggplot(tbl_11_13 ) +
  geom_step(aes(y = survival,
                x = time,
                group = strata,
                colour = strata)) +
  geom_function(fun = function(x) {exp(-lambda_11_13 * x ^ gamma_11.13)},
                linetype = 2,
                colour =  "#111111") +
  geom_function(fun = function(x) {exp(-lambda_11_13 * exp(coef_11_13) *
                                         x ^ gamma_11.13)},
                linetype = 2,
                colour =  "#004B73") +
  labs(x = "Time (weeks",
       y = "Probability of survival") +
  scale_color_manual(values = c("#111111", "#004B73"))
```

We supply `ggplot` with the data frame we built, specify the geom as `geom_step` and specify the appropriate aesthetics. We specify `group` and `colour` aesthetics to `strata` to get two curves, one each for each level of cellularity. We use `geom_function` twice to draw the Weibull survival curves, supplying an anonymous function for each curve. The functions are similar except for inclusion of *coef_11_13* in the curve for high cellularity. The functions are realisations of the equations given in the textbook.

We can use `anova` to compare the model with the null model.

```
anova(wbph_11_13)
```

```
Analysis of Deviance Table

 distribution with  link

Response: Surv(time, events)
```

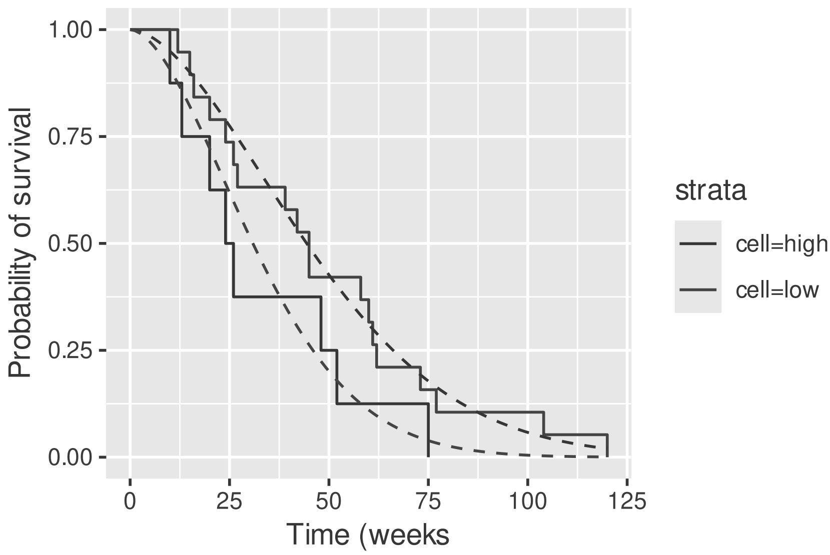


FIGURE 11.5
Replication of figure 11.13

```
Scale estimated

Terms added sequentially (first to last)
     Df Deviance Resid. Df -2*LL Pr(>Chi)
NULL                    25   249
cell  1     1.98        24   247     0.16
```

The result is similar to that in the textbook.

Example 11.14 (page 540)

To rework example 11.14, we use the data from the previous example.

```
survreg(Surv(survive, chd) ~ age + chol + bmi +sbp + smoke + active,
         data = data_10_12,
         dist = "weibull") -> wbph_11_14
summary(wbph_11_14)
```

The code to fit a Weibull proportional model is similar to our previous examples, except for the many explanatory variables on the right-hand side of the model formula. The `summary` method displays the information shown in output 11.9.

```
Call:
survreg(formula = Surv(survive, chd) ~ age + chol + bmi + sbp +
    smoke + active, data = data_10_12, dist = "weibull")
              Value Std. Error     z       p
(Intercept) 15.78271    0.92411 17.08 < 2e-16
age         -0.01415    0.01050 -1.35 0.17780
```

```
chol        -0.22729    0.04637 -4.90 9.5e-07
bmi         -0.03044    0.01643 -1.85 0.06389
sbp         -0.01599    0.00308 -5.19 2.1e-07
smoke2      -0.24895    0.19442 -1.28 0.20039
smoke3      -0.55459    0.17371 -3.19 0.00141
active2      0.15192    0.13774  1.10 0.27004
active3      0.09164    0.17793  0.52 0.60653
Log(scale)  -0.23127    0.07007 -3.30 0.00096

Scale= 0.794

Weibull distribution
Loglik(model)= -2290.3   Loglik(intercept only)= -2335
    Chisq= 89.39 on 8 degrees of freedom, p= 6.2e-16
Number of Newton-Raphson Iterations: 10
n= 4049
```

Table 11.7 (page 541)

We will extract the `table` component returned by `summary` to build table 11.7.

```
data.frame(summary(wbph_11_14)$table) -> tbl_11_7

mutate(tbl_11_7,
       variable = row.names(tbl_11_7),
       B = round(Value, 4),
       b = round(- Value / wbph_11_14$scale, 4),
       phi = round(exp(b),2)) |>
  filter(variable != "(Intercept)" & variable != "Log(scale)") |>
  select( B, b, phi)
```

We save the `table` component returned by `summary` as a data frame. We use `mutate` to build new columns *B*, *b* and *phi* from the `Value` column using the formula given in the textbook. The variable's name is given as the row name. Negative subsetting based on character value is not possible. In other words, we cannot ask R to return a modified data frame excluding certain rows by specifying row names. So, we create a new column from the row names. Now, we can `filter` this column to avoid rows of our choice. We exclude the rows corresponding to intercept and scale. Finally, we select the columns that we want to show.

11.5 Model checking

Example 11.15 (page 542)

To rework example 11.15, we use the table we prepared earlier.

```
tbl_11_1 |>
  mutate(logtime = log(`Time (t)`),
```

```
        lch = log(- log(`Survival (s)`))) -> tbl_11_8
select( tbl_11_8, `Time (t)`, logtime, `Survival (s)`, lch)
```

TABLE 11.4
Replication of table 11.7

	Regression coefficients		
Explanatory variable	B	b	Φ
age	−0.0142	0.0178	1.02
chol	−0.2273	0.2864	1.33
bmi	−0.0304	0.0384	1.04
sbp	−0.0160	0.0201	1.02
smoke2	−0.2490	0.3137	1.37
smoke3	−0.5546	0.6989	2.01
active2	0.1519	−0.1915	0.83
active3	0.0916	−0.1155	0.89

We `mutate` *tbl_11_1* to produce two new columns *logtime* and *lch*. The values in the column are calculated according to the formula given in the text. Note that the values calculated for the start and end of the time periods are infinity. We `select` our choice of columns to display.

```
# A tibble: 24 x 4
   `Time (t)` logtime `Survival (s)`      lch
        <dbl>   <dbl>          <dbl>    <dbl>
 1          0 -Inf             1      -Inf
 2         10    2.30          0.963   -3.28
 3         12    2.48          0.926   -2.56
 4         13    2.56          0.889   -2.14
 5         15    2.71          0.852   -1.83
 6         16    2.77          0.815   -1.59
 7         20    3.00          0.741   -1.20
 8         24    3.18          0.667   -0.903
 9         26    3.26          0.593   -0.648
10         27    3.30          0.556   -0.531
# i 14 more rows
```

Figure 11.14 (page 543)

We use the table to plot the graph given in figure 11.14.

```
ggplot(tbl_11_8, aes(x = logtime, y = lch)) +
  geom_line() +
  geom_point() +
  labs(x = "Log of survival time",
       y = "Log cumulative hazard")
```

We use ggplot. As, the aesthetic values are common to both the geoms we intend to use, we specify them inside ggplot. We use geom_line and geom_point to draw the graph.

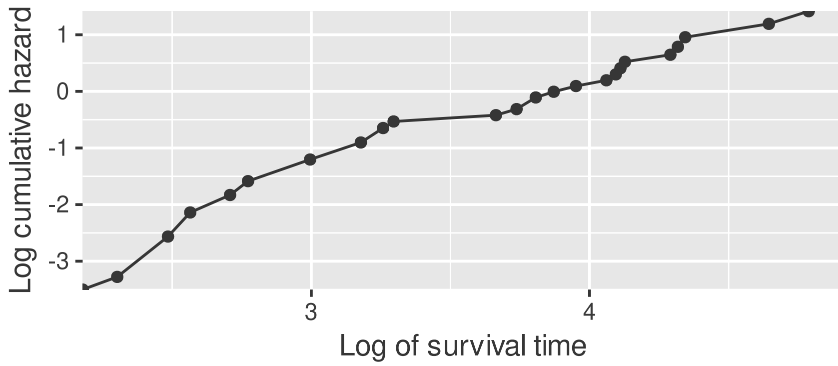


FIGURE 11.6
Replication of figure 11.14

Figure 11.15 (page 544)

We can prepare the graph in figure 11.15 in a manner similar to our previous example.

```
tbl_11_13 |>
  mutate(logtime = log(time),
         lch = log(- log(survival))) -> tbl_11_16
ggplot(tbl_11_16,
       aes(x = logtime, y = lch,group =  strata,colour = strata)) +
  geom_line() +
  geom_point() +
  scale_color_manual(values = c("#111111", "#004B73")) +
  labs(x = "Log of survival time",
       y = "Log cumulative hazard")
```

The difference from our previous example is that we provide group and colour arguments to ggplot. We do this to have different lines with different colours for each value in the strata variable.

Example 11.17 (page 544)

To rework the example 11.17, we need the Kaplan Meir curves for example 11.8.

```
survfit(Surv(survive, chd) ~  cholfifths,
        data = data_11_8) -> km_11_17
data.frame(strata = summary(km_11_17)$strata,
           time = summary(km_11_17)$time,
           survival = summary(km_11_17)$surv) |>
  mutate(logtime = log(time),
         lch = log(- log(survival))) -> tbl_11_17
ggplot(tbl_11_17,
```

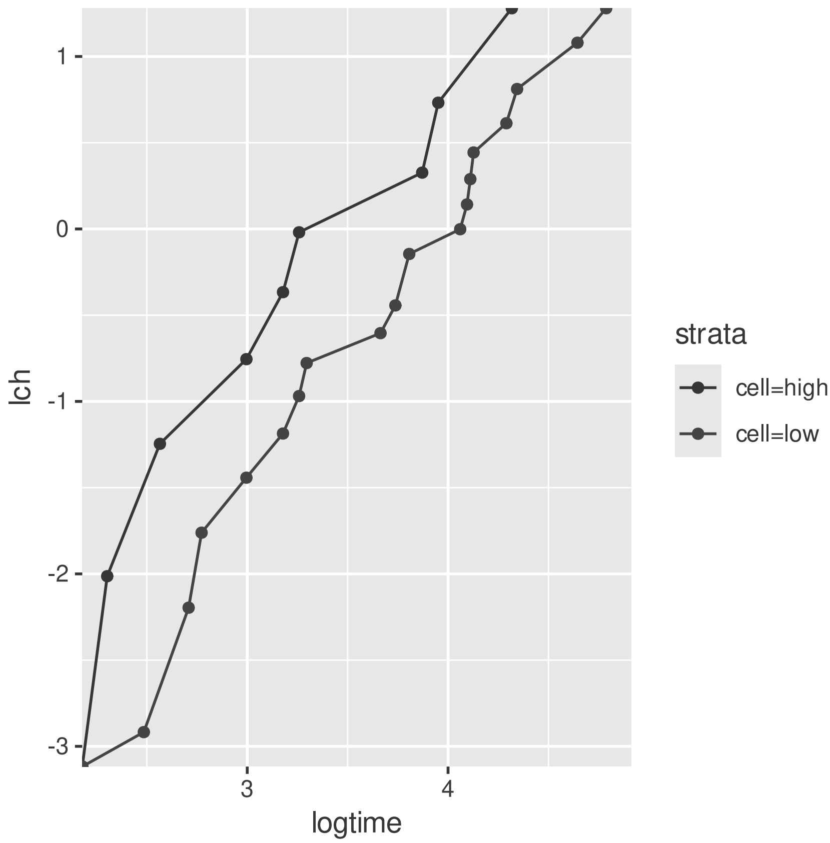


FIGURE 11.7
Replication of figure 11.15

```
       aes(x = logtime,
           y = lch,
           group =  strata,
           colour = strata,
           linetype = strata)) +
  geom_line()
```

Except for the fact that we needed to fit a KM curve, the steps for building the graph are similar to the previous examples. Here, we omitted `geom_point` as including it will cause crowding in the graph.

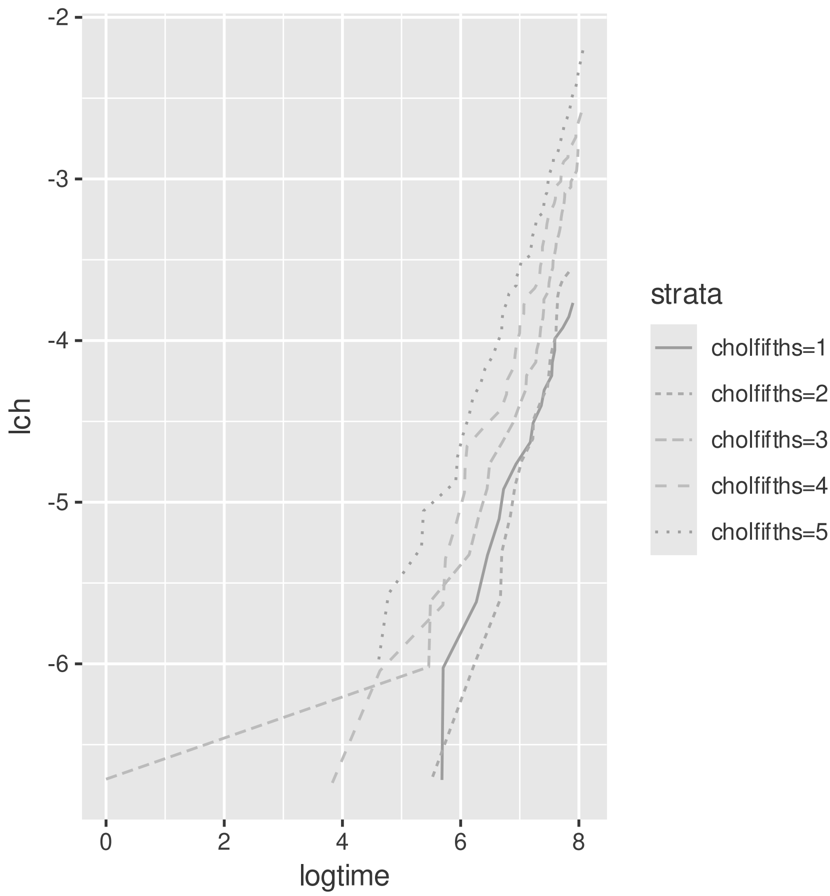


FIGURE 11.8
Replication of figure 11.16

Example 11.18 (page 545)

To rework example 11.18, we use the data from our previous example.

```
coxph(Surv(time,events) ~ cell ,
      ties = "breslow",
      data = data_11_13) -> cph_11_18c
survSplit(Surv(time, events) ~ cell,
          data=data_11_13,
          cut=data_11_13$time) -> newdata
coxph(Surv(tstart,time,events) ~ cell + tt(as.numeric(cell)),
      data =newdata,
      ties = "breslow",
      tt = function(x, t,...) x*t) -> cph_11_18tc
-2 * (cph_11_18c$loglik[2] - cph_11_18tc$loglik[2]) -> chistat
pchisq(chistat,df =1, lower.tail = FALSE)
```

First, we fit a Cox proportional hazards model and save with a suitable name. We use `survSplit` to create a new data frame. Each record in the `data` that is supplied to `survSplit` is split into multiple sub-records for each of the time specified in `cut`. It is sufficient to use the unique time periods. Here, we use the *time* component of *tbl_11_18* as the cut time. The new data frame is in a **counting process format**. This means that there will be a new column to indicate a start time, which will be named `tstart`. In the next step, we fit a Cox proportional hazards model using the *newdata*. There are some differences from the previous examples. First, `Surv` is supplied the `tstart` variable as the first option and *time* as the second option. Thus `Surv` will understand that the supplied dataframe is in counting process format. Second, the response variable supplied includes `tt(as.numeric(cell))`. We use `as.numeric` because we need to convert *cell*, a factor to numeric as expected by the function. The term `tt()` is to indicate that `coxph` needs to calculate a time varying variable from its argument *as.numeric(cell)*. How exactly the time varying variable is calculated depends on the value of `tt` argument. In our case, we supply an anonymous function which accepts two named variables x and t. `coxph` will supply the *time* and *as.numeric(cell)* to these variables when the function is called. The function returns the product of the supplied variables.

We calculate the difference of `loglik` components of the two models and multiply it with `-2` to calculate the statistic. We use `pchisq` to calculate the significance of the statistic. We can find the value of the coefficient corresponding to the time varying variable by printing *cph_11_18tc* to confirm that it agrees with the text value.

```
[1] 0.83277
```

Note that the method we followed would look like a rather convoluted path. We may think that modifying the original `coxph` call to include the term `as.numeric(cell) * time` as a predictor variable could achieve the same result. However, it is not so. We need `tt` to calculate the time dependent covariate correctly. But, we needn't follow this path at all. `survival` provides `cox.zph`, a function that will test the proportional hazards assumption directly. However, it doesn't use the methodology described in the textbook.

Example 11.19 (page 546)

To rework example 11.19, we fit the different models.

```
survreg(Surv(time, events) ~ cell,
        dist='weibull',
        data = data_11_13,
        subset = cell == "low") -> wbph_11_19lc
survreg(Surv(time, events) ~ cell,
        dist='weibull',
        data = data_11_13,
        subset = cell == "high") -> wbph_11_19hc
survreg(Surv(time, events) ~ cell,
        dist='weibull',
        data = data_11_13) -> wbph_11_19all
-2 * ( wbph_11_19all$loglik[2] -
         (wbph_11_19lc$loglik[2] +
            wbph_11_19hc$loglik[2])) -> stat_11_19
stat_11_19
pchisq(stat_11_19, df = 1,lower.tail = FALSE)
```

The difference from our previous examples is that we use the `subset` argument to restrict model fitting to suit our needs. We calculate the test statistic from the `loglik` component of the models. We find the significance of the statistic using `pchisq`.

```
[1] 0.021186
[1] 0.88427
```

11.5.1 Competing risk

Example 11.20 (page 547)

Fitting a Fine and Gray model is a two step process.

```
read_table("K11828 supplements/Datasets/Example 11.20.dat",
           col_names = c("smoker", "age", "time", "event"),
           col_types = cols(smoker = col_factor(levels = c("0","1")),
                            age = col_double(),
                            time = col_double(),
                            event = col_factor( levels =c("0","1","2")))
           ) -> data_11_20
finegray(Surv(time, event) ~ .,
         data = data_11_20) -> fgdata_11_20
coxph(Surv(fgstart, fgstop, fgstatus) ~ smoker + age,
      weight=fgwt,
      data = fgdata_11_20) -> fgmdl_11_20
```

After importing the data, the first step is to create a modified data frame. We use `finegray` for this step. It accepts a formula whose left-hand side is `Surv` object. The `.` on the right-hand side stand for all the remaining columns in the dataframe supplied as `data` argument. The result is stored. This dataframe contains four new columns instead of those used on the left-hand side of the formula. These are named `fgstart`, `fgstop`, `fgstatus` and `fgwt`. The columns specified by the right-hand side of the formula are preserved as such.

In the second step, we use the datafame created by `finegray` to fit a weighted Cox model. The left-hand side of the formula we supply to `coxph` uses the new columns `fgstart`, `fgstop`, `fgstatus` created by `finegray`. The `fgwt` is given as the `weight` argument for `coxph`.

We will print the Fine and Gray model to confirm that the coefficients are similar to that in output 11.11.

```
Call:
coxph(formula = Surv(fgstart, fgstop, fgstatus) ~ smoker + age,
    data = fgdata_11_20, weights = fgwt)

         coef exp(coef) se(coef) robust se   z    p
smoker1 0.530     1.698    0.227     0.222 2.4 0.02
age     0.035     1.036    0.037     0.037 1.0 0.34

Likelihood ratio test=5.7  on 2 df, p=0.058
n= 199, number of events= 83
```

To fit the wrong Cox model, ignoring the information given by the competing risk, we need to modify the data frame.

```
mutate(data_11_20,
       eventmod = ifelse(event == "1",1,0)) -> data_11_20mod
coxph(Surv(time, eventmod) ~ smoker + age,
      data = data_11_20mod) -> cxph_11_20
```

We create a new column *eventmod* from *event*. All values except *1* are converted to zero; *1* is retained as one. Thus, the competing risk indicated by *2* is treated as censored. Also note that *event* was originally factor, but *eventmod* is numeric. We use this new column inside `Surv` and fit a Cox model using `coxph`.

We print the model object to confirm that the coefficients are similar to that on output 11.10.

```
Call:
coxph(formula = Surv(time, eventmod) ~ smoker + age, data = data_11_20mod)

         coef exp(coef) se(coef)   z     p
smoker1 0.846     2.330    0.232 3.6 3e-04
age     0.058     1.060    0.036 1.6   0.1

Likelihood ratio test=13  on 2 df, p=0.0013
n= 160, number of events= 83
```

Figure 11.17 (page 549)

We use base graphics to prepare the graphs in figure 11.17.

```
expand.grid(age = mean(data_11_20$age),
            smoker = c("0", "1")) -> new_data

par(mfrow =  c(1,2))
cxplt <- plot(survfit(cxph_11_20, newdata= new_data),
              fun = function(x) 1-x,
              col = c("blue", "black"),
              lty = 2:1,
              conf.int = FALSE,
              xscale = 365.25,
              xlab = "Time (years)",
              ylab = "cumulative incidence",
              ylim = c(0,0.9))
text(cxplt, c("Non-smokers", "Smokers"), adj = c(1,-0.25))
title("Cox analysis")
fgplt <- plot(survfit(fgmdl_11_20, newdata= new_data),
              fun = function(x) 1-x,
              col = c("blue", "black"),
              lty = 2:1,
```

```
              conf.int = FALSE,
              xscale = 365.25,
              xlab = "Time (years)",
              ylab = "Cumulative Incidence",
              ylim = c(0,0.9))
text(fgplt, c("Non-smokers", "Smokers"), adj = c(1,-0.25))
title("Fine and Gray analysis")
```

First, we create a new data frame using `expand.grid`. We ask `expand.grid` to build a data frame using the unique combination of values from each of the vectors supplied. In our case, we get two rows with two columns. One column will be named *age* and will have the mean *age* of our original data frame in both rows. The second column *smoker* will have the values *0* and *1* corresponding to the *smoker* column of the original data set.

We call `survfit` to produce survival curves from the models we fitted. Each time, `survfit` is provided the relevant model object and the *newdata*. The `plot` command is provided many more arguments in addition to the `survfit` object. One important argument is `fun`. We specify an anonymous function which can accept a named argument `x`. The function returns the 1 minus the supplied value, the cumulative incidence. Thus, `plot` will use the survival curve to calculate the cumulative incidence from the survival curve and plot it. We use `xscale` to scale the x axis by a factor of 365.25. Thus, we get the number of years in the x axis instead of days. We supply `ylim` to both the graphs so that the y axis spread is the same in both the graphs. The arguments `col`, `lty` each with a vector of two values specify the colour and line type of each of the two lines drawn. The command `par(mfrow = c(1,2))` given before the `plot` commands asks R to print the next plots in a one row, two column grid. Thus, we get the two plots side by side. However, we save the plots instead of drawing them immediately. We do this to add labels to the curves using `text`. The argument `adj` is used to adjust the position of the labels. We add titles to the graphs using `title`.

It is unclear how cumulative incidence was calculated in our text. Though we calculated the cumulative incidence as 1 minus survival, it is not valid when competing risks are to be accounted for. We should prefer to use `cuminc` function from `cmprsk` package. The `cmprsk` pack also provide `crr` to fit Fine and Gray models.

11.6 Poisson regression

Example 11.21 (page 551)

We can fit Poisson regression models using `glm`.

```
data.frame(house = c("rented", "owned"),
           events =  c(115, 104),
           pyears = c(14200.945,18601.467)) -> data_11_21
glm(events ~ offset(log(pyears)) + house,
    family = "poisson",
    data = data_11_21) -> psn_11_21
```

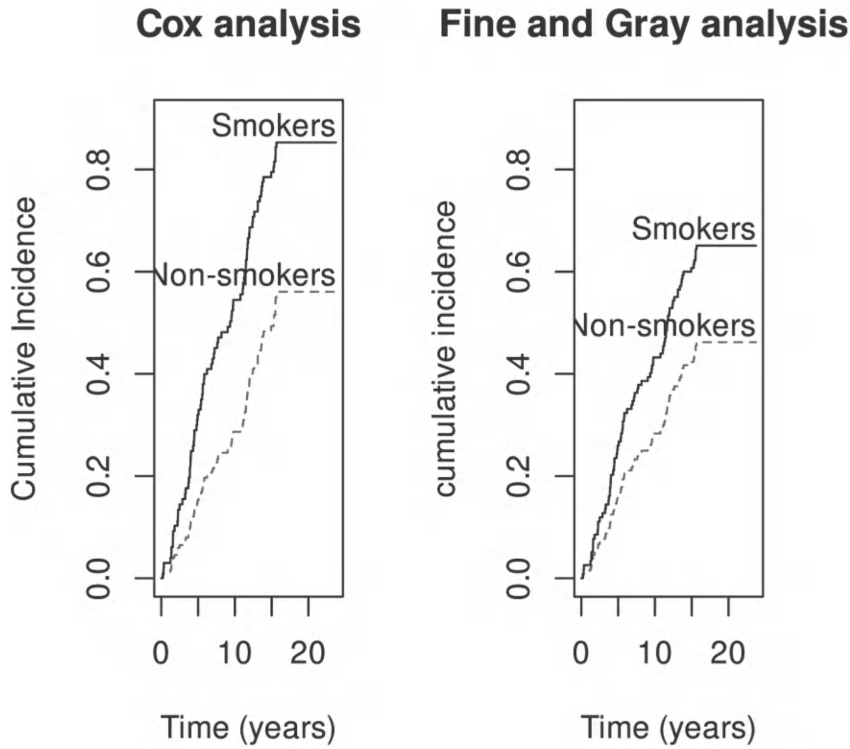


FIGURE 11.9
Replication of figure 11.17

First, we prepare a data frame to hold the data in a form appropriate for `glm`. We call `glm` in a manner similar to our previous examples. The difference is that we use `offset` to specify that the log of *pyears* is the offset and that the value supplied to `family` argument is `poisson`.

```
tidy(psn_11_21, conf.int = 0.95)
```

We use `tidy` to print the coefficients and their confidence interval to confirm that they agree with output 11.12.

```
# A tibble: 2 x 7
  term        estimate std.error statistic p.value conf.low conf.high
  <chr>          <dbl>     <dbl>     <dbl>   <dbl>    <dbl>     <dbl>
1 (Intercept)   -5.19     0.0981    -52.9  0          -5.39     -5.00
2 houserented    0.370    0.135       2.74 0.00619     0.105     0.637
```

We may ask `tidy` to exponentiate the result.

```
tidy(psn_11_21, exponentiate = TRUE, conf.int = 0.95)
```

```
# A tibble: 2 x 7
  term        estimate std.error statistic p.value conf.low conf.high
  <chr>          <dbl>     <dbl>     <dbl>   <dbl>    <dbl>     <dbl>
1 (Intercept)  0.00559    0.0981    -52.9  0         0.00458   0.00674
2 houserented  1.45       0.135       2.74 0.00619   1.11      1.89
```

The exponentiated estimate for intercept and its confidence interval gives us the event rate for owners (which we will have to multiply by 1000 to get the per thousand rate). The exponentiated estimate for renters gives us the relative rate and its confidence interval. To get the event rate for renters, we need to manually add the original coefficients and then exponentiate it. For calculating its confidence interval using the formula given in the textbook, we need `vcov` to obtain the variance covariance matrix.

Example 11.22 (page 553)

For reworking the example 11.22, we prepare the de-aggregated data.

```
data.frame(age = factor(rep(c("40-44", "45-49", "50-54",
                              "55-59", "60-64", "65-69"),
                            2)),
           house = factor(rep(c("rented", "owned"), each = 6)),
           events = c(2,24,31,28,28,2,3,19,25,27,26,4),
           pyears = c(1107.447,3058.986,
                      3506.53,3756.65,
                      2419.622,351.71,
                      1619.328,4550.166,
                      4857.904,4536.832,
                      2680.843,356.394)) -> data_11_22
anova(glm(events ~ offset(log(pyears)) + age * house ,
          family = "poisson",
          data = data_11_22),
      test = "Chisq")
```

We provide to `anova`, the model that we propose to fit. It returns an analysis of deviance table showing results as if the terms were added sequentially. We also specify the `test` to be performed to determine the significance of the terms.

```
Analysis of Deviance Table

Model: poisson, link: log

Response: events

Terms added sequentially (first to last)

          Df Deviance Resid. Df Resid. Dev Pr(>Chi)
NULL                         11       35.1
age        5    24.97         6       10.1  0.00014 ***
house      1     6.44         5        3.7  0.01114 *
age:house  5     3.66         0        0.0  0.59917
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Having found that the interaction term is not significant, we fit the simpler model

```
relevel(data_11_22$age, ref = "65-69") -> data_11_22$age
glm(events ~ offset(log(pyears)) + age + house ,
    family = "poisson",
    data = data_11_22) -> psn_11_22
```

We use `relevel` to change the reference level for the variable *age* to align with the textbook results. The use of `glm` is similar to our previous example, except for the model formula.

```
tidy(psn_11_22)
```

We confirm that the results agree with output 11.13 given in our text.

```
# A tibble: 7 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)   -4.96      0.416   -11.9   9.28e-33
2 age40-44      -1.50      0.606    -2.48  1.33e- 2
3 age45-49      -0.372     0.436    -0.854 3.93e- 1
4 age50-54      -0.209     0.430    -0.486 6.27e- 1
5 age55-59      -0.230     0.430    -0.535 5.93e- 1
6 age60-64       0.230     0.430     0.535 5.92e- 1
7 houserented    0.344     0.136     2.54  1.12e- 2
```

Similar to our previous example, we may ask `tidy` to exponentiate the estimates and confidence intervals if we desire so.

11.6.1 Comparison of standardised event ratios

Example 11.23 (page 555)

Comparing standardised event ratios is essentially the same as our previous examples. Here we rework example 11.23.

```
data.frame(deaths =  c(4,43),
           expect = c(1.488, 38.755),
           factory = factor(c("a", "b"))) -> data_11_23
glm(deaths ~ offset(log(expect)) + factory,
    data = data_11_23,
    family = "poisson") -> psn_11_23
tidy(psn_11_23)
```

As with our previous examples, we prepare a data frame and use `glm`. The point to note is that the log of the expected number is specified as the offset.

```
# A tibble: 2 x 5
  term        estimate std.error statistic p.value
  <chr>          <dbl>     <dbl>     <dbl>   <dbl>
1 (Intercept)    0.989     0.500      1.98  0.0480
2 factoryb      -0.885     0.523     -1.69  0.0905
```

Example 11.24 (page 556)

To rework example 11.24, we prepare the data.

```
data.frame(deprive = factor(rep(c("I", "II", "III", "IV"),
                                each = 8)),
           agegrp = factor(rep(c("25-29", "30-34","35-39","40-44",
                                 "45-49", "50-54","55-59", "60-64"),
                               4)),
           events = c(0,0,1,6,7,16,17,25,0,0,4,7,13,11,28,44,
                      0,0,1,9,17,19,43,53,0,1,5,10,15,24,28,56),
           popln = c(4784,4210,3396,3226,2391,2156,2182,2054,4972,4045,
                     3094,2655,2343,2394,2597,2667,4351,3232,2438,2241,
                     2360,2708,2968,2802,4440,3685,2966,2763,2388,2566,
                     2387,2380)) -> data_11_24
anova(glm(events ~ offset(log(popln)) + agegrp * deprive,
          family = "poisson",
          data = data_11_24),
      test = "Chisq")
```

We supply the model with interaction term to `anova` to get the analysis of deviance table. Note that this won't give the row corresponding to row 3 of table 11.10. If we need that row, we will need to call `anova` with that model.

```
Analysis of Deviance Table

Model: poisson, link: log

Response: events

Terms added sequentially (first to last)

               Df Deviance Resid. Df Resid. Dev Pr(>Chi)
NULL                              31        698
agegrp          7      665        24         33   <2e-16 ***
deprive         3       20        21         13   0.0002 ***
agegrp:deprive 21       13         0          0   0.8916
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Once the anova table informs us that the interaction terms are not significant, we fit the simpler model. We need to `relevel` the age group variable to align the results with that of the textbook.

```
relevel(data_11_24$agegrp, ref = "60-64") -> data_11_24$agegrp
glm(events ~ offset(log(popln)) + agegrp + deprive,
    family = "poisson",
    data = data_11_24) -> psn_11_24
```

The model fitting command is different only in the model formula and the offset used.

Table 11.11 (page 557)

We will prepare only the column corresponding to Poisson regression of table 11.11.

```
rownames_to_column(data.frame(estimate = coefficients(psn_11_24),
                              lower = confint.default( psn_11_24)[,1],
                              upper = confint.default( psn_11_24)[,2])) |>
  filter(str_starts(rowname, "deprive")) |>
  transmute(`Deprivation Group` = str_remove(rowname, "deprive"),
            Poisson = paste0(round(exp(estimate),2),
                             " (",
                             round(exp(lower), 2),
                             ",",
                             round(exp(upper), 2), ")"))
```

First, we prepare a data frame containing the coefficients and their confidence intervals. By default, R uses a method called profile likelihood for estimating the confidence interval for `glm` objects. In this example however, R is unable to calculate the confidence interval by the profile likelihood method. So, we ask R to use the normal approximation by calling `confint.default`. We use `rownames_to_column` to include a column in the dataframe with values corresponding to its rownames. The result of `coefficients` and `confint.default` are vector and matrix respectively. When we join them to form a data frame, each row has a name assigned corresponding to the coefficient's name. This is not a column that we can manipulate like other columns. In order to make it manipulable, we make it a new column.

Next, we `filter` the dataframe to output only those rows corresponding to the deprivation groups. We use `transmute` to make two new columns with the result we want. The function `transmute` is similar to `mutate` except that `transmute` will return only the newly made columns. This saves us an extra step of hiding / deleting the columns that we don't want to show.

TABLE 11.5
Replication of table 11.11

Deprivation group	Poisson regression method
II	1.3 (0.96,1.75)
III	1.62 (1.22,2.16)
IV	1.77 (1.33,2.36)

To fit a model postulating linear trend of the effect of deprivation group, we need to create a new column with the appropriate value.

```
as.numeric(data_11_24$deprive) -> data_11_24$depriven
anova(glm(events ~ offset(log(popln)) + agegrp + depriven +
            I(depriven^2) + I(depriven^3),
          family = "poisson",
          data = data_11_24),
      test = "Chisq")
```

We coerce the factor variable *deprive* to numeric using `as.numeric`. This works because factors are internally represented as numbers and the numerical code used for the various factor levels agree with the numeric value we need for each level. We use the new column *depriven* to specify the model. We use `anova` to build an analysis of deviance table. We specify a model with upto the cubic power of the new variable we created. Each power of *depriven* is specified inside `I`. This is done to insulate the mathematical expression inside it from being assigned the special meanings within a model formula. Thus `depriven ^ 2` will be treated as *depriven* squared. Note that our analysis of deviance model doesn't have the second row of table 11.12. If we really want to include it, we will have to specify another `anova` command with the relevant model and add the corresponding row to our table.

```
Analysis of Deviance Table

Model: poisson, link: log

Response: events

Terms added sequentially (first to last)

              Df Deviance Resid. Df Resid. Dev Pr(>Chi)
NULL                             31        698
agegrp         7      665        24         33  < 2e-16 ***
depriven       1       19        23         14  1.6e-05 ***
I(depriven^2)  1        1        22         14     0.34
I(depriven^3)  1        0        21         13     0.80
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The model named "depriven" is the linear model, followed by the quadratic and the cubic models.

11.7 Pooled logistic regression

Example 11.25 (page 560)

We will use the package "Greg" to prepare our data for pooled logistic regression. Remember to install it using `install.packages` as discussed in chapter 1.

```
read_table("K11828 supplements/Datasets/Example 5.9.DAT",
                                 col_names = c("age","house","chd","survive"),
```

```
                              col_types = cols(house = col_factor())) |>
  rownames_to_column() |>
  mutate(agedays = age * 365.25) -> data_11_25
library(Greg)

timeSplitter(data_11_25, by = 365.25,
             time_var = "survive", event_var = "chd",
             time_related_vars = "agedays") -> data_11_25split
```

We import the data and add two new columns. One, the rownumber, to serve as an id and another to represent the age in days. This step is required to bring the survival time and age, the variables we want to modify for each interval, to the same unit of measurement.

We use the function `timeSplitter` from `Greg` to prepare the data. The first argument to `timeSplitter` is the dataframe we wish to modify. The argument `by` tells the width of the intervals into which the data will be split. Here, we want each record to be split into one year periods. As the survival time is stored in days, we use 365.25 to represent an year. The `time_var` tells which variable represents the survival time. The `event_var` tells which variable represents the outcome – censored or death. The `time_related_vars` is used to indicate the variables that need to be calculated for each interval. Here, we supply the age represented in days. This variable will be incremented by 365.25 after each interval starting from the original value. Note that though we supply a `data` argument, we cannot refer directly to the data frame's columns by their name; they need to be enclosed in quotes.

We can confirm that the number of rows in the newly created data frame agrees with that given in the textbook.

```
nrow(data_11_25split)
```

```
[1] 35126
```

We may inspect a set of the split records against the original.

```
filter(data_11_25, rowname == 10)
```

```
# A tibble: 1 x 6
  rowname   age house   chd survive agedays
  <chr>   <dbl> <fct> <dbl>   <dbl>   <dbl>
1 10       46.7 2         1    1635  17068.
```

```
filter(data_11_25split, rowname == 10)
```

```
 agedays rowname   age house chd Start_time Stop_time
```

```
17068       10 46.73    2   0     0.00    365.25
17433       10 46.73    2   0   365.25    730.50
17799       10 46.73    2   0   730.50   1095.75
18164       10 46.73    2   0  1095.75   1461.00
18529       10 46.73    2   1  1461.00   1635.00
```

Note how the *agedays* column increment by 365.25 in each row, while *age*, which we left untouched, remains the same. Note the new columns `Start_time` and `Stop_time` which indicate the interval during the observation period for which the information in the row is valid. Note how the value of *chd* is `0` except for the last period.

Now that we have verified that the data frame has been modified correctly, we may perform a logistic regression on it and confirm that it agrees with the textbook value. As we do not intend to use the model anywhere else, we will specify the model directly within `tidy` asking it to provide us with exponentiated values including the confidence interval.

```
tidy(glm(chd ~ agedays + house,family = binomial(),data = data_11_25split),
     exponentiate = TRUE,
     conf.int = TRUE) |>
  filter(term == "house2") |>
  select(c(term, estimate, conf.low, conf.high))
```

```
# A tibble: 1 x 4
  term   estimate conf.low conf.high
  <chr>     <dbl>    <dbl>     <dbl>
1 house2     1.42     1.09      1.86
```

Note that we may use the same modified dataframe for pooled Cox regression by providing the columns that mark the start and end of the intervals to `Surv`.

11.8 Recap

11.8.1 Concepts

- counting process format

11.8.2 Commands introduced in this chapter

- dplyr::lead
- survival::survreg
- survival::coxph
- stringr::str_sub
- stringr::str_starts
- graphics::text
- base::as.numeric
- SurvRegCensCov::ConvertWeibull
- survival::survSplit
- survival::finegray
- base::expand.grid
- stats::offset
- graphics::title
- dplyr::transmute
- Greg::timeSplitter

12

Meta-analysis

We will use the package `metafor` in this chapter on meta-analysis. Most of the functions expect a data frame and use the column names directly or by means of a formula. The `tidyverse` packages we use include `readr`, `dplyr`, `ggplot2` and stringr. First, we will try to recreate the graph in figure 12.1 using `ggplot`.

Figure 12.1 (page 566)

```
data.frame(author = c("Alderson", "Benhamou", "Chan"," Chan",
                      "de Stefani (a)","de Stefani (b)","Hu"," Hu",
                      "Ives", "Maclennan", " Maclennan","Engeland",
                      " Engeland","Hawthorne","Reid"),
           sex = c("Male","Male","Female","Male","Male",
                   "Male","Female","Male","Female","Female",
                   "Male","Female","Male","Male","Male"),
           rr = c(1.46,1.28,0.47,1.40,1.67,2.00,2.89,1.27,
                  2.39,0.69,1.64,1.56,1.06,1.94,1.67),
           lci = c(1.11,0.98,0.22,0.80,1.22,1.28,0.79,0.74,
                   1.11,0.31,0.96,0.91,0.79,0.95,1.11),
           uci = c(1.91,1.67,1.01,2.46,2.30,3.12,10.5,2.19,
                   5.13,1.52,2.79,2.69,1.43,3.97,2.51)
           ) -> tbl_12_1

ggplot(tbl_12_1, aes(y = author)) +
  geom_pointrange(aes(x = rr, xmin = lci, xmax = uci)) +
  geom_text(aes(x = 0,label = sex),hjust = "left",size = 3) +
  geom_text(aes(x = 25,
                label = paste0(rr, "(", lci, ", ", uci, ")"),
                hjust = "right")) +
  geom_vline(aes(xintercept = 1), lty = 2) +
  scale_x_log10(limits = c(0.1,25),breaks = c(0.2,0.5,1:4)) +
  xlab("Relative risk") +
  scale_y_discrete( limits = rev(tbl_12_1$author)) +
  ylab(NULL) +
  theme(panel.grid = element_blank(),
        panel.background = element_blank(),
        axis.ticks = element_blank())
```

We use the values given in fig 12.1 to prepare the dataframe. The rows are sorted according to the order in figure 12.1. We use `geom_pointrange` to plot the estimated relative risk and its confidence interval. We use `geom_text` to plot the sex group studied in each study as well

DOI: 10.1201/9781003589563-12

the values of relative risk and its confidence interval. The `hjust` argument determines the horizontal alignment of the text relative to the specified coordinates. We use `geom_vline` to place a vertical line corresponding to relative risk of 1. We use `scale_x_log10` to tell `ggplot` that we need log scaled x axis. The `limits` argument decides the extend of the scale and the `breaks` argument determines the tick marks that will be placed. We use `rev` inside `scale_y_discrete` to order the y axis properly as the default behaviour of `ggplot` is to start from origin and go up the positive y axis. `ggplot` uses only unique values in *author* and thus overplots whenever the *author* is repeated. We have resorted to a trick to make each value of *author* unique – we add a space before the duplicated value. `xlab` and `ylab` determine the axis labels. Finally, we use `theme` to unset the panel grid, panel background and axis ticks. While we don't have the plot exactly as in figure 12.1, we will move ahead and see more of forest plots later.

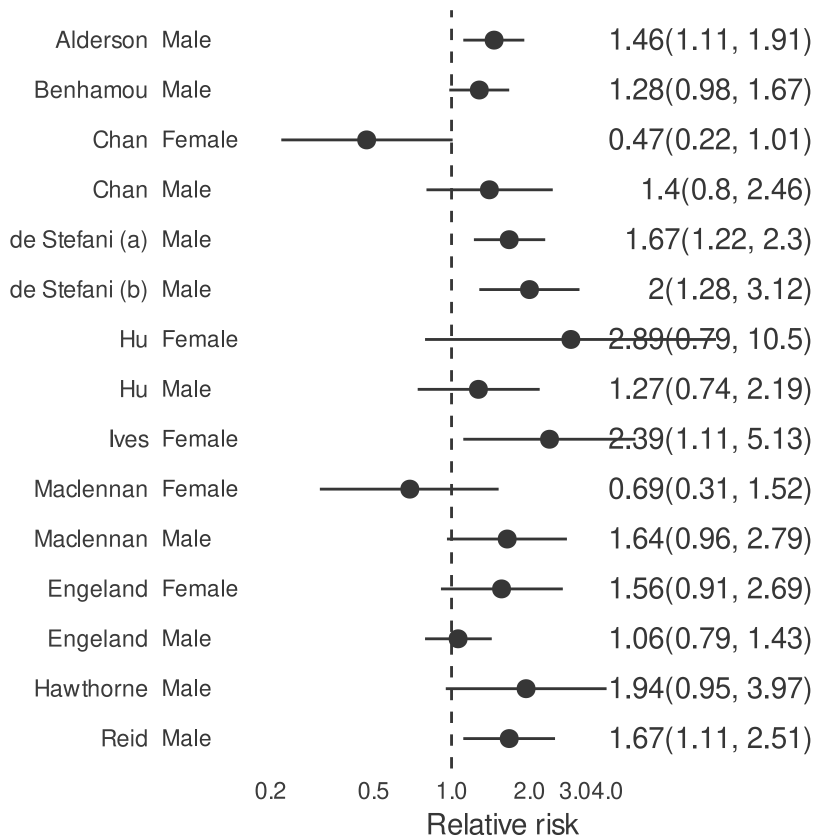


FIGURE 12.1
Replication of figure 12.1

12.1 A general approach to pooling

Example 12.3 (page 577)

Reworking example 12.3 requires only `mutate` to create the new columns.

```
mutate(tbl_12_1,
       logrr = log(rr),
       stderr = (((logrr - log(lci)) / 1.96 ) +
                  ((log(uci) -logrr) / 1.96))/2) -> tbl_12_2
tbl_12_2[tbl_12_2$author == "Alderson",]
```

In a single step we calculate the standard error by averaging the value calculated from both the confidence limits.

```
    author  sex   rr  lci  uci   logrr  stderr
1 Alderson Male 1.46 1.11 1.91 0.37844 0.13845
```

Example 12.4 (page 578)

For example 12.4, we need few more columns.

```
mutate(tbl_12_2,
       weightfe = 1/ stderr^2,
       wfelogrr = weightfe * logrr) -> tbl_12_2

sum(tbl_12_2$wfelogrr) / sum (tbl_12_2$weightfe) -> fe_12_2
1/ sqrt(sum(tbl_12_2$weightfe)) -> fesr_12_2
fe_12_2
fesr_12_2
```

We add the columns with weight for fixed effects and for the product of the log of RR and the weight. From these columns, we calculate the pooled log relative risk and its standard error.

```
[1] 0.34064
[1] 0.056666
```

We can test the null hypothesis of pooled log risk using `pnorm`

```
pnorm(fe_12_2/fesr_12_2, lower = FALSE)
```

```
[1] 9.201e-10
```

We can calculate D,Q, and I^2 using the formula given in the textbook.

```
mean(tbl_12_2$weightfe) -> meanwfe_12_2
var(tbl_12_2$weightfe) -> varwfe_12_2
length(tbl_12_2$author) -> k
(k - 1) * ((k * meanwfe_12_2 ^ 2) - varwfe_12_2 ) / (k * meanwfe_12_2) -> D
mutate(tbl_12_2,
       q = (logrr - fe_12_2)^ 2 * weightfe) -> tbl_12_2
sum(tbl_12_2$q) -> Q
100 * (Q - (k -1)) / Q -> I
```

```
sqrt(Q / (k -1)) -> H
(log(Q) - log(k-1))/ ( 2 * (sqrt(2*Q)  - sqrt(2 * k -3))) -> selogH
exp(log(H) - (1.96 * selogH)) -> L
exp(log(H) + (1.96 * selogH)) -> U
```

We confirm that the upper confidence limit of I^2 is as given in our example.

```
100 - (100 / U ^2 )
```

```
[1] 67.922
```

Table 12.2 (page 578)

To calculate the weights for random effects model, we need to calculate the between-studies variance estimate.

```
(Q - k+1) / D -> tsq
mutate(tbl_12_2,
       weightre = 1/ (( 1/ weightfe) + tsq)) -> tbl_12_2
select(tbl_12_2, c(author, sex, logrr,stderr, weightfe, weightre)) |>
  mutate(author = str_trim(author)) |>
  arrange(author, sex)
```

We add a new column of weights to the dataframe. We now have all the columns of table 12.2. We use `str_trim` to remove the extra space in front of the repeated author's names.

TABLE 12.1
Replication of table 12.2

		Log relative risk		Weights	
Study author	Sex	Estimate	Standard error	Fixed effect	Random effects
Alderson	Male	0.378436	0.13845	52.1655	18.4606
Benhamou	Male	0.246860	0.13598	54.0847	18.6954
Chan	Female	−0.755023	0.38880	6.6154	5.3717
Chan	Male	0.336472	0.28656	12.1780	8.5386
Engeland	Female	0.444686	0.27649	13.0807	8.9728
Engeland	Male	0.058269	0.15138	43.6397	17.2668
Hawthorne	Male	0.662688	0.36481	7.5139	5.9493
Hu	Female	1.061257	0.65997	2.2959	2.1251
Hu	Male	0.239017	0.27679	13.0529	8.9597
Ives	Female	0.871293	0.39050	6.5579	5.3337
Maclennan	Female	−0.371064	0.40559	6.0791	5.0126
Maclennan	Male	0.494696	0.27216	13.5006	9.1684
Reid	Male	0.512824	0.20814	23.0820	12.7676
de Stefani (a)	Male	0.512824	0.16175	38.2220	16.3499
de Stefani (b)	Male	0.693147	0.22729	19.3572	11.5394

We add new columns to the dataframe to facilitate the calculation of the pooled estimate according to the random effect model.

```
mutate(tbl_12_2,
       wrelogrr = weightre * logrr,
       serenum = weightre ^ 2 * stderr ^ 2) -> tbl_12_2
sum(tbl_12_2$wrelogrr) / sum (tbl_12_2$weightre) -> re_12_2
1/ sqrt(sum(tbl_12_2$weightre)) -> resr_12_2
```

We confirm that the result agree with our example.

```
exp(re_12_2)
exp(re_12_2 + c(-1.96,1.96) * resr_12_2)
```

```
[1] 1.4184
[1] 1.2115 1.6607
```

We will see in the next example that we needn't calculate the pooled effects by hand. We followed manual calculation to reproduce the table showing the weights for fixed effects and random effects models.

Example 12.5 (page 580)

We use `rma` from `metafor` to calculate the pooled effects. Remember to install the package using `install.packages` as discussed in chapter 1.

```
data.frame(study = c("Busselton", "CISCH", "Civil Service",
                     "Fletcher Challenge","Ohasama",
                     "Seven Cities","Singapore NHS",
                     "Singapore Heart", "Tanno Soubetsu"),
           DMyCBVy = c(17,1,1,7,8,9,20,22,3),
           DMyCBVn = c(85,24,52,251,216,116,300,195,104),
           DMnCBVy = c(454,37,8,77,46,284,24,53,30),
           DMnCBVn = c(4718,1736,2102,9976,1970,10264,2987,2072,1677),
           est = c(1.351,0.770,1.611,1.334,0.679,1.468,2.079,1.852,0.509),
           stderr = c(0.248,1.013,1.061,0.395,0.390,0.340,0.310,0.255,0.606)
           ) -> tbl_12_3
library(metafor)
rma(yi = est, sei = stderr,
    data = tbl_12_3,
    method = "FE",
    slab = study) -> fe_12_3
```

The function `rma` accepts many arguments. When the estimate from the study is provided directly, we use the argument `yi`. The standard error is supplied via `sei`. The `data` argument specifies the data frame in which to find the variables specified. The `method` argument specifies whether we want a fixed effect model or a variable effect model. We have specified `FE` to mean that we need the fixed effect model. The `slab` argument is used to specify the column that carries the study labels.

We may print the returned object, use `summary` or select only those components that we need. We can confirm that the pooled estimate, its standard error and the I^2 statistic all agree with the values given in the textbook.

```
fe_12_3$beta
fe_12_3$se
fe_12_3$I2
```

```
         [,1]
intrcpt 1.4849
[1] 0.12103
[1] 41.006
```

To fit a random effects model, we need to specify one of the various estimators of heterogeneity as the `method`. Here, we use `DL`. We can confirm the agreement of the calculated values with the textbook values.

```
rma(yi = est,
    sei = stderr,
    data = tbl_12_3,
    method = "DL",
    slab = study) -> re_12_3
re_12_3$beta
re_12_3$se
re_12_3$tau2
```

```
         [,1]
intrcpt 1.4247
[1] 0.17133
[1] 0.097772
```

The weights assigned to each of the studies can be obtained using `weights`.

```
weights(fe_12_3)
```

```
        Busselton             CISCH     Civil Service Fletcher Challenge
          23.8180            1.4275            1.3013             9.3889
          Ohasama      Seven Cities     Singapore NHS    Singapore Heart
           9.6312           12.6722           15.2435            22.5283
   Tanno Soubetsu
           3.9890
```

However, this gives us the percentage contribution of each study, corresponding to the second and fourth column of table 12.4. We may specify the argument `type="matrix"` to get the values corresponding to that in column one and three of table 12.4. However, the values are in the diagonal elements of the matrix returned by the function. We use `diag` to get the diagonal elements.

```
diag(weights(fe_12_3, type="matrix"))
```

```
      Busselton             CISCH      Civil Service Fletcher Challenge
       16.25911           0.97450            0.88832            6.40923
        Ohasama      Seven Cities      Singapore NHS    Singapore Heart
        6.57462           8.65052           10.40583           15.37870
 Tanno Soubetsu
        2.72304
```

Table 12.4 (page 581)

We can now build table 12.4.

```
mutate(tbl_12_3,
       WeightsFE = diag(weights(fe_12_3, type="matrix")),
       PercentageFE  = weights(fe_12_3),
       WeightsRE = diag(weights(re_12_3, type="matrix")),
       PercentageRE  = weights(re_12_3)) -> tbl_12_3
```

We still don't have the heterogeneity measure, which we will have to calculate by hand.

```
fe_12_3$beta["intrcpt",] -> fepm
mutate(tbl_12_3,
       q = WeightsFE * (est - fepm) ^ 2,
       Percentageq = 100 * q / sum(q)) -> tbl_12_3
select(tbl_12_3, study, WeightsFE, PercentageFE,
       WeightsRE, PercentageRE, q, Percentageq)
```

TABLE 12.2
Replication of table 12.4

Study name	Weights					
	Fixed effects	%	Random effects	%	Components of heterogeneity	%
Busselton	16.25911	23.8%	6.27842	18.4%	0.2912976	2.1%
CISCH	0.97450	1.4%	0.88973	2.6%	0.4979797	3.7%
Civil Service	0.88832	1.3%	0.81733	2.4%	0.0141364	0.1%
Fletcher Challenge	6.40923	9.4%	3.94016	11.6%	0.1458477	1.1%
Ohasama	6.57462	9.6%	4.00205	11.7%	4.2695272	31.5%
Seven Cities	8.65052	12.7%	4.68666	13.8%	0.0024562	0.0%
Singapore NHS	10.40583	15.2%	5.15805	15.1%	3.6733984	27.1%
Singapore Heart	15.37870	22.5%	6.14263	18.0%	2.0730293	15.3%
Tanno Soubetsu	2.72304	4.0%	2.15050	6.3%	2.5931125	19.1%

12.2 Investigating heterogeneity

Figure 12.2 (page 585)

We use the function `forest` to plot a forest plot of figure 12.2.

```
forest(fe_12_3,
       atransf = exp,
       at = log(c(0.5,1,2,4,8,16)),
       order = "prec",
       header = TRUE,
       annotate = FALSE,
       ylim = c(-2,12))
addpoly(re_12_3)
```

The first argument that we provide is the object returned by `rma`. We ask for transforming the x-axis labels using `exp` by specifying `atransf = exp`. The argument `at` determines the placement of tick marks on the x-axis. We use `log` as we have specified `atransf=exp`. The argument `order` determines the order in which the studies appear. Our choice `prec` arranges them according to their variance. The arguments `header` determines whether a header should be drawn. The argument `annotate` determines whether the numerical value of the estimate and confidence intervals are shown or not. We use `addpoly` to add the pooled measure calculated according to the random effects model. It is in anticipation of this that we increased the y limits of the graph drawn using `forest` by specifying the `ylim` argument. The argument we supply to `addpoly` is an object returned by `rma`.

Figure 12.3 (page 587)

We don't have a ready-made function to prepare the influence plot given in figure 12.3.

```
ggplot(tbl_12_3, aes(y = ifelse(est > fepm,sqrt(q), -1 * sqrt(q)))) +
  geom_point(aes(x = sqrt(WeightsFE))) +
  geom_text(aes(label = study,x = sqrt(WeightsFE) - 0.05),
            vjust = "top",
            hjust = "right") +
  geom_hline(yintercept = 0) +
  geom_hline(yintercept = c(2,-2), lty = 2) +
  xlim(c(0,5)) +
  xlab("Square root of weight") +
  ylab("Square root of heterogeneity measure") +
  theme_minimal() +
  theme(panel.grid = element_blank())
```

We use `ggplot` to draw the graph. We specify the `y` aesthetics inside `ggplot` as it is common to the two layers that we plan to include. The value of `y` is calculated from the existing columns of *tbl_12_3* according to the directions in our text. For drawing the points, we use `geom_point` and supply it with the square root of the weights as mentioned in our text.

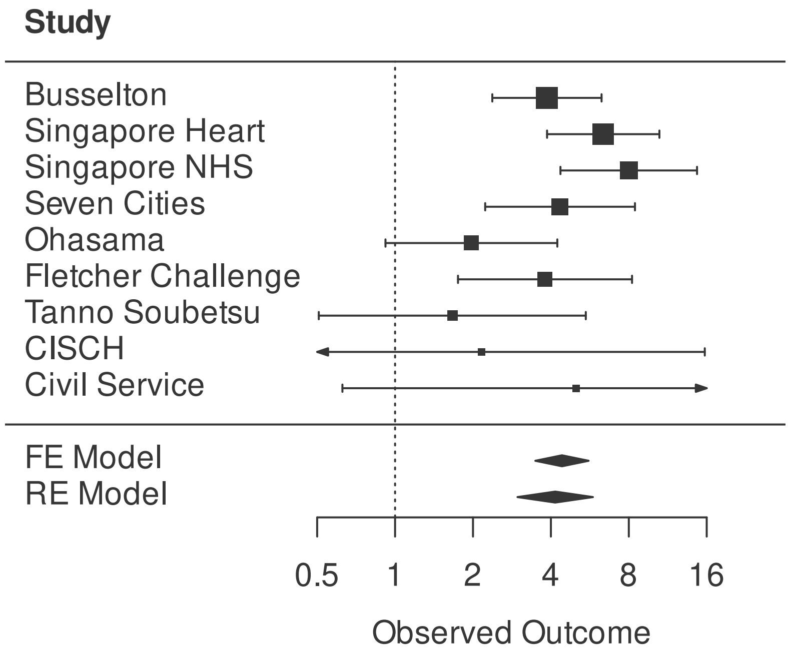


FIGURE 12.2
Replication of figure 12.2

For drawing the labels, we reduce 0.05 from these values so that there is some separation of text from the points. We use the arguments `vjust` and `hjust` to adjust the alignment of the labels relative to the coordinates we specified. We use `geom_hline` to add the horizontal lines of reference. We use it twice because we use two different line types. We set the x-axis limits, x label and y label by hand. Finally, we use the `theme_minimal` and remove the grid lines.

Though we don't have a ready-made function to prepare the influence plot, `metafor` provides `radial` to draw radial plots.

12.2.1 Meta regression

Example 12.8 and 12.9 (page 588 and 590)

Fitting a meta regression model is accomplished through `rma`.

```
c(46.3,53.9,44.2,44.4,59.5,53.8,38.8,40.1,50.8) -> tbl_12_3$age
rma(est ~ age,
    sei = stderr,
    data = tbl_12_3,
    method = "FE",
    slab = study) -> mr_12_3
mr_12_3
```

First, we create a new column in *tbl_12_3* to hold the mean age. Then, we call `rma`. The difference from our previous example is that instead of `yi` we supply a formula. The left-hand

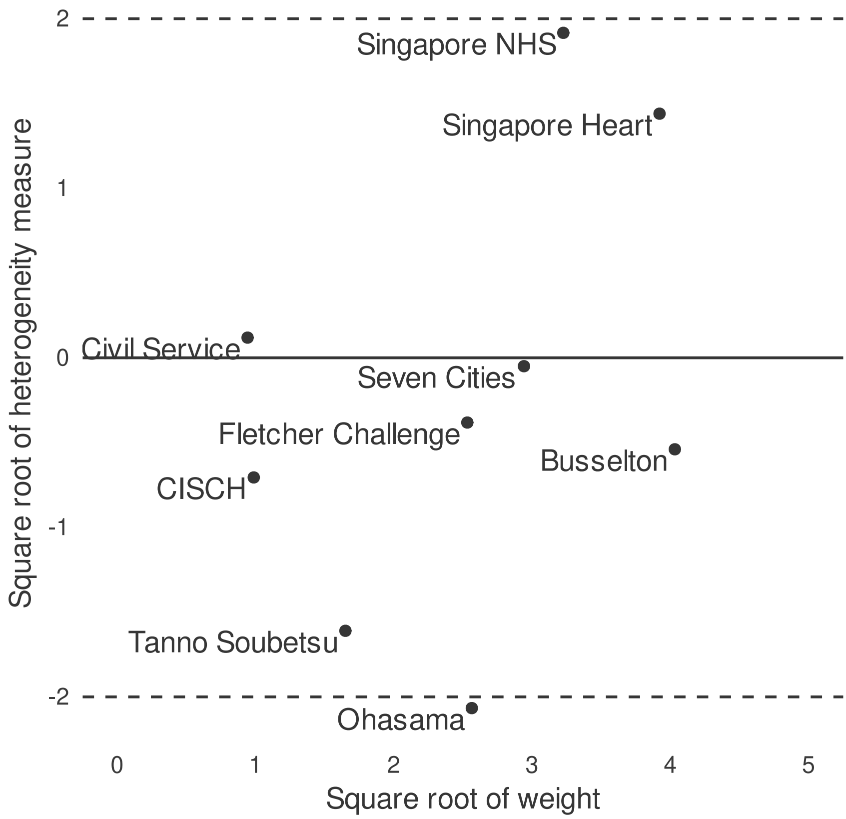


FIGURE 12.3
Replication of figure 12.3

side of the formula contains the same column of effect measure. The right-hand side contains the variable `age`, which we think explains part of the heterogeneity of the studies. Printing the model object provides all the information in example 12.9.

```
Fixed-Effects with Moderators Model (k = 9)

I^2 (residual heterogeneity / unaccounted variability): 0.00%
H^2 (unaccounted variability / sampling variability):   0.57
R^2 (amount of heterogeneity accounted for):            66.58%

Test for Residual Heterogeneity:
QE(df = 7) = 3.9652, p-val = 0.7838

Test of Moderators (coefficient 2):
QM(df = 1) = 9.5956, p-val = 0.0020

Model Results:

         estimate      se     zval    pval    ci.lb    ci.ub
intrcpt    4.1150  0.8577   4.7980  <.0001   2.4340   5.7960  ***
age       -0.0571  0.0184  -3.0977  0.0020  -0.0932  -0.0210   **
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Figure 12.4 (page 589)

We can use `regplot` to draw the graph in figure 12.4

```
regplot(mr_12_3, ci = FALSE,
        bg = "white",
        ylab = "Log hazard ratio",
        xlab = "Mean age (years)")
```

The main argument for `regplot` is the model object returned by `rma`. We specify that we don't want a confidence interval band around the regression line by specifying `ci = FALSE`. The argument `bg = "white"` determines the shading of the bubbles.

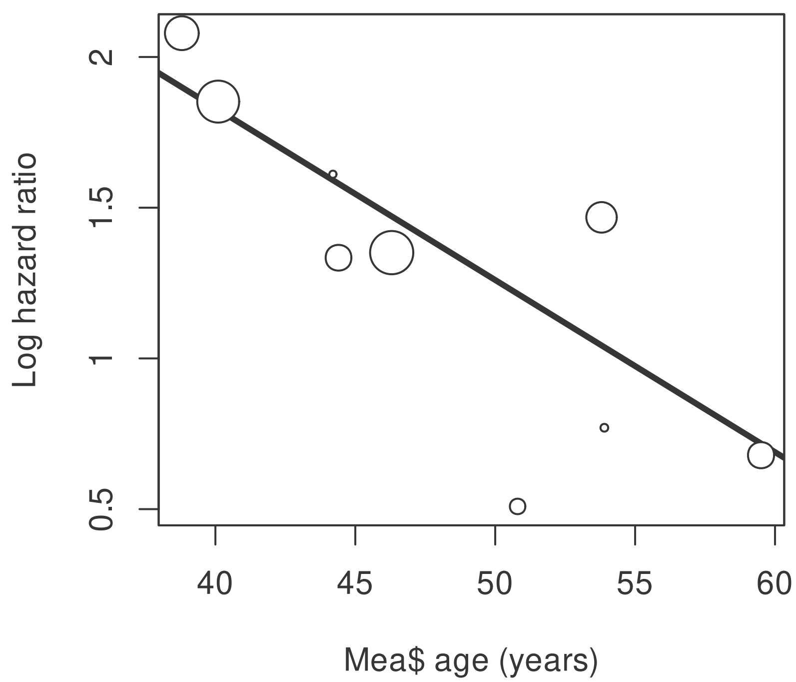


FIGURE 12.4
Replication of figure 12.4

12.3 Pooling tabular data

Example 12.10 (page 591)

We can use `rma` to calculate pooled estimates from tabular data too.

```
rma(ai = DMyCBVy, bi = DMyCBVn,
    ci = DMnCBVy, di = DMnCBVn,
    data = tbl_12_3, measure = "RR",
    method = "FE", slab = study) -> ivfe_12_10

rma(ai = DMyCBVy, bi = DMyCBVn,
    ci = DMnCBVy, di = DMnCBVn,
    data = tbl_12_3, measure = "RR",
    method = "DL", slab = study) -> ivre_12_10
```

The difference from our previous examples is that instead of log RR, we are specifying the value of the two-by-two cells from which the RR is calculated. The columns specified as values for the arguments `ai`,`bi`,`ci` and `di` are taken as containing the values in each of the cells in a two-by-two table, the names being the standard names assigned to these cells. We need to specify `measure` as well. We specify it as `"RR"` here because we want `rma` to calculate the log RR. The arguments `method` and `slab` have the same use as in our previous examples.

The coefficient and its standard error returned by the function are in the log scale. We can exponentiate them to confirm that they agree with the values in the text.

```
exp(c(ivfe_12_10$beta,ivfe_12_10$ci.lb,ivfe_12_10$ci.ub))
exp(c(ivre_12_10$beta,ivre_12_10$ci.lb,ivre_12_10$ci.ub))
```

```
[1] 3.0285 2.4171 3.7945
[1] 2.9669 1.9752 4.4567
```

Example 12.11 (page 592)

We can use `rma.mh` function to apply MH meta analysis. Here, we rework example 12.11.

```
rma.mh(ai = DMyCBVy, bi = DMyCBVn,
       ci = DMnCBVy, di = DMnCBVn,
       data = tbl_12_3, measure = "RR",
       slab = study) -> mh_12_11
mh_12_11
```

Except for the function name, using `rma.mh` is no different from using `rma` for tabular data. The argument `method` is not required.

```
Equal-Effects Model (k = 9)

I^2 (total heterogeneity / total variability):  62.05%
H^2 (total variability / sampling variability): 2.64
```

```
Test for Heterogeneity:
Q(df = 8) = 21.0829, p-val = 0.0069

Model Results (log scale):

estimate      se    zval    pval   ci.lb   ci.ub
  1.0679  0.1120  9.5337  <.0001  0.8483  1.2874

Model Results (RR scale):

estimate   ci.lb   ci.ub
  2.9092  2.3357  3.6233
```

The function `rma.peto` can apply the Peto method. The `rma` function also permits adding a non-negative number to cells with the value zero through its `add` argument. Further control is given by `to` which allows one to control whether this value should be added to all cells or only those with zero value.

12.4 Publication bias

Figure 12.6 (page 598)

To rework the example 12.12, we import the data. Note that the data file is named 12.13 though it is for example 12.12.

```
read_table("K11828 supplements/Datasets/Example 12.13.dat",
           col_names = c("study","adjust","rr","lci","uci")) -> data_12_12
mutate(data_12_12,
       logrr = log(rr),
       stderr =  (log(uci) - log(lci))/(2*1.96)) -> data_12_12
rma(yi = logrr,
    sei = stderr,
    data = data_12_12,
    method = "DL",
    slab = study) -> re_12_12
paste0("Overall (I-squared = ",
       round(re_12_12$I2,2),
       "%)") -> relabel
```

After importing the data, we `mutate` the data frame to introduce two columns corresponding to the log of RR and the standard error calculated from the confidence interval. We supply these values to `rma`.

```
forest(re_12_12,
       header = c("Study author", "Relative risk"),
       showweights = TRUE,
```

```
       order ="obs",
       ilab = adjust,
       ilab.xpos = -4.25,
       ilab.pos = 4,
       atransf = exp,
       at = log(c(0.25,0.5,1,2,4,6,8)),
       mlab = relabel,
       cex = 0.75,
       col = "white")
text(x = c(-4.25,3),
     y = 30,
     labels = c("Adjustments", "Wts"),
     cex = 0.75,
     font = 2,
     pos = 4)
```

The returned object is saved and `forest` called with it. We also build a label for the pooled estimate corresponding to that in figure 12.6, which we supply to `mlab` argument of `forest`. The `header` argument determines the heading labels for the column carrying the study names and the column carrying the numerical value of confidence interval. We ask for a new column showing the weights through `showweights` argument. We use the `order="obs"` argument to get the studies sorted according to the relative risk as in figure 12.6. The `ilab` argument allows us to include additional columns in the graph. Here, we want the code for adjustments shown. So, we specify the value of `ilab` as *adjust*. The x axis position where these additional rows will go is determined by `ilab.xpos`. The `ilab.pos` determines the alignment of the label at the specified coordinates. The `cex` determines the scaling factor for the font size. We specify this value so that we can reuse the value to have consistent looks when we add more headings in the next step. The `col` argument decides the colour of the fill of the pooled estimate. Note that `forest` doesn't have an option to determine the color of the boxes around the point estimate of individual studies. The `col` argument determines the colour used to plot the observed outcomes. It affects the line, the dot and the box together. Finally, we use `text` to add headings to the weight and adjustments columns. The `x` determines the x-axis positions and `y` the y-axis positions of the `labels`. The `pos` determines the relation of the text to the coordinate specified using `x` and `y`; here we specify `4` to mean right. The `font = 2` determines that the labels will be printed in bold, `cex` determines the proportion of font size displayed.

Figure 12.7 (page 599)

A funnel plot is drawn using `funnel`. Here, we reproduce figure 12.7.

```
funnel(re_12_12,
       atransf = exp,
       at = log(c(0.25,0.5,1,2,4,6,8)))
```

The first argument we supply to `funnel` is the object returned by `rma`. We use `atransf` to transform the x-axis labels using `exp`. The `at` argument determines the positioning of x-axis labels. We use `log` transformation of the required positions as we have specified `atransf`.

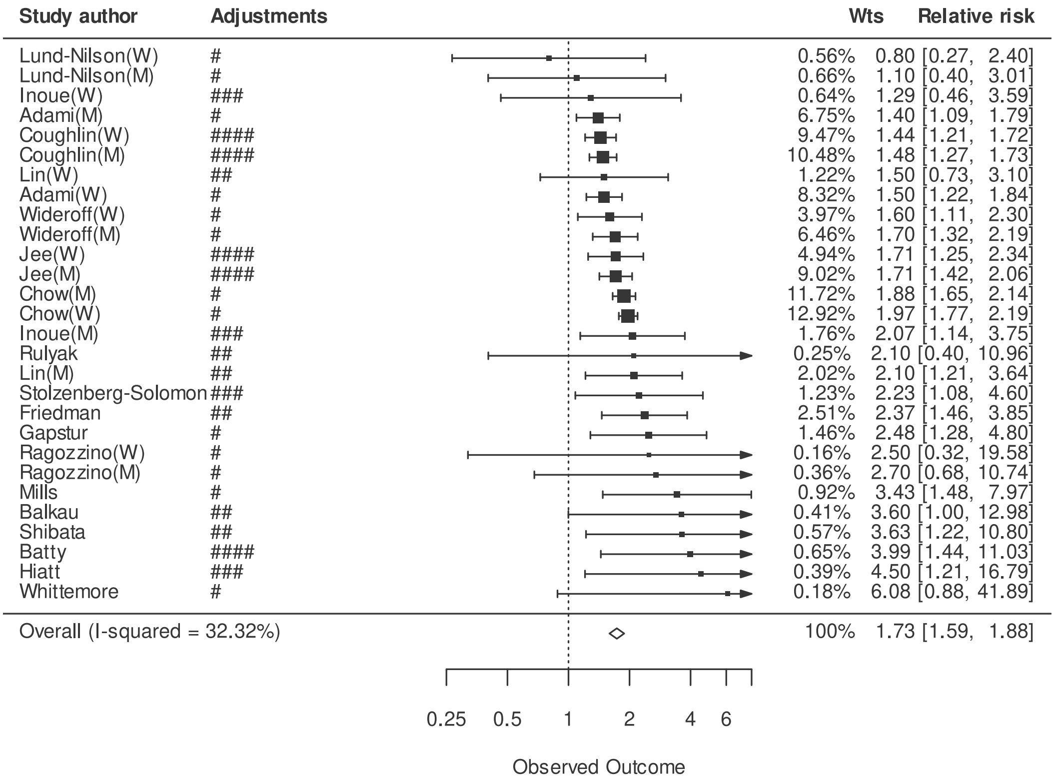


FIGURE 12.5
Replication of figure 12.6

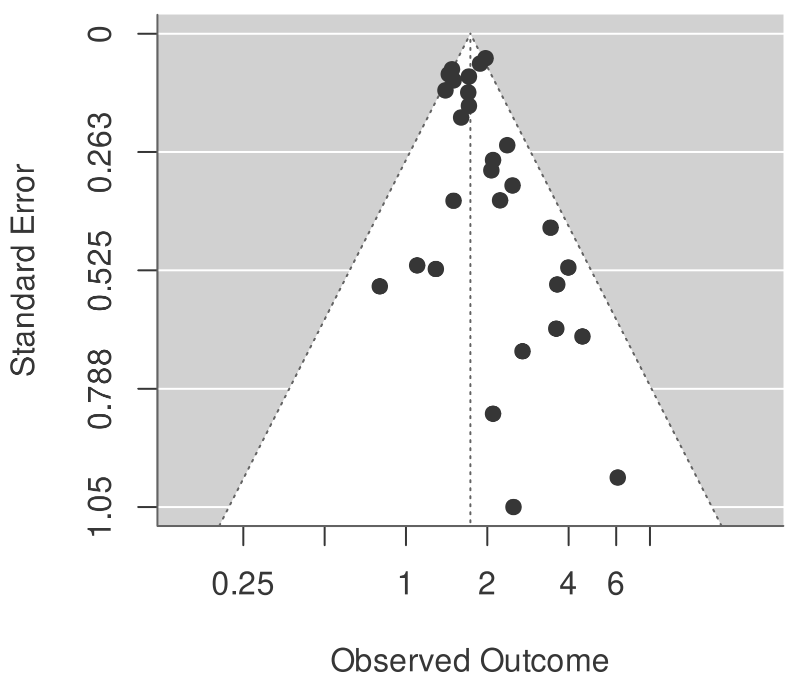


FIGURE 12.6
Replication of figure 12.7

Example 12.12 (page 597)

The function `trimfill` applies the trim and fill method to an object returned by `rma`.

```
trimfill(re_12_12) -> tf_12_12
funnel(tf_12_12,
       atransf = exp,
       at = log(c(0.25,0.5,1,2,4,6,8)))
```

The `trimfill` permits the use of different estimators. Here, we have gone with the default. We can reproduce figure 12.8 by passing the object returned by `trimfill` to `funnel`.

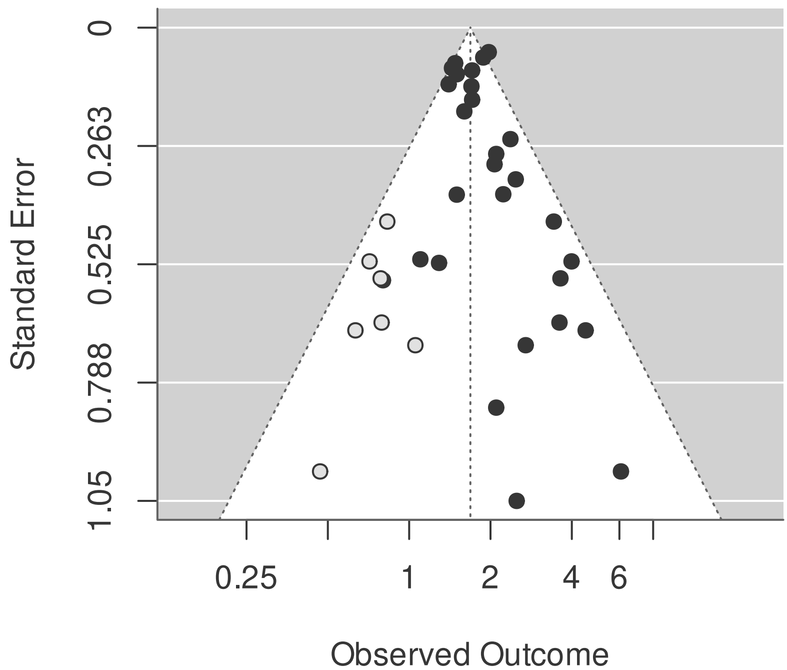


FIGURE 12.7
Replication of figure 12.8

12.5 Recap

12.5.1 Commands introduced in this chapter

- ggplot2::geom_text
- ggplot2::geom_vline
- ggplot2::scale_x_log10
- ggplot2::xlab
- ggplot2::scale_y_discrete
- base::rev
- ggplot2::ylab
- stats::pnorm
- stringr::str_trim
- metafor::rma
- metafor::weights
- base::diag
- metafor::forest
- metafor::addpoly
- metafor::regplot
- metafor::rma.mh
- metafor::rma.peto
- metafor::funnel
- metafor::trimfill

13

Risk scores and clinical decision rules

We will be using the `tidyverse` packages `readr`, `dplyr`, `ggplot2` and `purrr`. The other add on packages that we will use in this chapter include `broom`, `pROC`, `ROCit`, `effectsize`, rms,riskregression,epiR, `DescTools` and `Hmisc`.

13.1 Association and prognosis

Example 13.1 (page 609)

For example 13.1, we will import the data.

```
read_table("K11828 supplements/Datasets/Example 13.1.dat",
           col_names = c("sex", "fibrinogen", "cvd10","cvdfllw", "survive"),
           col_types = cols(sex = col_factor())) -> data_13_1
factor(data_13_1$sex,
       labels = c("women", "men")) -> data_13_1$sex
factor(data_13_1$cvd10,
       labels = c("no", "yes")) -> data_13_1$cvd10
factor(data_13_1$cvdfllw,
       labels = c("no", "yes")) -> data_13_1$cvdfllw
```

After importing, we modify the factor variables to assign appropriate labels. Note that the data doesn't include the columns for cholesterol, systolic blood pressure and smoking. Thus, we will not be able to replicate the examples when these data are needed.

Table 13.1 (page 609)

We will prepare the first three columns of table 13.1.

```
group_by(data_13_1, sex, cvdfllw) |>
  summarise(n = n(),
            mean =  round(mean(fibrinogen),2),
            median = round(median(fibrinogen),2)) |>
  arrange(desc(sex), cvdfllw)
```

We group the data by *sex* and *cvdfllw*, use `summary` to find the summary numbers and `arrange` them.

DOI: 10.1201/9781003589563-13

TABLE 13.1
Replication of table 13.1

Sex	CVD during follow-up?	n	Mean (g/l)	Median (g/l)
men	no	4875	2.71	2.60
men	yes	1634	2.87	2.74
women	no	5559	2.82	2.71
women	yes	992	3.04	2.92

Figure 13.3 (page 609)

To prepare the graph of figure 13.3, we need to create a column containing the info on tenth of fibrinogen.

```
filter(data_13_1,sex == "women") |>
  mutate(fib10 = cut(fibrinogen,
                     breaks =  quantile(fibrinogen,seq(0,1, by = .1)),
                     right = FALSE,
                     include.lowest = TRUE)) -> data_13_1f
```

We `filter` records of women and create a new column *fib10*, the value of which is determined by `cut`. Based on the `breaks` we supply, `cut` divides the range of values of the supplied numerical vector into intervals to make that many groups and assigns each value to its appropriate category. The `breaks` we supply is calculated using `quantile`, which in our case will return the values of *fibrinogen* corresponding to its deciles and the minimum and maximum. The argument `right` given to `cut` determines whether the right-hand side of the interval is closed or not. The `include.lowest` when `TRUE` includes the most extreme value into the extreme category though that side of the interval is open.

```
library(survival)
coxph(Surv(survive, cvdfllw == "yes") ~ fib10,
      data = data_13_1f,
      x = TRUE,
      y = TRUE,
      ties ="breslow") -> cphf_13_1
```

We use `coxph` to fit the Cox proportional hazards model on the data we prepared.

We use `tidy` to get a neat dataframe from the `coxph` model and use it to draw the graph in figure 13.3.

```
ggplot(bind_rows(c(estimate = 1, conf.low = 1,conf.high = 1),
                 tidy(cphf_13_1,exponentiate = TRUE,conf.int = TRUE))) +
  geom_hline(yintercept = 1, lty = 2) +
  geom_pointrange(aes(y = estimate,
                      ymin = conf.low,
```

```
                    ymax = conf.high,
                    x = 1:10)) +
  scale_y_log10(breaks = 1:5) +
  lims(x = as.character(1:10)) +
  ylab("Hazard ratio") +
  xlab("Fibrinogen tenths")
```

The data we supply to `ggplot` is made by `bind_rows` applied to the dataframe returned by `tidy` and a single row with the value 1 for estimate and its confidence interval corresponding to the reference group. We add a reference line at 1 using `geom_hline`. The hazard ratios and their confidence intervals are drawn using `geom_pointrange`. We use `scale_y_log10` to specify a log scale for the y axis. The `lims` is used to set the axis labels for the x-axis.

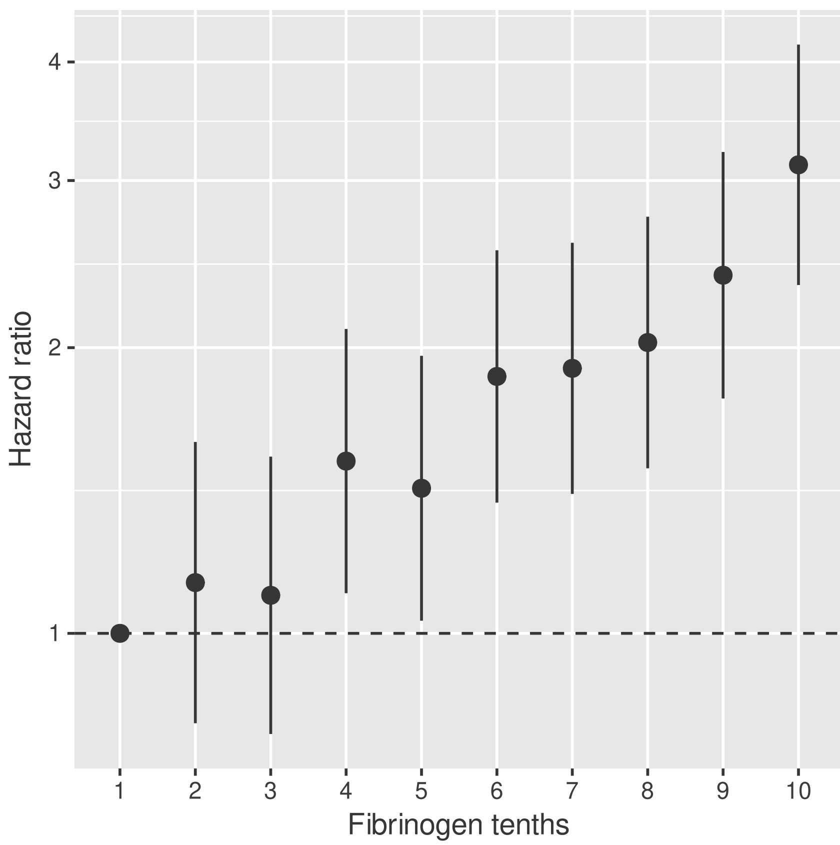


FIGURE 13.1
Replication of figure 13.3

Figure 13.6 (page 611)

To prepare the graph of figure 13.6, we use `geom_boxplot`.

```
ggplot(data_13_1) +
 geom_boxplot(aes(x = fibrinogen,
                  y = paste0(sex, "\n", cvdfllw)),
              outlier.shape = NA) +
 coord_cartesian(xlim = c(1,5)) +
 ylab(NULL) +
 xlab("Fibrinogen (g/l)")
```

Though there is need for only the `x` or `y` aesthetics, we supply both to get different boxplot for each of the categories. Because the category is determined by the value of both *sex* and *cvdfllw*, we use `paste0` to join them into one string. We use `outler.shape=NA` to specify that we don't want to include outliers in the graph. We may adjust the length of whiskers using the `coef` argument. However, it accepts a number that would be considered as multiple of the inter quartile range. As we cannot use it to extend the whiskers from 1 to 99 percentile, we have not modified the default value. We specify `xlim` inside `coord_cartesian` so that the x limits are as per our desire, but the data falling outside that range is not excluded from calculations.

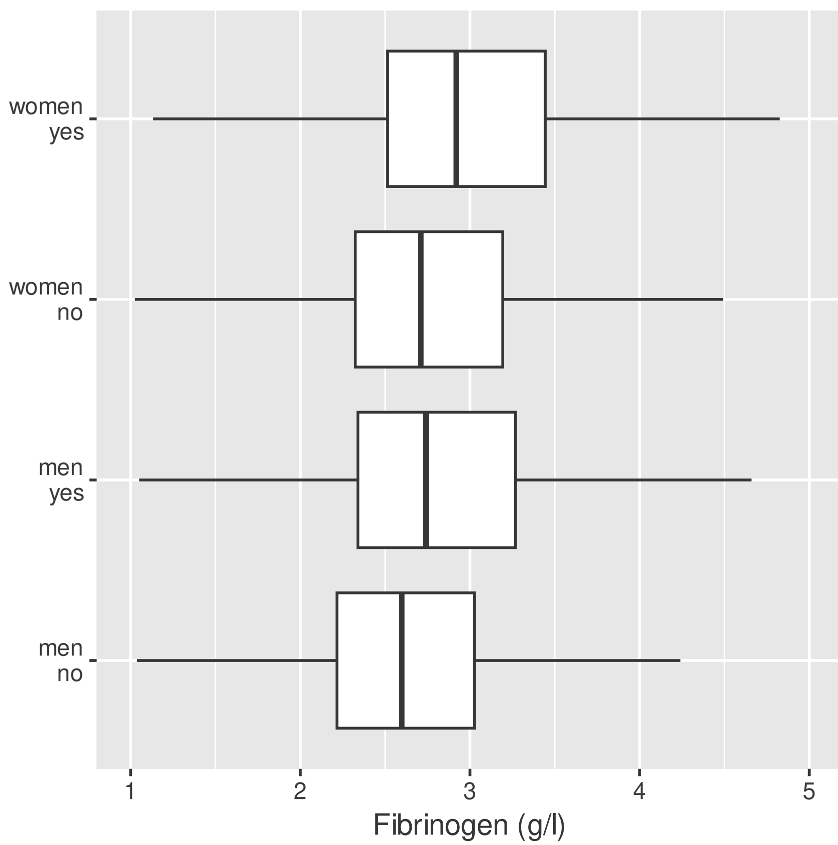


FIGURE 13.2
Replication of figure 13.6

Table 13.4 (page 613)

To prepare table 13.4, we will use `table`.

```
table(data_13_1f$cvd10,data_13_1f$fib10) |> addmargins()
```

We pass the result of `table` to `addmargins` to get the margin totals. As we don't specify a margin, row, column and grand totals are returned. Note that our result differs from that in the textbook. I was unable to find the reason for the discrepancy.

Figure 13.9 (page 615)

We need to prepare the graph of figure 13.9 by hand. Though `table` can give us the grouped data of table 13.4, it is not sufficient to calculate sensitivity and specificity. To calculate them, we need the numbers above or below a cutoff.

```
bind_cols(tenths = 0:10,
          maxval = quantile(data_13_1f$fibrinogen,seq(0,1, by = 0.1))
          ) -> data_13_2
data_13_2[1,"maxval"] <- data_13_2[1,"maxval"] * 0.99
data_13_2[11,"maxval"] <- data_13_2[11,"maxval"] * 1.01
mutate(data_13_2,
       pos = map_int(maxval,
                     function(x) nrow(filter(data_13_1f,
                                             cvd10 == "yes",
                                             fibrinogen >x))),
       neg = map_int(maxval,
                     function(x) nrow(filter(data_13_1f,
                                             cvd10 == "no",
                                             fibrinogen < x)))
       ) -> data_13_2

nrow(filter(data_13_1f,cvd10 == "yes")) -> cvdtot
nrow(filter(data_13_1f,cvd10 == "no")) -> nocvdtot
mutate(data_13_2,
       sns =  pos / cvdtot,
       spc = neg / nocvdtot) |>
  ggplot(aes(x = tenths)) +
  geom_point(aes(y = sns), shape = 21, colour = "#111111" ) +
  geom_line(aes(y = sns), colour = "#111111") +
  geom_point(aes(y = spc), shape =3, colour = "#004B73") +
  geom_line(aes( y = spc), colour = "#004B73") +
  labs(x = "Fibrinogen tenths",
       y = NULL)
```

We start by building a dataframe, with a column to indicate the cutoff points. We need one more than the number of cutoffs. So, we use the values zero to ten. We then add a column corresponding to the maximum value of each tenth calculated using `quantile`. We then move the extreme values a bit away so that the extreme values get included into the extreme categories. We use `mutate` again to add the number of future cvd cases that would fall in

TABLE 13.2
Replication of table 13.4

	Tenth(range; g/l)										
CVD	[0.441,2.01)	[2.01,2.25)	[2.25,2.43)	[2.43,2.59)	[2.59,2.74)	[2.74,2.91)	[2.91,3.11)	[3.11,3.38)	[3.38,3.86)	[3.86,11.3]	Sum
no	631	599	649	606	626	625	617	615	599	580	6147
yes	23	26	27	30	36	46	43	41	55	77	404
Sum	654	625	676	636	662	671	660	656	654	657	6551

the risk group if each of the cut offs were used. Similarly, we add a column to indicate the number of non-cases that would fall in the complementary category. We use `nrow` to find the total number of cases with and without cvd. We then calculate sensitivity and specificity according to the standard formula. We pass the dataframe to `ggplot` to plot the graph using `geom_point` and `geom_line`.

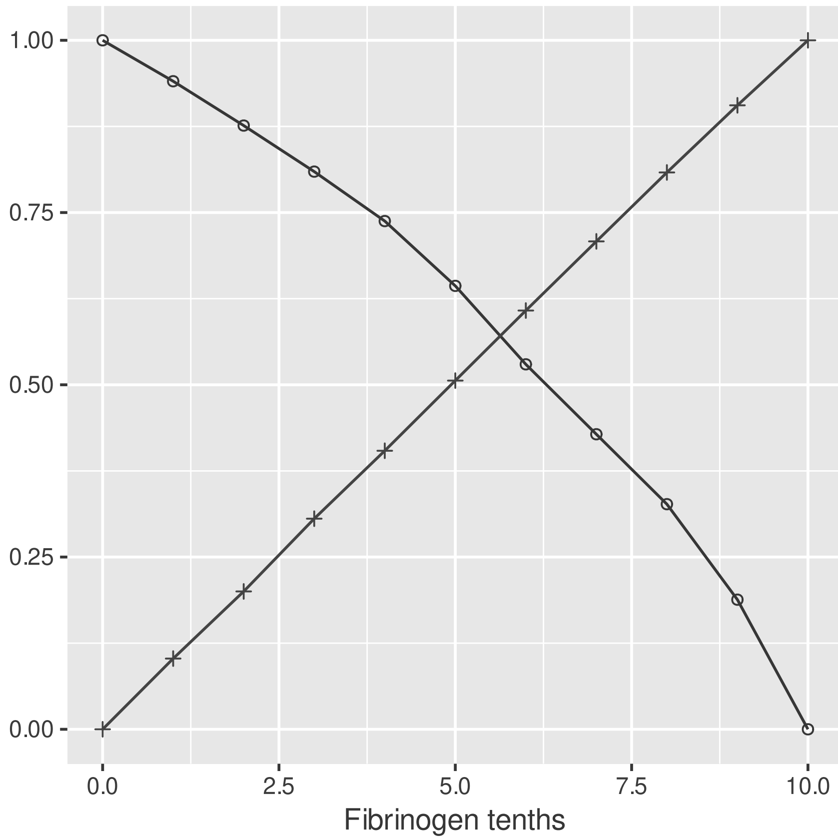


FIGURE 13.3
Replication of figure 13.9

Table 13.6 (page 617)

We will now try to reproduce table 13.6.

```
seq(0.1, 0.9, by = 0.1) -> values
expand.grid(Sensitivity = values, Specificity = values) |>
  mutate(odds = round(Sensitivity * Specificity /
                        ((1-Sensitivity) * (1- Specificity)),
                      1)) |>
  filter(odds >= 1) |>
  pivot_wider(names_from = Specificity,
              values_from = odds) |>
  arrange(Sensitivity)
```

First, we create a variable with the values we desire for sensitivity and specificity using `seq`. We then use `expand.grid` to prepare a dataframe containing all combinations of sensitivity and specificity values we specified. We use `mutate` to calculate odds ratio according to the formula given in the textbook. We then use `filter` to remove those rows with odds ratio less than 1. We rearrange the columns using `pivot_wider` and then sort according to the value of sensitivity.

TABLE 13.3
Replication of table 13.6

	Specificity								
Sensitivity	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.1									1.0
0.2								1.0	2.3
0.3							1.0	1.7	3.9
0.4						1.0	1.6	2.7	6.0
0.5					1.0	1.5	2.3	4.0	9.0
0.6				1.0	1.5	2.2	3.5	6.0	13.5
0.7			1.0	1.6	2.3	3.5	5.4	9.3	21.0
0.8		1.0	1.7	2.7	4.0	6.0	9.3	16.0	36.0
0.9	1	2.3	3.9	6.0	9.0	13.5	21.0	36.0	81.0

Figure 13.10 (page 618)

We will use `geom_function` to create the graph in figure 13.10.

```
ggplot() +
  geom_function(fun = dnorm,
                args = list(mean = 6.287, sd = .757),
                colour = "#004B73") +
  geom_function(fun = dnorm,
                args = list(mean = 6.680, sd = .757),
                lty = 2,
                colour = "#111111") +
  geom_text(aes(x =c(5,7.75), y = 0.3),
            label = c("No CHD", "CHD")) +
  ylab(NULL) +
  xlab("Serum total cholesterol (mmol/l)") +
  theme(panel.grid = element_blank(),
        axis.ticks.y = element_blank(),
        axis.text.y = element_blank()) +
  scale_x_continuous(limits = c(3,10),
                     breaks = c(3:6,(6.287 + 6.680) /2,7:10),
                     labels = c(3:6, "Mean", 7:10))
```

As we have no data to specify, `ggplot` doesn't have an argument. The function `geom_function` accepts a function that will calculate the y values for the range of x values. Here we provide `dnorm` which gives the density for the normal distribution. The arguments that `dnorm` require are given inside a `list` as the value of the argument `args` of `geom_function`. In our examples, we provide `mean` and `sd` as these are the arguments for `dnorm` that we want to change. We use the mean of cholesterol in the diseased and healthy group along with their common standard deviation as provided in the textbook. To add labels to mark the curves, we use `geom_text`. We use `scale_x_continuous` to limit the x-axis and to set appropriate labels. The argument `limits` to `scale_x_continuous` determines the limits. The argument `breaks` determine the location of axis tick marks and `labels` determine the labels at the tick mark. For `breaks` and `labels` we add a point to mark the average of the mean of the two distributions in addition to the whole numbers within the specified limits.

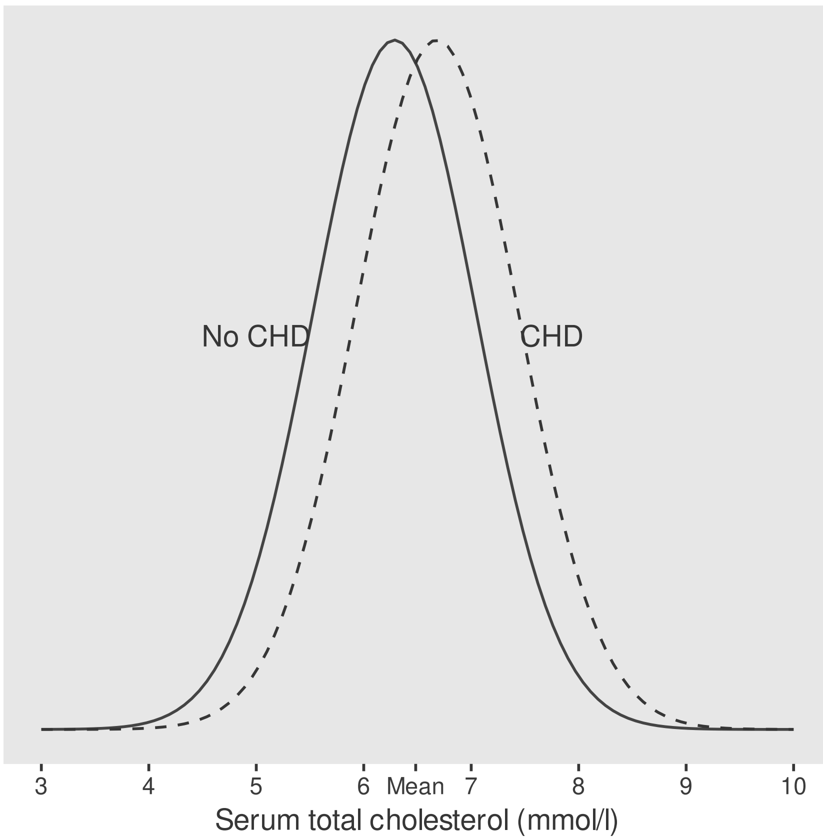


FIGURE 13.4
Replication of figure 13.10

13.2 Risk scores from statistical models

Example 13.4 (page 619)

To rework example 13.4, we need to fit the logistic model.

```
glm(cvd10 ~ fibrinogen,
    data = data_13_1f,
```

```
    family = binomial()) -> lg_13_4
bind_cols(data_13_1f,
          predict = predict(lg_13_4,type = "response")) -> data_13_4
ggplot(data_13_4,
       aes(x = fibrinogen,  y = predict)) +
  geom_point( alpha = 0.3) +
  geom_function(fun = function(x) (1 + exp(-(coef(lg_13_4)[1] +
                                               (coef(lg_13_4)[2] *
                                                  x))))^-1) +
  ylab("Expected risk") +
  xlab("Fibrinogen (g/l)") +
  scale_x_continuous(breaks = 0:10,
                     labels = as.character(0:10))
```

After fitting the logistic model, we use `predict` to calculate the predicted risk for the observed values. That we want risks is specified by using `type = "response"`. We column bind the predicted values with the original data frame. We then supply this data frame to `ggplot` and use `geom_point` and `geom_function` to plot the predicted risk for observed values and the fitted curve respectively. The `fun` argument supplied to `geom_function` is an anonymous function to predict the risk according to the formula derived from the fitted model.

Example 13.5 (page 620)

Before we can fit the logistic model of example 13.5, we need to modify the data.

```
read_table("K11828 supplements/Datasets/Example 10.12.DAT",
           col_names = c("age", "chol", "bmi", "sbp",
                         "smoke", "active", "chd", "survive"),
           col_types = cols( smoke = col_factor(levels = c("1","2","3")),
                             active =col_factor(levels = c("1","2","3")))
           ) -> data_13_15
mutate(data_13_15,
       chd5 =  ifelse(survive >= 1827, 0, chd)) -> data_13_15
glm(chd5 ~  chol + sbp + smoke ,
    data = data_13_15,
    family = binomial()) -> lg_13_15
lg_13_15
```

After importing the data, we `mutate` the data frame to create a new column to show the outcome at five years. We use `ifelse` to return `0` if the survival period is more than five years, otherwise the original outcome. We use this column as the response variable in the model formula.

```
Call:  glm(formula = chd5 ~ chol + sbp + smoke, family = binomial(),
    data = data_13_15)

Coefficients:
(Intercept)         chol          sbp       smoke2       smoke3
    -9.1372       0.3391       0.0208       0.4016       0.7089
```

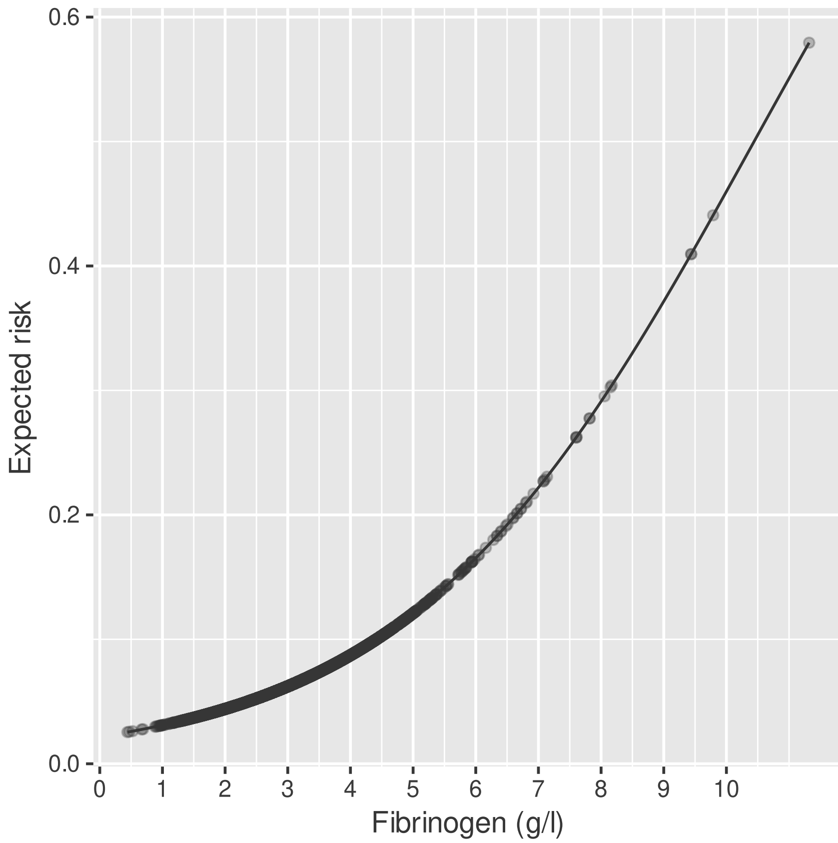


FIGURE 13.5
Replication of figure 13.11

```
Degrees of Freedom: 4048 Total (i.e. Null);  4044 Residual
Null Deviance:      1040
Residual Deviance: 989  AIC: 999
```

Example 13.6 (page 622)

We use `coxph` to fit the Cox proportional hazards model of example 13.6.

```
coxph(Surv(survive, cvdfllw == "yes") ~ fibrinogen,
      data = data_13_1f,
      x =TRUE,
      y =TRUE,
      ties ="breslow") -> cphf_13_6
predict(cphf_13_6,
        newdata = data.frame(fibrinogen =mean(data_13_1f$fibrinogen),
                             survive = 3652,
                             cvdfllw = "yes"),
        type = "survival") -> meansurv_13_6
1 - meansurv_13_6 ^ exp(coef(cphf_13_6) * (2-mean(data_13_1f$fibrinogen)))
```

We use `predict` to calculate the 10 year survival at the mean level of fibrinogen. In addition to the fitted model, `predict` needs `newdata`, a dataframe containing the values at which it should predict. The fields in the `newdata` should be named as in the original data frame used to fit the model. All columns used in the formula should be available in the new data frame. Thus, our `newdata` contains *fibrinogen* with its value set as the mean of *fibrinogen* in the original data frame, *survive* set to 3652 to correspond to ten years and *cvdfllw* set to "yes". The value of *cvdfllw* doesn't actually matter, but the field needs to be included. Finally, we specify `type = "survival"` to say that we want the estimate of survival. We then use the formula given in our textbook to calculate the predicted risk for a fibrinogen value of 2.

```
fibrinogen
  0.047102
```

We may follow another path to arrive at the same figure.

```
1 - summary(survfit(cphf_13_6,newdata = data.frame(fibrinogen = 2)),
            times = 3652 )$surv
```

We use `survfit` supplying it with the model object and `newdata`. The value returned by `survfit` is given as an argument to `summary` along with `times`, the time period we want. We select the `surv` component from the list returned. This is deducted from 1 to arrive at the predicted risk at 10 year for a fibrinogen level of 2.

```
[1] 0.047102
```

The `newdata` that we supply to `predict` or `survfit` is expected to be in the same form as the original data frame used to build the model. This means that factors should be supplied as factors. We cannot give the proportion of a particular value, a number, in its place. So, we cannot use `predict` and `survfit` to rework the example 13.7. We will try to reproduce table 13.8.

Table 13.8 (page 623)

```
coxph(Surv(survive, chd) ~ chol + sbp +smoke,
      data = data_13_15,
      ties ="breslow",
      x = TRUE,
      y = TRUE) -> cph_13_7
bind_cols(Variable =  names(coef(cph_13_7)),
          b = coef(cph_13_7),
          Mean = c(mean(data_13_15$chol),
                   mean(data_13_15$sbp),
                   prop.table(table(data_13_15$smoke))[-1]))|>
  mutate(Product = Mean * b) -> tbl_13_8
tbl_13_8
```

After fitting the requisite model using `coxph`, we use `bind_cols` to column bind the variable names, the coefficients of the model and mean of the variables. We obtain the coefficients using `coef` and the names associated with the coefficients using `names`. For the mean, we use `mean` to calculate the mean of the continuous variables and for the factor, we tabulate the data using `table`, convert them to proportions using `prop.table` and remove the value for the reference level by negative subsetting.

TABLE 13.4
Replication of table 13.8

Variable	b	Mean	Product
chol	0.29483	6.35832	1.87465
sbp	0.02186	133.22289	2.91223
smoke2	0.32380	0.25537	0.08269
smoke3	0.65895	0.52038	0.34290

As told earlier, `predict` and `survfit` expects the `newdata` to contain the same columns as in the database originally used for fitting the model. Thus, we cannot assign proportions to factors like *smoke*. So, we cannot derive the risk score for the mean combination of factor levels as given in example 13.7 of our textbook. However, the purpose of deriving the risk score for the mean combination is to predict the risk score for a set of values. This is possible using `predict` or `survfit`. Here we predict the risk for an ex-smoker with systolic blood pressure 150 and cholesterol 6.5 using `survfit`.

```
1 - summary(survfit(cph_13_7,
                    newdata = data.frame(smoke = "2",
                                         sbp = 150,
                                         chol = 6.5)),
            times = 1827 )$surv
```

Comparing it with the value derived using the formula given in our text, we can confirm that there is only rounding error.

```
1 - (0.97648 ^ exp((0.29483*6.5) + (0.02186 * 150) + 0.32380 - 5.21247))
```

```
[1] 0.03182
[1] 0.031827
```

Note that if we don't provide a `newdata` argument to `survfit`, it will return the value for the baseline combination. The difference from the mean combination given in our textbook is that though the continuous variables are considered at their mean values, factors are considered at their reference levels. Thus, the value given by

```
1 - summary(survfit(cph_13_7),
            times = 1827 )$surv
```

is same as that given by

```
1 - (0.97648 ^ exp((0.29483*mean(data_13_15$chol)) +
                    (0.02186 * mean(data_13_15$sbp)) -
                    5.21247))
```

```
[1] 0.015427
[1] 0.015431
```

except for rounding error.

Table 13.9 (page 623)

To prepare table 13.9, we will use `epiR` and `DescTools`.

```
library(epiR)
library(DescTools)
Rev(
  table(
    ifelse(predict(lg_13_4, type = "response") <= 0.1,"N", "Y"),
    data_13_1f$cvd10)) -> tbl_13_9
tbl_13_9
```

We use `table` to create the two-by-two table. We provide two vectors to `table`. The first is returned by `ifelse` and has the values "N" or "Y" based on whether the value returned by `predict` is less than or equal to 0.1. The second is the *cvd10* column of fibrinogen data. We use `Rev` to reverse the order of the columns and rows.

TABLE 13.5
Replication of table 13.9

	True CVD outcome		
Test	Positive	Negative	Total
Y	30	224	254
N	374	5923	6297

This is passed to `epi.tests` and the relevant components from the result are selected for printing.

```
filter(epi.tests(tbl_13_9)$detail,
       statistic == "se" |  statistic == "sp")
```

```
  statistic      est    lower   upper
1        se 0.074257 0.050659 0.10431
2        sp 0.963559 0.958569 0.96810
```

Figure 13.12 (page 624)

To prepare the inverse ogive of figure 13.12, we need a custom function that returns the percentage with the specified risk or higher.

```
prophigher <- function(x) {
  map_dbl(x,
          function(val)
            prop.table(
              table(
                predict(lg_13_15,
                        type = "response") >= val))["TRUE"])}
```

We name our custom function as *prophigher*. We specify that it will accept one argument which is called *x*. This argument is passed on to `map_dbl` for processing. The function `map_dbl` passes each value of its first argument to its second argument. In our example, its second argument is an anonymous function that calculates the proportion of values greater than or equal to the supplied value in the vector returned by `predict`, the arguments for which is *lg_13_15*, our logistic model object. Thus, when we pass a vector to *prophigher* we will get a vector equal in length to the vector passed to *prophigher*, with each element denoting the proportion of predicted values above or equal to the corresponding value in the vector passed to *prophigher*.

Now, we use `ggplot` and `geom_function` to plot the inverse ogive.

```
ggplot() +
  geom_function(fun = prophigher) +
  xlim(0,.2) +
  labs(x = "10-year risk (%)",
       y = "Percentage with this risk or higher")
```

13.3 Quantifying discrimination

Figure 13.13 (page 625)

We will use the package `ROCit` to prepare ROC plot of figure 13.13.

```
library(ROCit)
rocit(lg_13_4$fitted.values, lg_13_4$y) -> roc_13_13
plot(roc_13_13, YIndex = FALSE, legend = FALSE)
```

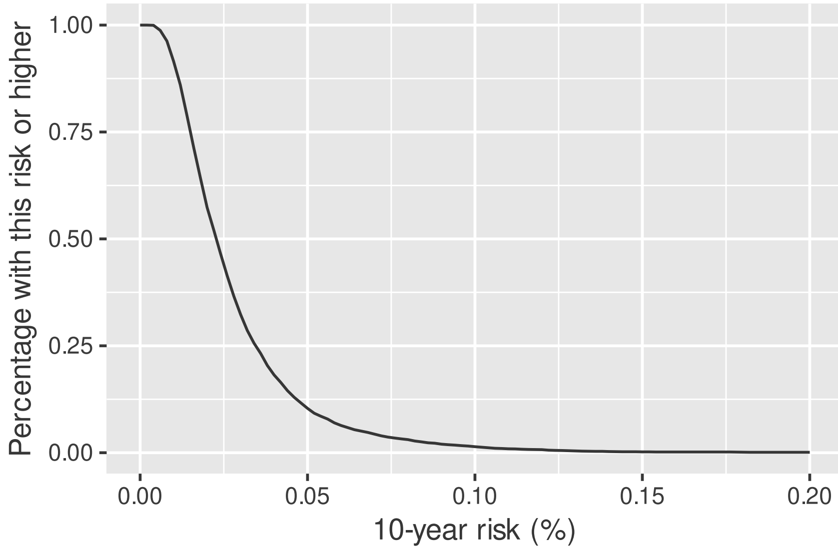


FIGURE 13.6
Replication of figure 13.12

The function that does all the calculations is `rocit`. It requires two main arguments – `score` and `class`. The `score` is a vector that contains the diagnostic / risk score and `class` is a vector of same length as `score` containing the category of outcome. In our example, we provide the `fitted` component of the model object as `score` and the `y` component as `class`. We store this object with a name. To plot the graph, we simply pass it to `plot`. We use `legend = FALSE` and `YIndex = FALSE` to remove the legend and the optimal Youden index, which are plotted by default.

To obtain AUC, we may use `summary` method for the ROC object or use `ciAUC`.

```
ciAUC(roc_13_13)
```

The argument required for `ciAUC` is again the ROC object. It returns the AUC as well as its confidence interval.

```
estimated AUC : 0.606916035673846
AUC estimation method : empirical

CI of AUC
confidence level = 95%
lower = 0.576933312481726     upper = 0.636898758865965
```

We need to calculate Somer's D by hand using the AUC.

```
ciAUC(roc_13_13)$AUC * 2 -1
```

```
[1] 0.21383
```

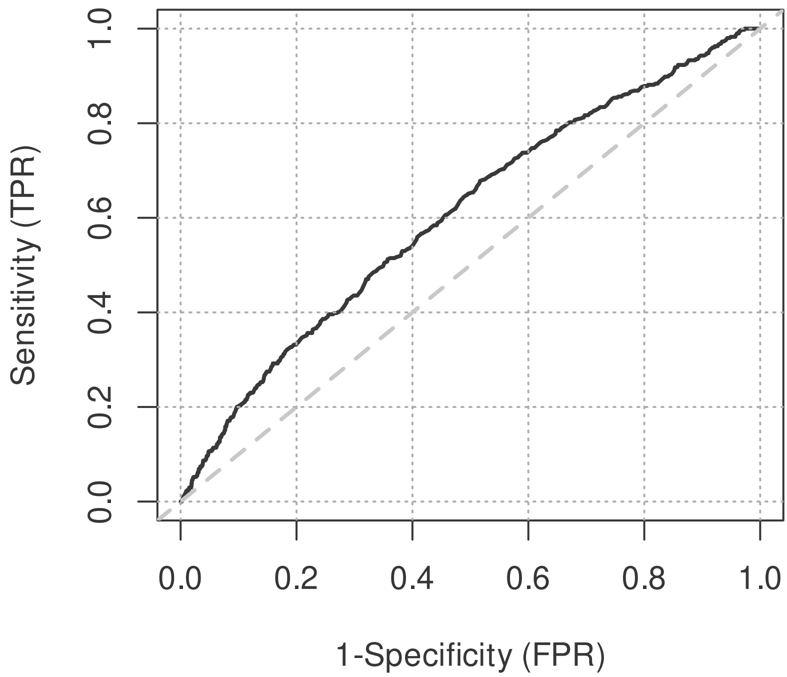


FIGURE 13.7
Replication of figure 13.13

Figure 13.14 (page 627)

We may plot the ROC curves in figure 13.14 similarly, but by typing the `score` and `class` arguments directly.

```
rocit(c(2,4,6,8,10,12,14,9,11,13,15),
      c(0,0,0,0,0,0,0,1,1,1,1)) -> roc_13_14b
rocit(c(2,4,6,8,10,12,14,9,10,13,15),
      c(0,0,0,0,0,0,0,1,1,1,1)) -> roc_13_14d
plot(roc_13_14b,legend = FALSE, YIndex = FALSE)
plot(roc_13_14d,legend = FALSE, YIndex = FALSE)
ciAUC(roc_13_14b)
ciAUC(roc_13_14d)
```

```
estimated AUC : 0.785714285714286
AUC estimation method : empirical

CI of AUC
confidence level = 95%
lower = 0.475375966723316     upper = 1

estimated AUC : 0.767857142857143
AUC estimation method : empirical
```

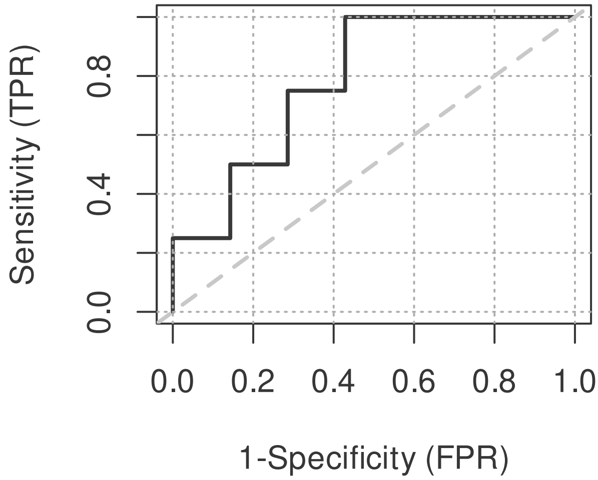


FIGURE 13.8
Replication of figure 13.14(b)

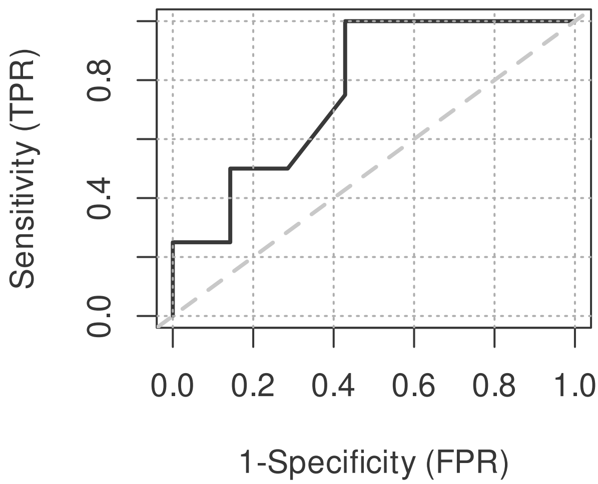


FIGURE 13.9
Replication of figure 13.14(d)

```
CI of AUC
confidence level = 95%
lower = 0.448466991941638     upper = 1
```

Example 13.10 (page 629)

We will use the library `pROC` to compare different ROC curves / AUC. First, we need to fit the additional logistic models.

```
library(pROC)
roc(lg_13_15$y, lg_13_15$fitted.values) -> roc_13_10a
glm(chd5 ~  chol + sbp + smoke + bmi,
```

```
    data = data_13_15,
    family = binomial()) -> lg_13_10bmi
glm(chd5 ~  sbp,
    data = data_13_15,
    family = binomial()) -> lg_13_10sbp
roc(lg_13_10bmi$y, lg_13_10bmi$fitted.values) -> roc_13_10bmi
roc(lg_13_10sbp$y, lg_13_10sbp$fitted.values) -> roc_13_10sbp
roc.test(roc_13_10a, roc_13_10sbp)
roc.test(roc_13_10a, roc_13_10bmi)
```

The function to prepare ROC curves `roc` is similar in usage to `rocit`, except that the order of the two arguments is reversed. We save the ROC objects and supply them as arguments to `roc.test`, which performs the DeLong's test. Note that the function accepts the argument `alternative` if you want to do a one sided test.

```
    DeLong's test for two correlated ROC curves

data:  roc_13_10a and roc_13_10sbp
Z = 3.61, p-value = 3e-04
alternative hypothesis: true difference in AUC is not equal to 0
95 percent confidence interval:
 0.033049 0.111484
sample estimates:
AUC of roc1 AUC of roc2
    0.69772     0.62545

    DeLong's test for two correlated ROC curves

data:  roc_13_10a and roc_13_10bmi
Z = -0.909, p-value = 0.36
alternative hypothesis: true difference in AUC is not equal to 0
95 percent confidence interval:
 -0.0091952  0.0033681
sample estimates:
AUC of roc1 AUC of roc2
    0.69772     0.70063
```

The ROC objects that are returned by `roc` can be provided as argument to `plot` to plot the curve. Here, we prepare the graph in figure 13.15.

Figure 13.15 (page 630)

```
plot(roc_13_10a, lwd =1, legacy.axes = TRUE)
plot(roc_13_10bmi, add = TRUE, lty = 5, lwd =1, col = 5)
plot(roc_13_10sbp, add = TRUE, lty = 4, lwd =1, col = 4)
```

The `add = TRUE` argument is supplied to all calls of `plot` except the first. This is to make sure that the ROC plots are added to the first. The `legacy.axes = TRUE` chooses one set of axes labels. The `lwd` argument determines the thickness of the lines used for plotting; the `lty` determines the line type – continuous or dashed or dotted and `col` determines the colour of the line.

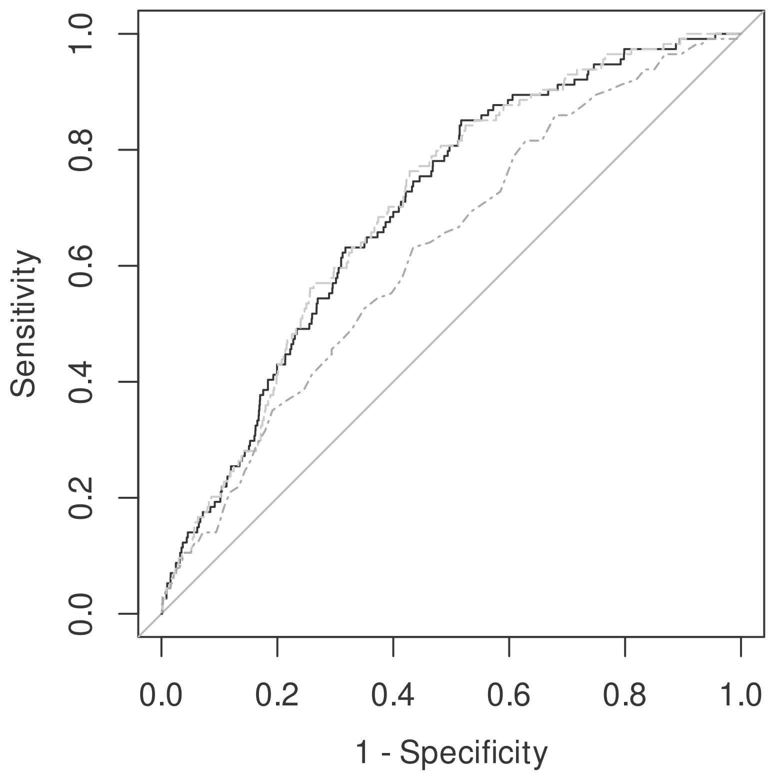


FIGURE 13.10
Replication of figure 13.15 using pROC

The package `rocit` also allows us to add multiple ROC lines, but by using `lines`. While the graphs of `ROCit` looks better to me, it doesn't have a function to compare different ROCs / AUCs. We may prepare the same graph using `ROCit`.

```
rocit(lg_13_15$fitted.values, lg_13_15$y) -> rocit_13_10a
rocit(lg_13_10bmi$fitted.values,
      lg_13_10bmi$y) -> rocit_13_10bmi
rocit(lg_13_10sbp$fitted.values,
      lg_13_10sbp$y) -> rocit_13_10sbp
plot(rocit_13_10bmi, YIndex = FALSE, legend = FALSE)
lines(rocit_13_10a$TPR~rocit_13_10a$FPR,
      col = 2, lty = 2, lwd =2)
lines(rocit_13_10sbp$TPR~rocit_13_10sbp$FPR,
      col = 3, lty = 3, lwd =2)
legend("bottomright",
       c( "Cholesterol, blood pressure, smoking and BMI",
          "Cholesterol, blood pressure, smoking",
```

```
          "Blood Pressure"),
       lwd = 2,
       col = c(1,2,3),
       lty = c(1,2,3))
```

The three ROC objects are prepared using `rocit`. The first curve is plotted using `plot`. The latter lines are added using `lines`. The argument for `lines` is a formula with the `TPR` component of the relevant ROC object as the left-hand side and `FPR` component as the right-hand side. We may provide `lty`, `lwd` and `col` as per our wish. Finally, we use `legend` to add a legend. We need to specify the location of the legend by means of a string. Here we say `"bottomright"`. The labels are specified next. In our case, there are three lines; hence, a character vector of length 3. Similarly, `lty` and `col` are also vectors of length 3, with values corresponding to what we used in `lines`. The `lwd` is same for all three lines; hence, only one value is supplied.

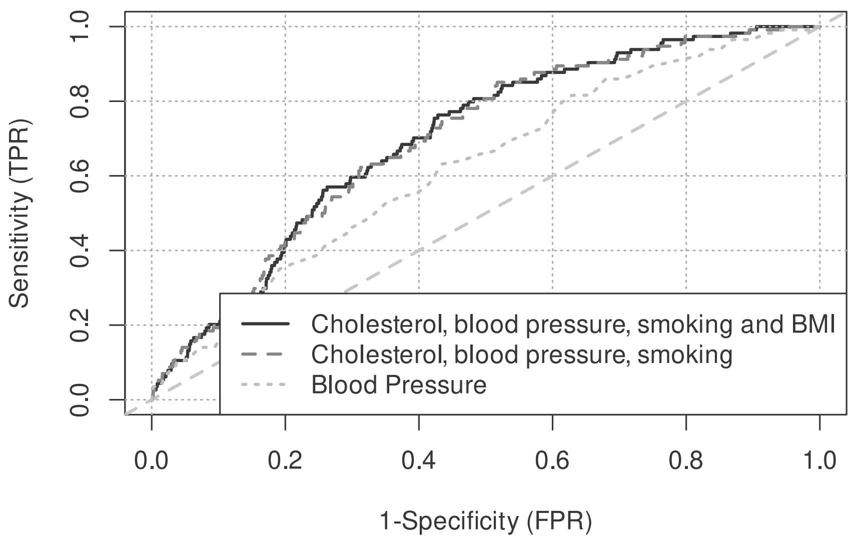


FIGURE 13.11
Replication of figure 13.15 using ROCit

Section 13.4.3 (page 631)

The function `concordance` of `survival` returns Harrel's c statistic when it is passed a Cox proportional hazards model object. We use the Cox model object that we fitted for fibrinogen data using all survival times.

```
concordance(cphf_13_6)
```

```
Call:
concordance.coxph(object = cphf_13_6)

n= 6551
```

```
Concordance= 0.6 se= 0.0092
concordant discordant     tied.x     tied.y    tied.xy
   3093011    2057184      16396         75          0
```

For other worked examples in section 13.4.3, we need to fit the appropriate Cox models. We do this inside `concordance` without storing the model object.

```
concordance(coxph(Surv(survive, cvd10 == "yes") ~ fibrinogen,
                   data = data_13_1f,
                   ties ="breslow"))$concordance
concordance(coxph(Surv(survive, chd5) ~ chol + sbp +smoke,
                   data = data_13_15,
                  ties ="breslow"))$concordance
concordance(coxph(Surv(survive, chd5) ~
                    chol + sbp +smoke + bmi,
                   data = data_13_15,
                  ties ="breslow"))$concordance
```

```
[1] 0.60594
[1] 0.69556
[1] 0.69854
```

Example 13.11 (page 634)

We will use the package `effectsize` to calculate the standardised mean effect sizes.

```
library(effectsize)
hedges_g(predict(lg_13_4) ~ factor(lg_13_4$y,levels = c("1", "0")))
```

We use `hedges_g` which accepts a formula. The left-hand side is the risk scores as provided by `predict`. The right-hand side is a factor of same length as the first, with two values, which is used to sort the first vector into two groups. We provide `levels` argument while building the factor vector to ensure that the without CVD group is subtracted from the with CVD group. Note that the result is different from that in our text. It looks like that the difference is due to the use Bessel's correction (n -1 instead of n as the denominator) while calculating the means.

```
Hedges' g |       95% CI
------------------------
0.37      | [0.27, 0.47]

- Estimated using pooled SD.
```

Table 13.10 (page 634)

Now, we try to recreate table 13.10. We build custom functions to obtain the standardised mean effect sizes and their confidence interval as one string.

```
buildcph <- function(f) {
  mdl <- coxph(f,data = data_13_15,ties ="breslow")
  hg <- hedges_g(predict(mdl) ~ factor(data_13_15$chd5,levels = c("1","0")))
  paste0(round(hg$Hedges_g,4),
         " (",
         round(hg$CI_low,4),
         ", ",
         round(hg$CI_high,4), ")")}

buildlgt <- function(f) {
  mdl <- glm(f, data = data_13_15,family = binomial())
  hg <- hedges_g(predict(mdl)~ factor(data_13_15$chd5,levels = c("1", "0")))
  paste0(round(hg$Hedges_g,4),
         " (",
         round(hg$CI_low,4),
         ", ",
         round(hg$CI_high,4),
         ")")}
```

Our custom functions *buildcph* and *buildlgt* are similar. They both accept a formula. In *buildph*, a Cox model is built from this formula, which is passed on to `hedges_g` with the appropriate arguments. The appropriate components of the result of `hedges_g` is joined together using `paste0` and returned. The custom function *buildlgt* uses the formula to build a logistic model; otherwise, it is similar to *buildcph*.

```
data.frame(`Variables in model` =
             c("SBP",
               "SBP, total cholesterol, and smoking",
               "SBP, total cholesterol, smoking and BMI"),
           `Logistic model` =
             map_chr(c(chd5 ~ sbp,
                       chd5 ~ chol + sbp +smoke,
                       chd5 ~ chol + sbp +smoke + bmi),
                     .f =  buildlgt),
           `Cox model` =
             map_chr(c(Surv(survive, chd5) ~ sbp,
                       Surv(survive, chd5) ~ chol+sbp+smoke,
                       Surv(survive, chd5) ~
                         chol + sbp +smoke + bmi),
                     .f =  buildcph))
```

We call these functions using `map_chr` which is provided a vector of appropriate formulas and the function name. `map_chr` will pass each element of the vector to the function and collect the result into a character vector. We call `map_chr` twice – once with *build_cph* and once with *buildlgt*. `map_chr` is called from inside `data.frame`. Thus, the results of `map_chr` are joined together as a dataframe along with a vector of characters to show the variable names. As we use column headings with embedded spaces, those names are enclosed in a pair of back ticks.

TABLE 13.6
Replication of table 13.10

Variables.in.model	Logistic.model	Cox.model
SBP	0.477 (0.2905, 0.6634)	0.477 (0.2905, 0.6634)
SBP, total cholesterol, and smoking	0.6935 (0.5067, 0.8802)	0.693 (0.5062, 0.8798)
SBP, total cholesterol, smoking and BMI	0.6981 (0.5113, 0.8849)	0.6977 (0.5109, 0.8845)

Section 13.4.4 (page 632)

We need to calculate the standardised median effect size by hand.

```
(qnorm(0.75) - qnorm(0.25)) *
  (median(predict(lg_13_4)[data_13_1f$cvd10 == "yes"]) -
     median(predict(lg_13_4)[data_13_1f$cvd10 == "no"])) /
  IQR(predict(lg_13_4)[data_13_1f$cvd10 == "no"])
```

We use `qnorm` to calculate the distance between the third and first quartiles of a standard normal distribution. We use `median` and `IQR` to calculate median and inter quartile range. The argument for these functions is the vector returned by `predict` subset by logical subscripting.

```
[1] 0.40509
```

The functions `pnorm` and `qnorm` can be used to solve equations 13.12 and 13.13.

```
qnorm(0.9, mean = 20, sd =2)
pnorm(0.1184)
```

When using `qnorm` we are asking what would be the value below which 90% of the values of a normal distribution with mean 20 and standard deviation 2 occurs. When using `pnorm` we are asking the reverse – given a normal distribution (mean 0 and sd 1), what proportion of values fall below the specified value.

```
[1] 22.563
[1] 0.54712
```

13.4 Calibration

Section 13.5.1 (page 638)

We need `rms` package to perform Spiegelhalter test.

```
library(rms)
val.prob(lg_13_4$fitted.values, lg_13_4$y, pl = FALSE)["S:z"]
```

We use the function `val.prob`. It requires a vector of predicted probabilities which is taken from the `fitted.values` of the model object and a vector which denotes the outcome which is taken from the `y` component of the model object. By default, `val.prob` plots a calibration plot. We ask it not to plot the graph by saying `pl = FALSE`. Multiple statistics are returned by `val.prob`. We select the value we are interested in by subscripting.

```
     S:z
0.095867
```

Table 13.11 (page 640)

We will now try to recreate table 13.11.

```
lg_13_4$fitted.values -> data_13_1f$predlg
cut(lg_13_4$fitted.values,
    quantile(lg_13_4$fitted.values, seq(0,1, by = 0.1)),
    include.lowest = TRUE,
    labels = FALSE) -> data_13_1f$tenth

group_by(data_13_1f, tenth) |>
  summarise(maxlg = max(predlg),
            meanlg = mean(predlg),
            numcvd10 =  sum(cvd10 == "yes"),
            numrisk = n(),
            obsrisk = numcvd10 /numrisk,
            dfrnc = obsrisk - meanlg,
            hlcmp = (numrisk * dfrnc ^ 2) /  (meanlg * (1-meanlg))
            ) -> tbl_13_11
tbl_13_11
```

We add two new columns to our fibrinogen data frame – one containing the predicted risks and another categorising these risks into their tenths using `cut`. We have used `labels = FALSE` to say that we don't want the default labels, but integers to indicate the tenth. Then we group the dataframe by these tenths and then calculate the columns by simple arithmetic. The number of CVDs is calculated by `sum(cvd == "yes")`. Here, we get a logical vector, whose values gets coerced to zeros and ones and then summed. When the condition is true, we get the number 1 and when it is false 0. Thus, the sum will equal to the number of instances where the condition is true. Note that the categories are different from that in our textbook as is expected.

We can now sum the values of the last column to get the Hosmer Lemeshow statistic and calculate its p value.

```
sum(tbl_13_11$hlcmp) -> hl_13_11
pchisq(hl_13_11, df = nrow(tbl_13_11) - 2, lower.tail = FALSE)
```

```
[1] 0.64687
```

Again, the value is slightly different from our textbook.

TABLE 13.7
Replication of table 13.11

Tenth of predicted risk	Maximum value	Mean predicted risk ($\overline{p}$)	Number with CVD in 10 years	No. at risk	Observed risk (r)	Difference ($r - \overline{p}$)	Component of HL
1	0.044270	0.040285	24	674	0.035608	−0.0046770	0.381340
2	0.048095	0.046457	26	638	0.040752	−0.0057046	0.468680
3	0.051168	0.049720	27	677	0.039882	−0.0098384	1.386916
4	0.053980	0.052626	29	639	0.045383	−0.0072424	0.672275
5	0.056937	0.055518	38	664	0.057229	0.0017110	0.037074
6	0.060292	0.058702	46	668	0.068862	0.0101600	1.247909
7	0.064612	0.062446	41	634	0.064669	0.0022226	0.053494
8	0.070821	0.067541	41	659	0.062215	−0.0053258	0.296794
9	0.083237	0.076080	56	650	0.086154	0.0100743	0.938516
10	0.579411	0.108473	76	648	0.117284	0.0088111	0.520207

Instead of doing the calculations by hand, we may use the `DescTools` package.

```
HosmerLemeshowTest( lg_13_4$fitted.values, lg_13_4$y)$C
```

The function `HosmerLemeshowTest` requires two arguments similar to `val.prob`. The function returns both C and H statistics. As we are interested only in the C statistic, we subset that component.

```

    Hosmer-Lemeshow C statistic

data:  lg_13_4$fitted.values and lg_13_4$y
X-squared = 6, df = 8, p-value = 0.65
```

Table 13.12 (page 641)

We now turn our attention to table 13.12.

```
1 - predict(cph_13_7,
            newdata = mutate(data_13_15, survive = 1827),
            type = "survival") -> data_13_15$predcph
cut(data_13_15$predcph,
    quantile(data_13_15$predcph, seq(0,1, by = 0.1)),
    include.lowest = TRUE,
    labels = FALSE) -> data_13_15$tenth
summary(survfit(Surv(survive, chd5) ~ strata(tenth),
                data = data_13_15),
        times = 1827,
        data.frame = TRUE)  -> km_13_12
data.frame(obsrisk = 1 - km_13_12$surv ,
           lciobr = (1-km_13_12$surv) - 1.96 * km_13_12$std.err,
```

```
           uciobr = (1-km_13_12$surv) + 1.96 * km_13_12$std.err
           ) -> tbl_13_12part
group_by(data_13_15, tenth) |>
  summarise(meanrisk = mean(predcph),
            numrisk = n()) |>
  bind_cols(tbl_13_12part) |>
  mutate(dfrnc = obsrisk - meanrisk,
         hlcmp = (numrisk * dfrnc ^ 2) /(meanrisk * (1-meanrisk))
         ) -> tbl_13_12

select(tbl_13_12, -c("lciobr", "uciobr"))
```

We add predicted risk as a new column to the original data frame. `predict` can provide us many predicted values including survival probability, the complement of which is the risk of event. However, `predict` will calculate the survival probability for up to the survival period specified in the data used to fit the model. So, we give it a `newdata` which is the original data modified to make all survival periods 1827. We may instead use `predictRisk` of `riskRegression` to get the predicted probabilities. The result is added to the original data frame as a new column.

In order to obtain the observed risk, we use `survfit`. On the right-hand side of the formula we supply the term `strata(tenth)`, *tenth* being the column we added by using `cut` to assign labels according to the tenths of the predicted risks. Thus ten different KM curves are calculated corresponding to each of the tenths. We pass the result to `summary` and extract the `surv` component, subtract it from one and column bind it with its upper and lower confidence intervals to create a temporary data frame. The original dataframe is grouped by the tenths and the number at risk and mean predicted risk calculated for each group. To this, the temporary data frame is joined. Finally, the difference between observed and predicted risk and the contribution of each tenth to the test statistic are calculated.

TABLE 13.8
Replication of table 13.12

Tenth of predicted risk	Mean predicted risk ($\bar{p}$)	No. at risk	Observed risk (r)	Difference ($r - \bar{p}$)	Component of HL
1	0.0088176	405	0.0049938	−0.0038238	0.677561
2	0.0126936	405	0.0075252	−0.0051684	0.863233
3	0.0154448	405	0.0124202	−0.0030246	0.243658
4	0.0182038	405	0.0100260	−0.0081778	1.515452
5	0.0212371	405	0.0225144	0.0012772	0.031786
6	0.0248020	404	0.0322223	0.0074204	0.919713
7	0.0288538	405	0.0350529	0.0061991	0.555418
8	0.0346723	405	0.0450612	0.0103889	1.305991
9	0.0437504	405	0.0596532	0.0159028	2.448214
10	0.0756900	405	0.0548204	−0.0208696	2.521319

The first and sixth tenths in our categories differs by one from our textbook values. Consequently, the difference and HL statistic components calculated for those rows also differ slightly.

To calculate the p value of the test statistic, we use `pchisq` with the appropriate degrees of freedom.

```
sum(tbl_13_12$hlcmp)
pchisq(sum(tbl_13_12$hlcmp),
       df = nrow(tbl_13_12) - 1,
       lower.tail = FALSE)
```

```
[1] 11.082
[1] 0.27011
```

The HL statistic calculated and its p value are essentially the same as in our textbook.

Figure 13.18 (page 642)

To prepare the graph in figure 13.18, we will use the table we prepared above.

```
ggplot(tbl_13_12) +
  geom_point(aes(x = meanrisk, y = obsrisk )) +
  geom_abline(intercept = 0, slope = 1, colour = "grey") +
  ylim(0,0.08)+
  xlim(0,0.08) +
  labs(x = "Predicted risk",
       y = "Observed risk")
```

We pass the data frame to `ggplot` and use `geom_point` to plot the points. We use `geom_abline` to draw the diagonal line of perfect calibration. We specify the same `xlim` and `ylim` to have a pleasing appearance.

Figure 13.19 (page 642)

To prepare the graph of figure 13.19 too we use the table we prepared earlier.

```
ggplot(tbl_13_12) +
  geom_pointrange(aes( x = 1:10,
                       y = obsrisk / meanrisk,
                       ymin = lciobr / meanrisk,
                       ymax = uciobr / meanrisk)) +
  geom_hline(yintercept = 1, colour = "grey") +
  scale_x_discrete(limits = 1:10) +
  theme_minimal() +
  xlab("Tenth of predicted risk") +
  ylab("Observed/Predicte risk")
```

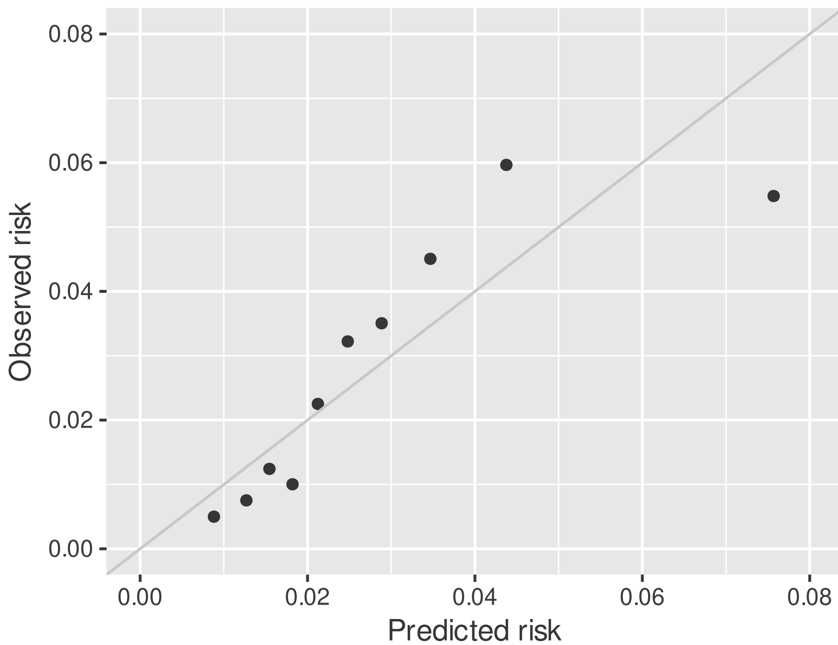


FIGURE 13.12
Replication of figure 13.18

We use `geom_pointrange` to plot the graph. Note that we provide a vector with value 1 to 10 as the `x` aesthetic and use `scale_x_discrete` with the same `limits`. This is done to plot the points and lines at equal distance on the x-axis rather than at a distance dictated by the mean predicted risk.

13.5 Recalibration

Example 13.15 (page 644)

To do the Cox's calibration test, we need the fibrinogen data for men.

```
filter(data_13_1,sex == "men") -> data_13_1m
predict(lg_13_4,
        newdata =  data_13_1m) -> data_13_1m$predict
glm(formula = cvd10 ~ offset(predict),
    family = binomial(),
    data = data_13_1m) -> lg_13_16a
summary(lg_13_16a)
```

After saving the fibrinogen data for men in a new data frame, we use the logistic model fitted for women to predict the logit for men. The predicted logits are added as a new column to the data frame for men. In the next step, we fit a logistic model using the dataframe for

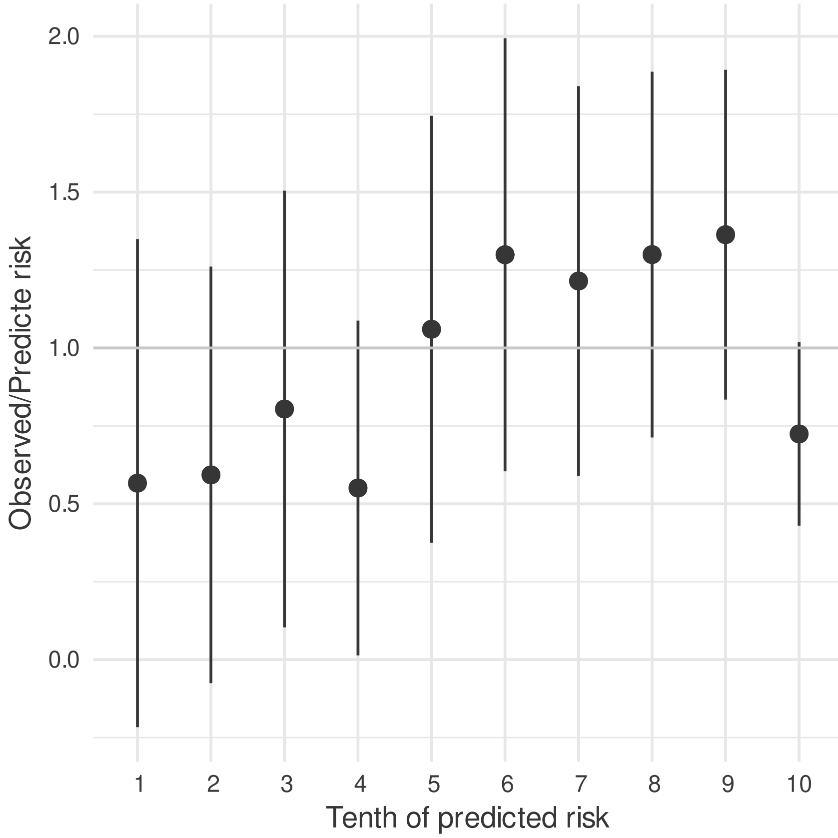


FIGURE 13.13
Replication of figure 13.19

men. The response variable in the model formula is CVD death, while the right-hand side is the predicted logits specified as an `offset`.

```
Call:
glm(formula = cvd10 ~ offset(predict), family = binomial(), data = data_13_1m)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)    0.737      0.039    18.9   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 4627.3  on 6508  degrees of freedom
Residual deviance: 4627.3  on 6508  degrees of freedom
AIC: 4629

Number of Fisher Scoring iterations: 4
```

Example 13.16 (page 645)

For Cox recalibration, we need to fit the model without an offset.

```
glm(formula = cvd10 ~ predict,
    family = binomial(),
    data = data_13_1m) -> lg_13_16b
summary(lg_13_16b)
```

```
Call:
glm(formula = cvd10 ~ predict, family = binomial(), data = data_13_1m)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)    0.165      0.323    0.51     0.61
predict        0.791      0.117    6.79  1.1e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 4667.6  on 6508  degrees of freedom
Residual deviance: 4624.1  on 6507  degrees of freedom
AIC: 4628

Number of Fisher Scoring iterations: 4
```

To confirm that we get the predicted risk as given in our textbook, we build a function to convert logits to risks.

```
logit2r <- function(logit) { (1 + exp(- logit)) ^ -1}
logit2r(predict(lg_13_4,
                 newdata = data.frame(fibrinogen = 5)))
logit2r(predict(lg_13_4,
                 newdata = data.frame(fibrinogen = 5)) +
          coef(lg_13_16a))
logit2r((predict(lg_13_4, newdata = data.frame(fibrinogen = 5)) *
           coef(lg_13_16b)["predict"]) +
          coef(lg_13_16b)["(Intercept)"])
```

Our function *logit2r* accepts a logit and does the necessary arithmetic to return the corresponding risk. We pass it the logits calculated for a fibrinogen level of 5 using the prediction from the female fibrinogen data, recalibrating it using intercept method and by Cox method to confirm agreement with textbook values.

```
      1
0.12094
      1
0.22325
```

```
      1
0.19698
```

Next, we add new columns to the male data corresponding to intercept recalibrated and Cox recalibrated logits.

```
data_13_1m$predict + coef(lg_13_16a) -> data_13_1m$irlg
(data_13_1m$predict *
    coef(lg_13_16b)["predict"]) +
  coef(lg_13_16b)["(Intercept)"] -> data_13_1m$crlg
mean(logit2r(data_13_1m$irlg))
mean(logit2r(data_13_1m$crlg))
sum(data_13_1m$cvd10 == "yes") / nrow(data_13_1m)
```

We confirm that the mean of predicted risks using intercept recalibration and Cox recalibration agree with the textbook values and with the observed risk.

```
[1] 0.11584
[1] 0.11584
[1] 0.11584
```

Figure 13.21 (page 647)

To prepare the graph in figure 13.21, we need to calculate the observed and predicted risk in each tenths and gather it in a data frame.

```
cut(data_13_1m$predict,
    quantile(data_13_1m$predict, seq(0,1, by = 0.1)),
    include.lowest = TRUE,
    labels = FALSE) -> data_13_1m$ucl10
cut(data_13_1m$irlg,
    quantile(data_13_1m$irlg, seq(0,1, by = 0.1)),
    include.lowest = TRUE,
    labels = FALSE) -> data_13_1m$irlg10
cut(data_13_1m$irlg,
    quantile(data_13_1m$irlg, seq(0,1, by = 0.1)),
    include.lowest = TRUE,
    labels = FALSE) -> data_13_1m$crlg10

bind_rows(summarise(group_by(data_13_1m, ucl10),
                    meanpred = mean(logit2r(predict)),
                    observed = sum(cvd10 == "yes") /n(),
                    calib = "uncalibrated"),
          summarise(group_by(data_13_1m, irlg10),
                    meanpred = mean(logit2r(irlg)),
                    observed = sum(cvd10 == "yes") /n(),
                    calib = "intercept"),
          summarise(group_by(data_13_1m, crlg10),
```

```
                     meanpred = mean(logit2r(crlg)),
                     observed = sum(cvd10 == "yes") /n(),
                     calib = "Cox")) -> data_13_21
```

First, we `cut` each of the three predicted logits into their tenths and add them as new columns to the original dataframe. Next, we group the dataframe according to these tenths and calculate the mean of the predicted risk returned by *logit2r* and the observed risk. This step is done separately for the three predictions and the values added together as a single dataframe along with a column to distinguish the type of prediction used.

```
ggplot(data_13_21) +
  geom_point(aes(x = meanpred,
                 y = observed,
                 group = calib,
                 colour = calib,
                 shape = calib),
             show.legend = FALSE) +
  geom_abline(intercept = 0, slope = 1, colour = "grey") +
  xlim(0.025,0.2) +
  ylim(0.025,0.2) +
  xlab("Predicted risk") +
  ylab("Observed risk") +
  scale_color_manual(values = c("#111111", "#004B73", "#713430"))
```

Next, we pass the data frame we built to `ggplot`. We use `geom_point` to plot points using the observed and predicted risks for each tenths. We ensure that each of the three different predictions used have different colours and plotting symbols by specifying the value of `group`, `colour` and `shape` aesthetics to that of the column with the identifier for prediction method. The `show.legend = FALSE` specified outside `aes` asks `ggplot` to not draw a legend for the geom within which it is specified.

The Hosmer Lemeshow test is done using `HosmerLemeshowTest`.

```
HosmerLemeshowTest( logit2r(data_13_1m$predict),
                    data_13_1m$cvd10 == "yes")$C
HosmerLemeshowTest( logit2r(data_13_1m$irlg),
                    data_13_1m$cvd10 == "yes")$C
HosmerLemeshowTest( logit2r(data_13_1m$crlg),
                    data_13_1m$cvd10 == "yes")$C
```

```
    Hosmer-Lemeshow C statistic

data:  logit2r(data_13_1m$predict) and data_13_1m$cvd10 == "yes"
X-squared = 382, df = 8, p-value <2e-16

    Hosmer-Lemeshow C statistic
```

```
data:  logit2r(data_13_1m$irlg) and data_13_1m$cvd10 == "yes"
X-squared = 4.93, df = 8, p-value = 0.77

    Hosmer-Lemeshow C statistic

data:  logit2r(data_13_1m$crlg) and data_13_1m$cvd10 == "yes"
X-squared = 6.53, df = 8, p-value = 0.59
```

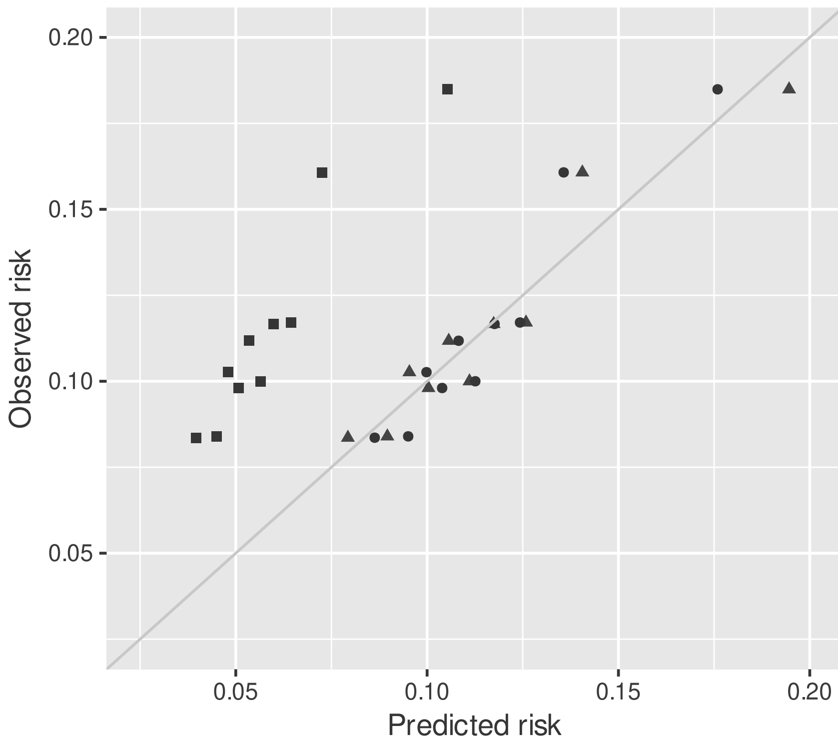


FIGURE 13.14
Replication of figure 13.21

Example 13.17

To rework the example 13.17, we need to fit the appropriate Cox model.

```
coxph(Surv(survive,cvdfllw == "yes") ~ fibrinogen,
      data = data_13_1m) -> cph_13_17
mean(data_13_1m$fibrinogen) -> meanfibm
summary(survfit(cph_13_17,
                newdata = data.frame(fibrinogen = meanfibm)),
        times = 3652)$surv -> survm
calfib <- function(fib){1 - (survm ^ exp(coef(cphf_13_6) *(fib -meanfibm)))}
```

We use the mean fibrinogen value in males and 10 year survival corresponding to that level to build a formula that will return the risk probability.

We will use the formula to extend *data_13_1m* and prepare the graph in figure 13.22.

```
calfib(data_13_1m$fibrinogen) -> data_13_1m$calcox
cut(data_13_1m$calcox,
    quantile(data_13_1m$calcox, seq(0,1, by = 0.1)),
    include.lowest = TRUE,
    labels = FALSE) -> data_13_1m$calcox10

bind_rows(data_13_21,
summarise(group_by(data_13_1m, calcox10),
          meanpred = mean(calcox),
          observed = sum(cvd10 == "yes") /n(),
          calib = "Calibrated Cox")) |>
  filter(calib =="Calibrated Cox"|calib == "uncalibrated") |>
  ggplot() +
  geom_point(aes(x = meanpred,
                 y = observed,
                 group = calib,
                 shape = calib,
                 colour = calib),
             show.legend = FALSE) +
  geom_abline(intercept = 0, slope = 1, colour = "grey") +
  labs(x = "Observed risk",
       y = "Predicted risk") +
  xlim(0,0.2) +
  ylim(0,0.2) +
  scale_color_manual(values = c("#111111", "#004B73", "#713430"))
```

As with our previous example, we add the predicted risk and its tenths as new columns. Then, we `group_by` the tenths and `summarise` to get the mean of the predicted and observed risks for each tenth. Before we pass the data frame to `ggplot`, we `filter` it to select the data of our interest.

13.6 Accuracy of predictions

Table 13.13 (page 649)

We will now try to recreate table 13.13.

```
y <- c(rep(1,6), rep(0,18))
S1 <- rep(1,24)
S2 <- rep(0,24)
S3 <- y
S4 <- c(rep(0,6), rep(1,18))
S5 <- rep(0.5,24)
S6 <- rep(0.25,24)
```

```
S7 <- c(rep(0.9,6), rep(0.1,18))
S8 <- c(rep(0.6,6), rep(0.4,18))
S9 <- c(0.5,0.45,0.4,0.35,0.3,0.25,0.31,0.26,0.2,rep(0.1,15))
S10 <- c(0.7,0.65,0.6,0.5,0.4,0.3,0.51,0.41,0.31,rep(0.1,13),0.22, 0.25)
S11 <- c(0.95, 0.7,0.5,0.4,0.3,0.1,0.8,0.4,0.3,0.25,0.2,0.18,0.16,
         0.16,0.15,0.14,0.14,0.12,0.12,0.1,0.1,0.05,0.05,0.01)
data_13_13 <- data.frame(y,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11)

Brier <- function(x,y) { sum((y - x)^2) / length(x)}
AUC <- function(x,y) { concordance( y ~ x)$concordance}
Spiegel <- function(x,y) { sum((y - x) * (1- (2 * x))) /
    sqrt(sum( (1 - (2 * x)) ^2 * x * (1-x)))}
```

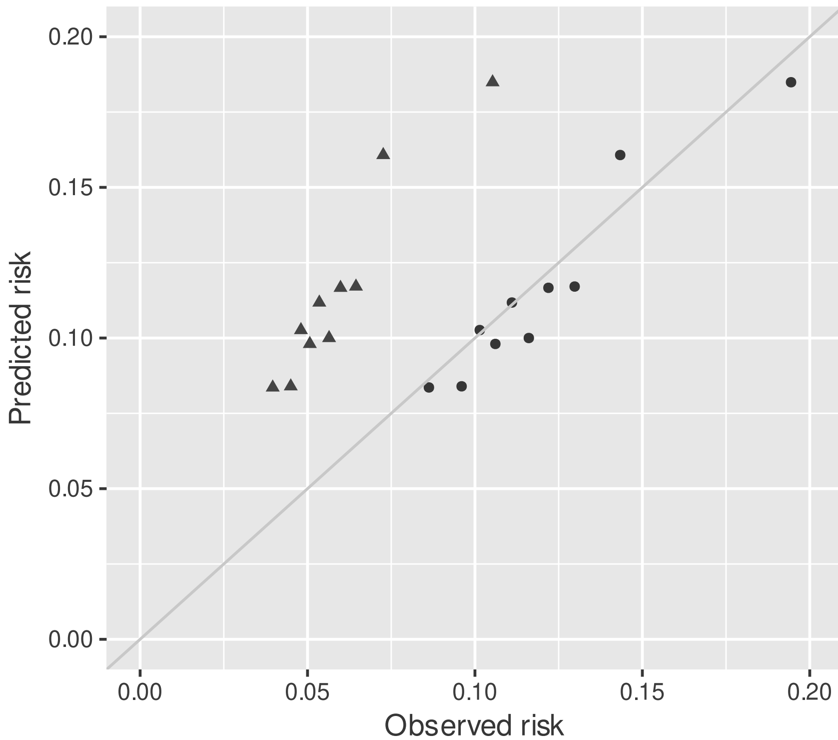


FIGURE 13.15
Replication of figure 13.22

The basic data is gathered together in a data frame; many of the individual columns built using `rep`. We then make three functions to calculate the Brier score, AUC and Spiegelhalter statistic. Though, `val.prob` does give all these results, it is not really suited for how we intend to call the functions.

```
bind_rows(data_13_13,
summarise(data_13_13,
          across(S1:S11, mean)),
```

```
summarise(data_13_13,
          across(S1:S11, ~mean(data_13_13$y) - mean(.x))),
summarise(data_13_13,
          across(S7:S11, ~Spiegel(.x, data_13_13$y))),
summarise(data_13_13,
          across(S1:S11, ~AUC(.x, data_13_13$y))),
summarise(data_13_13,
          across(S1:S11,
                 ~Brier(.x,data_13_13$y)))) -> tbl_13.13
tbl_13.13$y[is.na(tbl_13.13$y)] <- c("mean", "r-p","Spiegel", "AUC", "Bier")
tbl_13.13
```

We call these functions using `summarise`, once for each summary. The first argument for `summarise` is the data frame. But, the second argument is the function `across`. The function `across` accepts a selection of columns in the dataframe specified. There are multiple ways in which we may select columns. Here we are asking for columns from *S1* to *S11* for most summaries. The second argument for `across` is a function. The function name may be specified or a **lambda** may be specified. For the first summarisation, we use the name of the function `mean`. The result we get is what is returned by `mean` when each of the columns specified within `across` is passed to it, combined together as a vector. Rest of the summarisation use a **lambda**, a one sided function. It start with `~`. Next, we call the function as we would call it any time. However, the column that would be passed to it is referred to in the function call as `.x`. Thus, to get the difference between mean of a column from the mean of the true outcome, we use `~mean(tbl_13.13$y) - mean(.x)`. Here, the `.x` gets substituted with the columns we specify in `across`. We row bind all the summaries to the data. Next, we change the value of the y column for the summaries. As y was not one of the columns for which summaries were calculated, all the summary values for y is `NA`. We utilise this information to subset y and assign new values to them.

I couldn't find a pre made Redelmeier test. Also, our text doesn't say against what the calculated Redelmeier statistic is compared to get the p value. So, we will not calculate Redelmeier statistic.

13.7 Reclassification

Example 13.18 (page 655)

We will use `improveProb` of `Hmisc` to calculate IDI and NRI.

```
library(Hmisc)
improveProb(lg_13_10sbp$fitted.values,
            lg_13_15$fitted.values,
            lg_13_15$y) -> recal_13_8a
improveProb(lg_13_15$fitted.values,
            lg_13_10bmi$fitted.values,
            lg_13_15$y) -> recal_13_8b
```

TABLE 13.9
Replication of table 13.13

y	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11
1	1	0	1	0	0.5	0.25	0.9	0.6	0.5	0.7	0.95
1	1	0	1	0	0.5	0.25	0.9	0.6	0.45	0.65	0.7
1	1	0	1	0	0.5	0.25	0.9	0.6	0.4	0.6	0.5
1	1	0	1	0	0.5	0.25	0.9	0.6	0.35	0.5	0.4
1	1	0	1	0	0.5	0.25	0.9	0.6	0.3	0.4	0.3
1	1	0	1	0	0.5	0.25	0.9	0.6	0.25	0.3	0.1
0	1	0	0	1	0.5	0.25	0.1	0.4	0.31	0.51	0.8
0	1	0	0	1	0.5	0.25	0.1	0.4	0.26	0.41	0.4
0	1	0	0	1	0.5	0.25	0.1	0.4	0.2	0.31	0.3
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.25
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.2
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.18
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.16
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.16
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.15
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.14
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.14
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.12
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.12
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.1
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.1
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.1	0.05
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.22	0.05
0	1	0	0	1	0.5	0.25	0.1	0.4	0.1	0.25	0.01
$\bar{p}$	1	0	0.25	0.75	0.5	0.25	0.3	0.45	0.1883	0.25625	0.2658
$r - \bar{p}$	−0.75	0.25	0	−0.5	−0.25	0	−0.05	−0.2	0.0617	−0.00625	−0.0158
Spiegel							−1.63	−4	−0.5103	−1.37792	−0.0884
AUC	0.5	0.5	1	0	0.5	0.5	1	1	0.9722	0.94444	0.8148
Bier	0.75	0.25	0	1	0.25	0.19	0.01	0.16	0.1142	0.09324	0.1331

The function requires 3 vectors of equal length. The first two are predictions from the base and new models. The last is a vector denoting the observed outcome. The function prints out all the calculated indices including IDI, NRI and their confidence intervals by default. Here, we save the result and print the values corresponding to example 13.18.

```
recal_13_8a$idi
recal_13_8a$z.idi
pnorm(recal_13_8a$z.idi, lower.tail = FALSE)
recal_13_8a$idi + c(-1.96,1.96)*recal_13_8a$se.idi
```

While comparing the single model and multimodel, the score for the null hypothesis that the second model is no better than the first, its one sided p value and 95% confidence interval are similar to the textbook values, though not exact.

```
idi
[1] 0.0067989
z.idi
[1] 3.3409
z.idi
[1] 0.00041749
[1] 0.0028103 0.0107876
```

For the comparison of multiplus against multi model, the results are

```
recal_13_8b$idi
recal_13_8b$z.idi
pnorm(recal_13_8b$z.idi, lower.tail = FALSE)
recal_13_8b$idi + c(-1.96,1.96)*recal_13_8b$se.idi
```

```
idi
[1] 2.1862e-05
z.idi
[1] 0.054759
z.idi
[1] 0.47817
[1] -0.00076065  0.00080437
```

The `improveProb` doesn't calculate RIDI. However, we can calculate it with the value of IDI it returns.

```
recal_13_8a$idi /
  (mean(lg_13_10sbp$fitted.values[lg_13_10sbp$y == 1]) -
     mean(lg_13_10sbp$fitted.values[lg_13_10sbp$y == 0]))
```

```
idi
[1] 1.0079
```

Example 13.19 (page 656)

The `improveProb` also calculates NRI. Here, we check for the results in example 13.19.

```
recal_13_8a$nri
recal_13_8a$z.nri
pnorm(recal_13_8a$z.nri, lower.tail = FALSE)
recal_13_8a$nri + c(-1.96,1.96)*recal_13_8a$se.nri
```

```
nri
[1] 0.49849
z.nri
[1] 5.4641
z.nri
[1] 2.3258e-08
[1] 0.31968 0.67730
```

Example 13.20 (page 657)

We need to rework example 13.20 by hand. First, we categorise the scores according to the thresholds.

```
c(0,0.05,0.1,1) -> thresholds
cut(lg_13_10sbp$fitted.values,
    thresholds ,
    right = TRUE,
    include.lowest = TRUE) -> rsnglcat
cut(lg_13_15$fitted.values,
    thresholds,
    right = TRUE,
    include.lowest = TRUE) -> rmlticat
data_13_15$chd5 -> chd5

data.frame(rsnglcat = rsnglcat,
           rmlticat = rmlticat,
           chd = chd5) |>
  group_by( rsnglcat, rmlticat, chd) |>
  summarise(count = n()) |>
  filter(rsnglcat != rmlticat) |>
  mutate(chngdir = ifelse(as.numeric(rsnglcat) > as.numeric(rmlticat),
                          "d",
                          "u")) |>
  group_by(chngdir,chd) |>
  summarise(total = sum(count)) |>
  bind_cols(actual = rep(table(chd5),2)) |>
  mutate(prop = total /actual) -> data_13_20
```

We `cut` the fitted values of the two models we are comparing according to the thresholds given in our text. We join the results with the outcome column of original data. We, then group the resulting data frame by the categories in the two variables and by outcome and take the count in each cross classification group. Then, we filter out those rows where the categories are the same according to both risk scores as they don't contribute to the calculations. Now, we `mutate` to create a new column denoting the direction in which the multi model's score has moved the prediction. For this, we rely on the internal representation of factors, which is as integers. Thus, when the score category is higher, the internal representation of the `cut` category is a higher number. We ask R to consider the factors as number using `as.numeric`. Now, we `group_by` the score movement direction and outcome. We `sum` the counts in each group and calculate proportions. In calculating the proportions, the denominator is obtained using `table` on the outcome variable.

```
filter(data_13_20, chngdir == "u", chd == 1)$prop -> pud
filter(data_13_20, chngdir == "u", chd == 0)$prop -> puh
filter(data_13_20, chngdir == "d", chd == 1)$prop -> pdd
filter(data_13_20, chngdir == "d", chd == 0)$prop -> pdh


filter(data_13_20, chngdir == "u", chd == 1)$total -> nud
filter(data_13_20, chngdir == "u", chd == 0)$total -> nuh
filter(data_13_20, chngdir == "d", chd == 1)$total -> ndd
filter(data_13_20, chngdir == "d", chd == 0)$total -> ndh
```

We now, assign each of the four proportions and counts to four variables only to make it easier to refer to them in formulas.

```
(pud - pdd) - (puh - pdh) -> nri_13_20
nri_13_20
sqrt(((nud + ndd) / table(chd5)["1"]^2) -
       ((nud - ndd^2) / table(chd5)["1"]^3) +
       ((nuh + ndh) /table(chd5)["0"]^2) -
       ((nuh - ndh^2) / table(chd5)["0"]^3)) -> se_13_20
nri_13_20 + c(-1.96, 1.96) * se_13_20

((nud + ndd) /
    table(chd5)["1"]^2) +
  ((nuh + ndh) /
     table(chd5)["0"]^2) -> d_13_20
pnorm(nri_13_20/sqrt(d_13_20), lower.tail = FALSE)
```

The NRI, its 95% confidence interval and the p value of the test static for a zero value of NRI are calculated using the formula given in the text.

```
       1
0.021383
[1] -0.052443  0.095208
      1
0.28456
```

Our cross classification differs from that in the textbook by one or two in a cell which is the reason for the difference in the NRI calculated.

Table 13.16 (page 658)

To prepare table 13.16, we can use `ftable`.

```
ftable( rsnglcat, rmlticat,chd5,
        col.vars = c("chd5","rmlticat" ))
```

The function `ftable` outputs flat tables, where cross classification between more than two variables is better arranged. Apart from the variables that need to be cross classified, it can accept `col.vars` and / or `row.vars` to determine which all variables contribute groups as columns and which as rows. Here, we specified `col.vars` so that the two variables we specified as its argument are used to build the columns of the table.

TABLE 13.10
Replication of table 13.16

Risk using single score	Risk using multi score					
	No CHD			CHD		
rsnglcat	[0,0.05]_0	(0.05,0.1]_0	(0.1,1]_0	[0,0.05]_1	(0.05,0.1]_1	(0.1,1]_1
[0,0.05]	3439	252	17	88	10	2
(0.05,0.1]	97	92	24	4	5	1
(0.1,1]	1	3	10	0	1	3

Example 13.21 (page 659)

The `survIDINRI` package provides functions for calculating IDI and NRI for variable cohorts. However, I am not very sure if the statistic it calculates is the same as discussed in our text. So, we will rework the examples by hand. First, example 13.21.

```
coxph(Surv(survive, chd) ~ sbp,
      data = data_13_15,
      ties ="breslow",
      x = TRUE,
      y = TRUE) -> cph_sngl
coxph(Surv(survive, chd) ~ chol + sbp +smoke,
      data = data_13_15,
      ties ="breslow",
      x = TRUE,
      y = TRUE) -> cph_mlti
coxph(Surv(survive, chd) ~ chol + sbp +smoke + bmi,
      data = data_13_15,
      ties ="breslow",
      x = TRUE,
      y = TRUE) -> cph_mltp

mutate(data_13_15, survive =1827) -> data_13_15_1827
1 - predict(cph_sngl,
            newdata = data_13_15_1827,
            type = "survival") -> data_13_15$crsngl
1 - predict(cph_mlti,
            newdata = data_13_15_1827,
            type = "survival") -> data_13_15$crmlti
1 - predict(cph_mltp,
            newdata = data_13_15_1827,
            type = "survival") -> data_13_15$crmltp
```

```
summary(survfit(Surv(survive, chd5) ~ 1,
                data = data_13_15),
        times =1827)$surv -> surv_13_15

(var(data_13_15$crmlti) -
    var(data_13_15$crsngl)) /
  (surv_13_15 * (1 -surv_13_15)) -> idi_sm
(var(data_13_15$crmltp) -
    var(data_13_15$crmlti)) /
  (surv_13_15 * (1 -surv_13_15)) -> idi_mm
(var(data_13_15$crmlti) -
    var(data_13_15$crsngl)) /
  var(data_13_15$crsngl) -> ridi_sm
idi_sm
idi_mm
ridi_sm
```

First, we fit the Cox proportional hazards model, which we feed to `predict`. The `newdata` that we provide `predict` is the original data frame which is mutated to change the value of *survive* to 1827 in all rows. The call to `mutate` is made from inside the call to `predict`. We ask `predict` to return survival probability, subtract it from 1 to obtain the probability of event and add those values as new columns to the original data frame. We use the `summary` method for `survfit` to obtain the KM estimate of survival. We use `var` to obtain the variance of the predicted risk. The IDI for both comparisons and RIDI for the single-multi comparison is calculated by substituting the equations given in the textbook with the appropriate values.

```
[1] 0.0075616
[1] 0.00049983
[1] 0.90821
```

Example 13.22 (page 660)

We now turn to example 13.22.

```
mutate(data_13_15,
       dir = ifelse(crsngl >= crmlti, "d", "u")) -> data_13_15

table(data_13_15$dir)["u"] /nrow(data_13_15) -> propu
table(data_13_15$dir)["d"] /nrow(data_13_15) -> propd

1 - summary(survfit(Surv(survive, chd5) ~ 1,
                    data = data_13_15),
            times =1827)$surv -> kmra_13_15
1 - summary(survfit(Surv(survive, chd5) ~ 1,
                    data = data_13_15,
                    subset = dir == "u"),
            times =1827)$surv -> kmru_13.15
1 - summary(survfit(Surv(survive, chd5) ~ 1,
```

```
                    data = data_13_15,
                    subset = dir == "d"),
            times =1827)$surv -> kmrd_13_15

((kmru_13.15 * propu) - (kmrd_13_15 * propd)) / kmra_13_15 +
 (((1 - kmrd_13_15) * propd) - ((1 - kmru_13.15) * propu)) /
  (1 -kmra_13_15) -> nri_13_22
nri_13_22
```

Our first step is to create a category variable to show the direction of change between the risk scores. Next, we summarise the numbers in the two categories using `table` and find their proportions. Next, we find the KM estimate for the entire data and subsets with only one of the two directions of change. We then substitute the values in the formula given in the textbook.

```
      u
0.53121
```

We now will rework the NRI calculated according to the clinical cutoffs.

```
c(0,0.05,0.1,1) -> cutoffs
cut(data_13_15$crsngl,
    cutoffs,
    right = TRUE,
    include.lowest = TRUE) -> data_13_15$cat_sngl
cut(data_13_15$crmlti,
    cutoffs,
    right = TRUE,
    include.lowest = TRUE) -> data_13_15$cat_mlti
cut(data_13_15$crmltp,
    cutoffs,
    right = TRUE,
    include.lowest = TRUE) -> data_13_15$cat_mltp

mutate(data_13_15,
       codir = case_when(
         as.numeric(cat_sngl) > as.numeric(cat_mlti) ~ "d",
         as.numeric(cat_sngl) < as.numeric(cat_mlti) ~ "u",
         TRUE ~ "e")) -> data_13_15

table(data_13_15$codir)["u"]/nrow(data_13_15) -> propcou
table(data_13_15$codir)["d"]/nrow(data_13_15) -> propcod

1 - summary(survfit(Surv(survive, chd5) ~ 1,
                    data = data_13_15,
                    subset = codir == "u"),
            times =1827)$surv -> kmrcou_13_15
1 - summary(survfit(Surv(survive, chd5) ~ 1,
```

```
                    data = data_13_15,
                    subset = codir == "d"),
            times =1827)$surv -> kmrcod_13_15

((kmrcou_13_15 * propcou) - (kmrcod_13_15 * propcod)) /
  kmra_13_15 + (((1 - kmrcod_13_15) * propcod) -
                  ((1 - kmrcou_13_15) * propcou)) /
  (1 -kmra_13_15) -> nrico_13_22
nrico_13_22
```

Our first step is to `cut` the risk predictions into categories according to the clinical cut offs. Next, we create a new column to store the direction of change. For this, we use `case_when` inside `mutate`. As with our earlier example, we rely on the internal representation of factors to do the comparison. We use `table` to calculate the proportion of cases with upward or downward movement in prediction categories. We use `summary` of `survfit` to obtain the KM estimate of risk for each of the two categories. Finally, we substitute the appropriate values in the formula given in the text.

```
       u
0.023155
```

Table 13.17 (page 661)

We will now try to recreate table 13.17.

```
bind_cols(event = rep(c("Overall", "CHD", "No CHD"),
                     each = 3) ,
          bind_rows(data_13_15 |>
                     group_by(cat_sngl, cat_mlti) |>
                     summarise(count = n()) |>
                     pivot_wider(names_from = cat_mlti,
                                 values_from = count),
                   filter(data_13_15, chd == 1) |>
                     group_by(cat_sngl, cat_mlti) |>
                     summarise(count = n()) |>
                     pivot_wider(names_from = cat_mlti,
                                 values_from = count),
                   filter(data_13_15, chd == 0) |>
                     group_by(cat_sngl, cat_mlti) |>
                     summarise(count = n()) |>
                     pivot_wider(names_from = cat_mlti,
                                 values_from = count)))
```

We are using `bind_cols` to bind together a character vector with the result given by `bind_rows`. The character vector is made by repeating each of the three labels thrice. The `bind_row` binds the rows returned by three pipe flows. In each of the pipe flows, an appropriate data frame is grouped by the two score categories, their counts taken and then converted into a wide format. The final result is table 13.17.

TABLE 13.11
Replication of table 13.17

		Risk using multi score		
Event	Risk using single score	[0,0.05]	(0.05,0.1]	(0.1,1]
Overall	[0,0.05]	3522	240	8
Overall	(0.05,0.1]	105	123	24
Overall	(0.1,1]	2	6	19
CHD	[0,0.05]	145	21	2
CHD	(0.05,0.1]	8	10	3
CHD	(0.1,1]		3	4
No CHD	[0,0.05]	3377	219	6
No CHD	(0.05,0.1]	97	113	21
No CHD	(0.1,1]	2	3	15

We will leave this section by mentioning two packages that implement cross validation in R. One is the `caret` package. The package `rms` also implements cross validation and boot strapped calibration for common regression models.

We will now turn to example 13.23.

13.8 Presentation of risk scores

Example 13.23 (page 665)

```
1.3 -> bsmoke
2.5 -> bsex
-5.6 -> intercept
rep(c("female", "male"),each =2) -> sex
rep(c("non-smoker", "smoker"),2) -> smoke
bind_cols(sex = sex,
          smoke = smoke,
          rowlabel = paste0(sex,", ", smoke)) |>
  mutate( x1 = ifelse(sex == "female",0,1),
          b1 = bsex,
          x2 = ifelse(smoke == "smoker",1,0),
          b2 = bsmoke,
          bixi = b1*x1 + b2*x2,
          risk = logit2r(bixi + intercept),
          sexpts = round(b1*x1 / bsmoke),
          smokepts = round(b2*x2 / bsmoke),
          totpts= sexpts + smokepts,
          riskpoints = logit2r((totpts * bsmoke) + intercept)) |>
  select(-c("sex", "smoke"))
```

After storing the relevant data in variables, we use `bind_cols` to build the starting columns of the data frame. From these columns, we build the row labels and numerical values of `x1` and `x2`. We multiply the relevant regression coefficients with `x1` and `x2`, add them together and with the value of intercept and feed it to *logit2r* that we defined in an earlier example to get the exact risk. We divide the regression coefficients with the standard regression units that we choose, round it to nearest integer and add intercept with it before feeding to *logit2r*. Finally, we negative `select` the columns to exclude from display.

```
# A tibble: 4 x 11
  rowlabel         x1    b1    x2    b2  bixi    risk sexpts smokepts totpts
  <chr>         <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl>  <dbl>    <dbl>  <dbl>
1 female, non-~     0   2.5     0   1.3   0   0.00368      0        0      0
2 female, smok~     0   2.5     1   1.3   1.3 0.0134       0        1      1
3 male, non-sm~     1   2.5     0   1.3   2.5 0.0431       2        0      2
4 male, smoker      1   2.5     1   1.3   3.8 0.142        2        1      3
# i 1 more variable: riskpoints <dbl>
```

Table 13.20 (page 668)

Now, we will rework example 13.24. First, we will recreate table 13.20.

```
c("female", "male") -> sex
c("40-49", "50-59", "60-69", "70-79") -> agegrp
c(0.52, 0.48) -> sexprop
c(0.3,0.25,0.25,0.2) -> ageprop
c(0,1.8) -> sexbeta
c(0,0.7,2.1,3.1) -> agebeta
sum(agebeta *  ageprop) + sum(sexbeta * sexprop) -> baserisk
0.98 -> basesurv
bind_cols(sex = rep(sex, each =4),
          agegrp = rep(agegrp,2),
          sexbeta = rep(sexbeta, each = 4),
          agebeta = rep(agebeta,2)) |>
  mutate(bx = sexbeta + agebeta,
         w = bx - baserisk,
         `0.98z` = basesurv ^ exp(w),
         risk = 1 -`0.98z`) |>
  select(-c("sexbeta", "agebeta"))
```

This is similar to the previous example. We bind together an initial data frame from the beta coefficients for each value of the variables along with those values. We then `mutate` to produce each of the columns in table 13.20. A difference from the previous example is that we calculate the risk directly rather than by using a custom function.

TABLE 13.12
Replication of table 13.20

Sex	Age (yrs)	$\sum bx$	w	0.98^z	Risk
female	40-49	0.0	−2.184	0.99773	0.0022720
female	50-59	0.7	−1.484	0.99543	0.0045701
female	60-69	2.1	−0.084	0.98160	0.0184035
female	70-79	3.1	0.916	0.95076	0.0492385
male	40-49	1.8	−0.384	0.98633	0.0136665
male	50-59	2.5	0.316	0.97267	0.0273302
male	60-69	3.9	1.716	0.89371	0.1062884
male	70-79	4.9	2.716	0.73678	0.2632151

Table 13.21 (page 669)

To make table 13.21, we follow a similar path.

```
sexbeta[2] -> baseru
bind_cols(sex = rep(sex, each =4),
          agegrp = rep(agegrp,2),
          sexbeta = rep(sexbeta, each = 4),
          agebeta = rep(agebeta,2)) |>
  mutate(sexpts = round(sexbeta / baseru),
         agepts = round(agebeta / baseru),
         totpts = sexpts + agepts,
         bxsum = totpts * baseru,
         w = bxsum - baserisk,
         `0.98z` = basesurv ^ exp(w),
         risk = 1 -`0.98z`) |>
  select(-c("sexbeta", "agebeta"))
```

The difference is that we use the points derived from the coefficients by dividing them with the standard regression units we choose to calculate the risk instead of the coefficient themselves.

TABLE 13.13
Replication of table 13.21

Sex	Age (years)	Sex points	Age points	Total points	Est$(\sum bx)^a$	w	0.98^z	Risk
female	40-49	0	0	0	0.0	−2.184	0.99773	0.002272
female	50-59	0	0	0	0.0	−2.184	0.99773	0.002272
female	60-69	0	1	1	1.8	−0.384	0.98633	0.013666
female	70-79	0	2	2	3.6	1.416	0.92012	0.079876
male	40-49	1	0	1	1.8	−0.384	0.98633	0.013666
male	50-59	1	0	1	1.8	−0.384	0.98633	0.013666
male	60-69	1	1	2	3.6	1.416	0.92012	0.079876
male	70-79	1	2	3	5.4	3.216	0.60434	0.395659

Table 13.23 (page 670)

We now move on to re-create table 13.23.

```
bind_rows(bind_cols(term = agegrp,
                    value = seq(45,75, by =10),
                    coef = 0.107, ref =  45),
          bind_cols(term = sex,
                    value = c(0,1),
                    coef = 1.8, ref = 0)) |>
  mutate(`c-ref` = value - ref,
         `b*d` = coef * `c-ref`,
         points = round(`b*d` / (5 * 0.107)))
```

Here, we create the rows for sex and age groups separately and then join them together using `bind_rows`. Then, we `mutate` the data frame to make the new columns.

TABLE 13.14
Replication of table 13.23

Term	Value of concentration (c)	Estimated beta coeff. (b)	Reference	$c-$ reference (d)	$(b)\times(d)$	Points
40-49	45	0.107	45	0	0.00	0
50-59	55	0.107	45	10	1.07	2
60-69	65	0.107	45	20	2.14	4
70-79	75	0.107	45	30	3.21	6
female	0	1.800	0	0	0.00	0
male	1	1.800	0	1	1.80	3

Table 13.24 (page 671)

We follow the steps in our textbook to create table 13.24.

```
sum( c(0.107,1.8) * c(54.63, 0.48)) -> v
mutate(data.frame(points = 0:9),
       risk = 1 - 0.968 ^ exp(0.535 * points + (0.107 * 45 - v)))
```

We define the starting data frame inside `mutate` to contain only one column, a sequence of points from zero to nine. We use `mutate` to calculate the column of risk according to the formula given in the textbook.

Example 13.26 (page 672)

For the final example 13.26, as we know the textbook method, we will explore the R way using the package `rms`. Remember to install it using `install.packages` as described in chapter 1.

TABLE 13.15
Replication of table 13.24

Points	Risk
0	0.0048798
1	0.0083176
2	0.0141600
3	0.0240562
4	0.0407244
5	0.0685291
6	0.1141541
7	0.1869508
8	0.2976896
9	0.4530390

```
library(rms)
datadist(data_13_15) -> dd_13_15
options(datadist = dd_13_15)

cph(Surv(survive, chd) ~ chol + sbp +smoke,
    x = TRUE,
    y= TRUE,
    surv =TRUE,
    data = data_13_15) -> cph_13_15
lp2rsk <- function(lp){1 - survest(cph_13_15,times = 1827)$surv ^ exp(lp)}
```

The first steps are essential pre-requisites. We need to feed the data frame we are planning to work on to `datadist` and the resulting object should be specified as the value of `datadist` argument to `options`. This enables `rms` to do certain preparatory works like creating summaries of different variables. Next, we fit the model for which we want a point score estimated to be fitted using one of the functions of `rms`. Here, we are using `cph` to fit a Cox proportional model. The arguments to `cph` is similar to `coxph` we saw earlier. In this fit, we specify `x=TRUE` and `y=TRUE` to store the expanded design matrix of the model and the response object as part of the result returned. This enables / makes easier further calculations using the model object. Next, we construct a custom function, that will return the predicted risks for a given linear predictor. Note that in our function, we don't deduct the mean combination of coefficients. This deduction is already included in the linear predictor returned by `rms`.

```
nomogram(cph_13_15,
         fun = lp2rsk,
         fun.at = c(0.01, 0.02,0.04,0.08, 0.16, 0.32),
         maxscale = 24) -> ng_13_15
plot(ng_13_15)
```

Now, we feed `nomogram`, the function that calculates the points, its arguments. The first argument is the model object and this is the only one absolutely needed. Our custom function is given as `fun`. This is used for converting the linear predictor variable to whatever we want. In our case, we are using the custom function to convert the linear predictor to predicted risk. When we provide a `fun`, an additional axis to represent the values returned by the specified function is made. The argument `fun.at` determines the tick values for that axis. The argument `maxscale` determines the maximum points that can be assigned to any one variable. We store the result given by `nomogram`, which we may print or `plot`. We get a beautiful nomogram if we `plot` and the points assigned to each variable levels if we print it. Note that the points calculated by `nomogram` won't necessarily agree with the textbook scoring system, but should agree with the approximate risk. We can categorise the continuous variables into categories as given in the textbook and fit survival model using `cph` and proceed to make another nomogram. I will leave it to you to determine which would be a better scoring system.

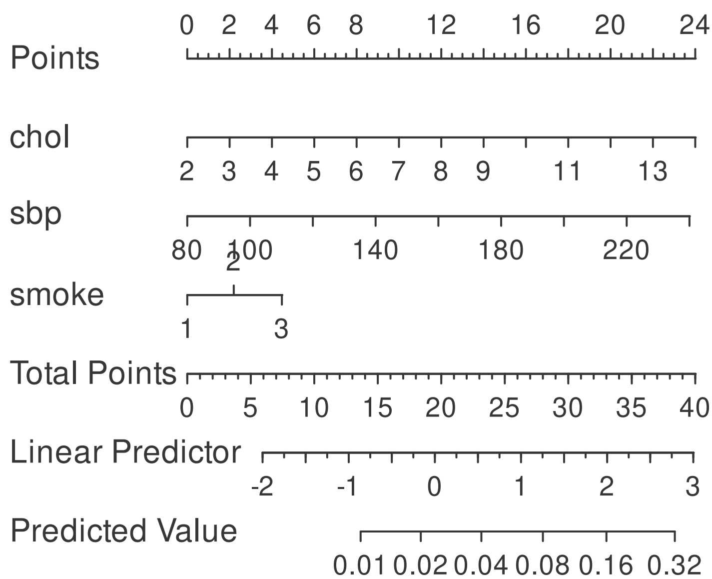


FIGURE 13.16
Nomogram for continuous values in example 13.26 prepared using rms

Before we leave this chapter, I want to point out a function in `rms` named `Function`.

```
cphfun_13.15 <- Function(cph_13_15)
cphfun_13.15(smoke = "1", chol = 6.1, sbp = 150)
```

It accepts a model object returned by `rms` and outputs a function, which we should store. This result can be called with our choice of values for explanatory variables used in fitting the model that is fed to `Function` and returns the linear predictor according to that model. Here, we feed the Cox proportional hazards model we fitted to `Function` and then call the result of `Function` with our choice of values for *chol*, *sbp* and *smoke* to get the linear predictor.

There are many more such functions that return a function to calculate some value from a fitted model. One such is `Survival` which will return a function that will accept `times` and `lp` argument and return the survival calculated according to the model that was fed to `Survival`.

```
surv_13_15 <- Survival(cph_13_15)
surv_13_15(times = 1827, lp =cphfun_13.15(smoke = "1",
                                        chol = 6.1,
                                        sbp = 150) )
```

13.9 Recap

13.9.1 Commands introduced in this chapter

- ggplot2::scale_y_log10
- ggplot2::lims
- ggplot2::coord_cartesian
- stats::addmargins
- purrr::map_int
- stats::dnorm
- base::names
- DescTools::Rev
- epiR::epi.tests
- ROCit::rocit
- ROCit::ciAUC
- pROC::roc
- pROC::roc.test
- graphics::lines
- graphics::legend
- survival::concordance
- effsize::hedges_g
- purrr::map_chr
- rms::val.prob
- DescTools::HosmerLemeshowTest
- riskRegression::predictRisk
- ggplot2::geom_abline
- ggplot2::scale_x_discrete
- Hmisc::improveProb
- stats::ftable
- dplyr::case_when
- rms::datadist
- rms::cph
- rms::nomogram
- rms::Function
- rms::Survival

14

Computer-intensive methods

We will use the package `boot` to perform bootstrap analysis, `coin` for permutation tests and `VIM` & `mice` for multiple imputation. In addition to `tidyverse` packages, we use `lmtest`, `sandwich` and `DescTools`. As the results are dependent on randomisation, we do not expect the results to agree exactly with that of the textbook. We will rework the examples in section 14.2.1

14.1 The bootstrap

Section 14.2.1 (page 681)

```
library(tidyverse)
library(boot)
read_table("K11828 supplements/Datasets/Table 2.10.DAT",
           col_names = c("chol", "dbp","sbp","alcohol",
                         "cig","co","cotinine", "chd")) -> tbl_2_10
meanCust <- function(data,index){mean(data[index])}
boot(tbl_2_10$chol, meanCust,R = 10000) -> b_mnchol
```

After importing the data, we define a custom function to calculate the mean. A custom function is needed because `boot` expects the function that is supplied to it to handle at least two arguments – a vector of values and a vector of indices, weights or frequencies. However, `mean` doesn't expect a vector of indices; hence, the need for a custom function. All that the custom function *meanCust* does is call `mean` using the index vector to subset the values. Now, we call `boot` with our cholesterol data, the custom function and the number of replicates required. The function `boot` will pass the cholesterol data, with the index vector specifying the random sample from the data to *meanCust* 10,000 times and collect the returned value. Printing the boot object will print the original statistic, the bias in the bootstrap estimate and its standard error. Here, we are interested in the graphs in figure 14.2 to 14.5. Passing the boot object to `plot` will print the histogram and qq plot. However, to replicate figure 14.2, we need to plot two histograms, one over the other.

Figure 14.2 (page 682)

The component `t` in the boot object contains the bootstrap replicates of the statistic. We pass it to `ggplot` as a data frame to prepare the graph in figure 14.2.

DOI: 10.1201/9781003589563-14

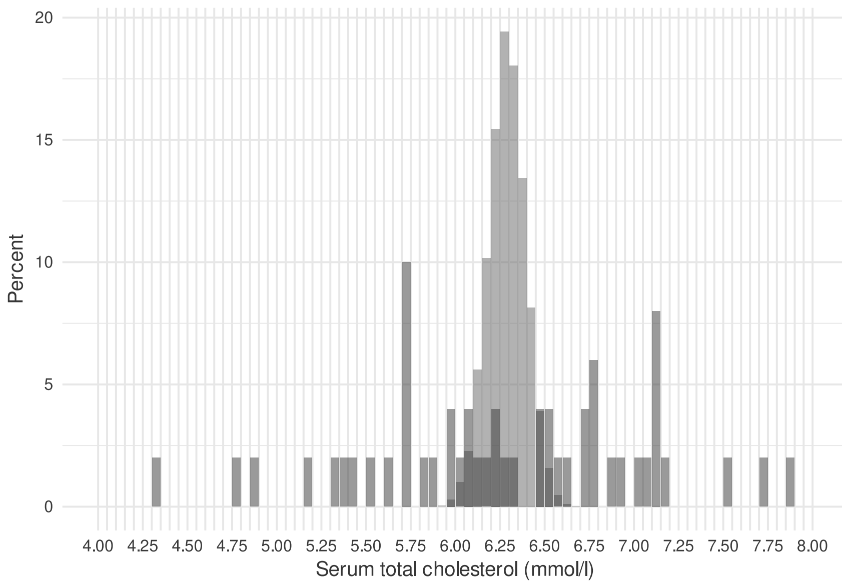


FIGURE 14.1
Replication of figure 14.2

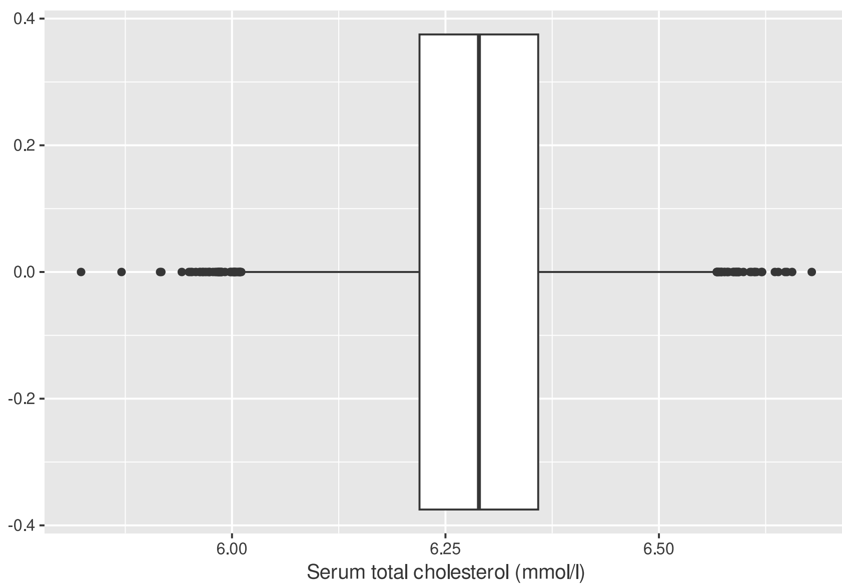


FIGURE 14.2
Replication of figure 14.3

```
ggplot() +
  geom_bar(data = data.frame(bmean = b_mnchol$t),
                           aes(x = bmean,y=..prop..* 100),
                           fill = "blue", alpha = 0.5) +
  geom_bar(data = tbl_2_10,
```

```
         aes(x = chol,y = ..prop.. * 100),
         alpha = 0.5, fill = "black") +
scale_x_binned(limits = c(4,8),
               n.breaks = 80,
               nice.breaks = TRUE,
               guide = guide_axis(check.overlap = TRUE)) +
xlab("Serum total cholesterol (mmol/l)") +
ylab("Percent") +
theme_minimal()
```

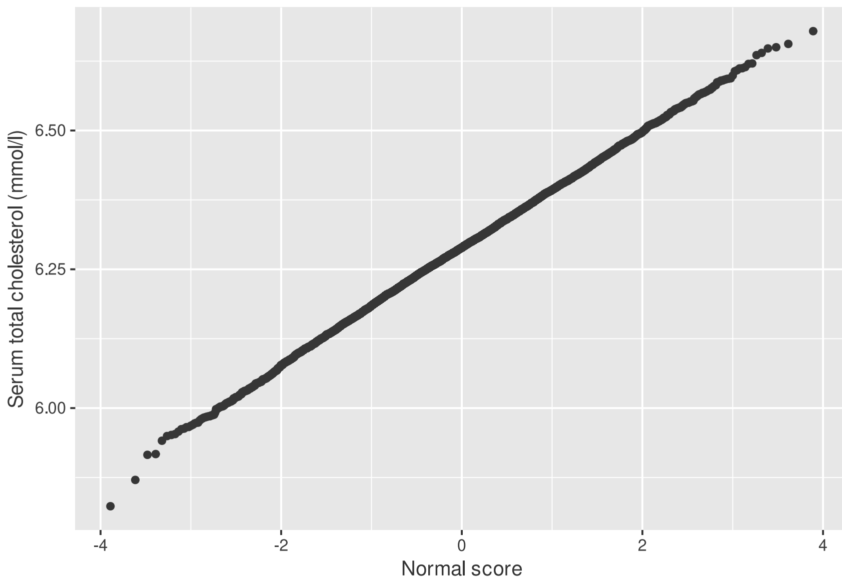


FIGURE 14.3
Replication of figure 14.4

We don't pass any arguments to `ggplot` as the two histograms we plan to draw have different `data`. We don't use `geom_histogram` as we need to scale the y axis to use percentages rather than counts. We use `geom_bars`; however, we don't have a `y` in the data frame supplied to each of the geoms. The `y` is supplied by `stat_count` using the specifications given to `scale_x_binned` which categorises the continuous variable according to our requirements and returns its count. But, we don't use this count directly. Instead, we modify it inside `aes`. The count calculated using `scale_x_binned` is available inside the `aes` as `..count..` or as `..prop..`. We choose `..prop..` and multiply it with 100 to obtain the percentage. We specify `alpha` inside the geoms to have a see through effect where the bars overlap. The `nice.breaks` argument for `scale_x_binned` permits it to use better looking cut points instead of exact cut points for categorisation. We supply `guide_axis` with its argument `check.overlap` set as `TRUE` as the value of `guide` of `scale_x_binned` to remove overlapping labels.

Figures 14.3 and 14.4 (page 682 and 683)

We may use the same data to prepare the boxplot and qqplot.

```
ggplot(data.frame(bmean = b_mnchol$t)) +
  geom_boxplot(aes(x = bmean))
ggplot(data.frame(bmean = b_mnchol$t)) +
  geom_qq(aes(sample = bmean))
```

Figure 14.5 (page 684)

To prepare the graph of figure 14.5, we need another custom function to return the inter quartile range.

```
iqrCust <- function(data,index) {IQR(data[index])}

boot(tbl_2_10$chol, meanCust,R = 10000) -> b_mnchol
boot(tbl_2_10$alcohol, meanCust,R = 10000) -> b_mnalc
boot(tbl_2_10$cotinine, meanCust,R = 10000) -> b_mncot
boot(tbl_2_10$chol, iqrCust,R = 10000) -> b_iqrchol
boot(tbl_2_10$alcohol, iqrCust,R = 10000) -> b_iqralc
boot(tbl_2_10$cotinine, iqrCust,R = 10000) -> b_iqrcot

par(mfrow = c(2,3))
hist(b_mnchol$t,
     breaks = 30,
     xlab = "Serum total cholesterol(mmol/L)",
     ylab = "Frequency",
     main = NULL)
hist(b_mnalc$t,
     breaks = 60,
     xlab = "Alcohol (g/day)",
     ylab = "Frequency",
     main = NULL)
hist(b_mncot$t,
     breaks = 30,
     xlab = "Cotinine (ng/ml)",
     ylab = "Frequency",
     main = NULL)
hist(b_iqrchol$t,
     breaks = 30,
     xlab = "Serum total cholesterol(mmol/L)",
     ylab = "Frequency",
     main = NULL)
hist(b_iqralc$t,
     breaks = 60,
     xlab = "Alcohol (g/day)",
     ylab = "Frequency",
     main = NULL)
hist(b_iqrcot$t,
     breaks = 30,
     xlab = "Cotinine (ng/ml)",
```

```
     ylab = "Frequency",
     main = NULL)
```

We feed `boot` with the appropriate custom function and data and store the results. We feed `hist` with the `t` component of the appropriate boot object. We supply `hist` with other arguments like `xlab`, `ylab`, `breaks` and `main`. The `breaks` determine the cut offs for the bins, `main` is used for title. As we don't want a title for the individual graphs, we set it to `NULL`. The `par(mfrow = c(2,3))` is used before calling `hist` to instruct R to combine the next plots into one with 2 rows each with 3 columns. As we have 6 graphs to plot, we call `hist` 6 times.

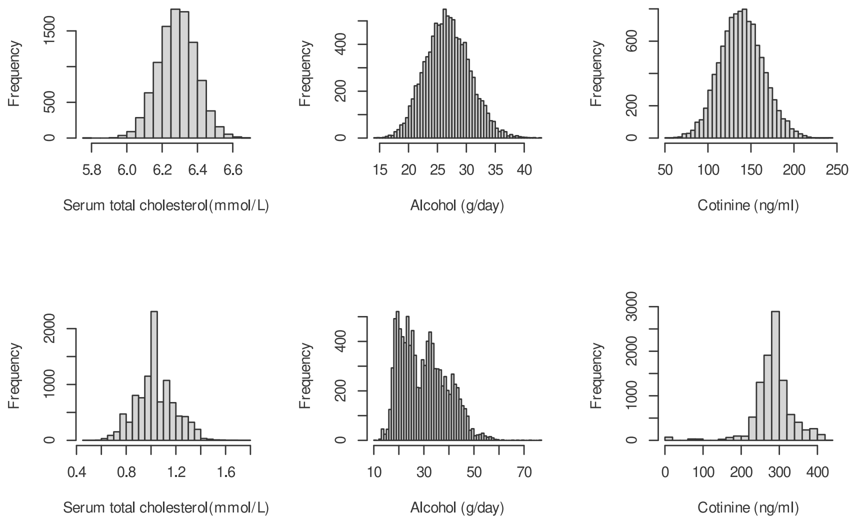


FIGURE 14.4
Replication of figure 14.5

Example 14.1 (page 685)

To obtain bootstrap confidence intervals, we use `boot.ci`, which accepts a boot object as its argument. First, bootstrap normal intervals.

```
boot.ci(b_mnchol, type = "norm")
```

All we have to do is feed the boot object to `boot.ci` and specify the type of interval required. Here we use `type = "norm"` to get the bootstrap normal interval.

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 10000 bootstrap replicates

CALL :
boot.ci(boot.out = b_mnchol, type = "norm")

Intervals :
```

```
Level      Normal
95%   ( 6.078,  6.496 )
Calculations and Intervals on Original Scale
```

Example 14.2 (page 687)

To obtain the percentile intervals, we need to say `type = "perc"`.

```
boot.ci(b_iqralc, type ="perc")
```

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 10000 bootstrap replicates

CALL :
boot.ci(boot.out = b_iqralc, type = "perc")

Intervals :
Level     Percentile
95%   (17.02, 48.05 )
Calculations and Intervals on Original Scale
```

Figures 14.6 and 14.7 (page 687)

The boxplot and normal plot for the bootstrap sample of alcohol IQR is easily made.

```
boxplot(b_iqralc$t,
        horizontal = TRUE,
        xlab = "Alcohol (g/day)")

qqnorm(b_iqralc$t,
       xlab = "Normal Score",
       ylab = "Alcohol (g/day)")
```

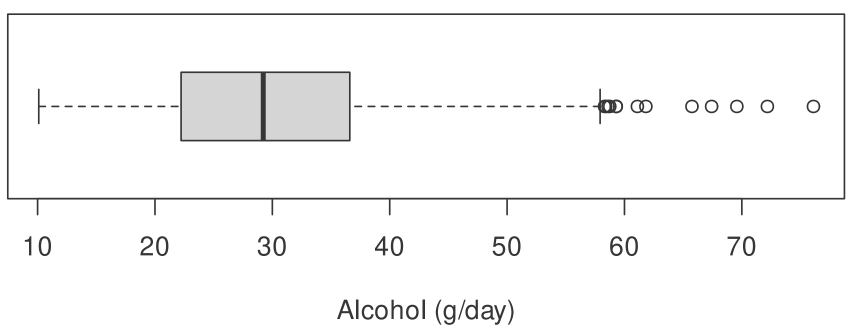


FIGURE 14.5
Replication of figure 14.6

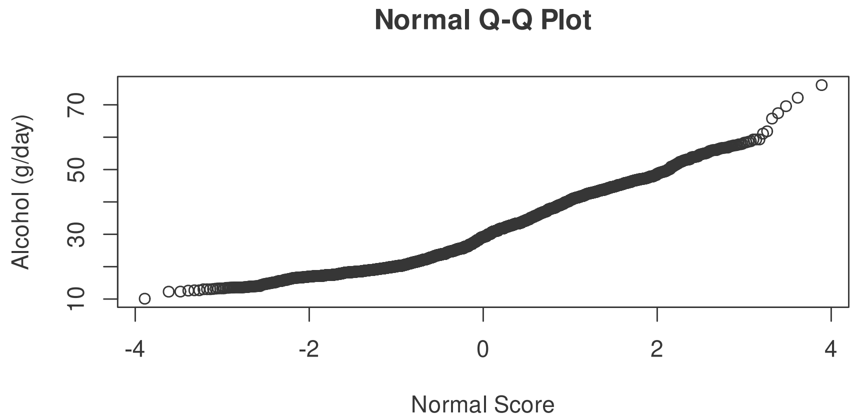


FIGURE 14.6
Replication of figure 14.7

Example 14.3 (page 689)

boot.ci doesn't provide bias corrected confidence intervals. However, we may calculate it from values given by the percentile method (or even directly). Here, we will use the *b_iqralc* object returned by boot in the earlier example.

```
qnorm(mean(b_iqralc$t <= b_iqralc$t0)) -> biascorr
quantile(b_iqralc$t, c(pnorm((2 * biascorr) + qnorm(0.025)),
                      pnorm((2 * biascorr) + qnorm(0.975))))
```

We use qnorm to calculate the bias correction. To provide the proportion of bootstrap values that are less than or equal to the sample value, we use mean relying on implicit conversion of the logical condition that is supplied to it. The logical condition checks if each of the value of t component of the boot object is less than or equal to the t0 object of the boot object. t is a vector with result of each iteration of bootstrap and t0 the result for the entire sample. qnorm is the inverse normal function which gives the quantile for a given probability.

We add to twice the bias correction the appropriate quantiles for the confidence interval, convert those to probabilities using pnorm and collect them as a vector. We supply to quantile, the t component of the boot object and the vector we made asking it to return values from the t corresponding to the given probabilities.

```
 3.55059% 98.27648%
   17.400    49.792
```

Example 14.4 (page 691)

boot.ci does give the bias corrected and accelerated interval.

```
boot.ci(b_iqralc, type ="bca")
```

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 10000 bootstrap replicates

CALL :
boot.ci(boot.out = b_iqralc, type = "bca")

Intervals :
Level       BCa
95%   (17.45, 51.24 )
Calculations and Intervals on Original Scale
```

Table 14.1 (page 686)

The `boot.ci` returns all the different types of confidence intervals if we don't specify a `type` or if we specify `type ="all"`. We use it to build table 14.1.

```
confint.str <- function(x) {paste0("(",
                                   round(x[1],2),
                                   ", " ,
                                   round(x[2],2), ")")}

build_bcistr <- function(bo){
  b <- boot.ci(bo)
  c(Normal = confint.str(b$normal[2:3]),
    Percentile = confint.str(b$per[4:5]),
    Basic = confint.str(b$basic[4:5]),
    BCa = confint.str(b$bca[4:5]))}
```

First, we define two custom functions. The first one combines after rounding, the first two elements of a numeric vector fed to it, presumably confidence limits, into a comma-separated string enclosed in parentheses. The second function *build_bcistr* accepts a boot object, passes it to `boot.ci` and collects the different confidence intervals and passes them to the first custom function *confint.str*. The returned strings are collected in an appropriately labelled vector.

```
n <- nrow(tbl_2_10)

bind_rows(list(label = "Observed PE",
               cholmean = as.character(round(mean(tbl_2_10$chol),2)),
               alcmean = as.character(round(mean(tbl_2_10$alcohol),2)),
               cotmean = as.character(round(mean(tbl_2_10$cotinine),2)),
               choliqr = as.character(round(IQR(tbl_2_10$chol),2)),
               alciqr = as.character(round(IQR(tbl_2_10$alcohol),2)),
               cotiqr = as.character(round(IQR(tbl_2_10$cotinine),2))),
          list(label = "Observed CI",
               cholmean = confint.str(mean(tbl_2_10$chol) + c(
                 (qt(0.025,df = n-1) *sd(tbl_2_10$chol) /sqrt(n)),
                 (qt(0.025,lower.tail = FALSE,df = n-1) *
                    sd(tbl_2_10$chol) /sqrt(n)))),
```

```
                alcmean =confint.str(mean(tbl_2_10$alcohol) + c(
                  (qt(0.025,df = n -1) * sd(tbl_2_10$alcohol) / sqrt(n)),
                  (qt(0.025,lower.tail = FALSE,df = n-1) *
                     sd(tbl_2_10$alcohol) / sqrt(n)))),
                cotmean = confint.str(mean(tbl_2_10$cotinine) + c(
                  (qt(0.025,df = n -1) *sd(tbl_2_10$cotinine) / sqrt(n)),
                  (qt(0.025,lower.tail = FALSE,df = n-1) *
                     sd(tbl_2_10$cotinine) / sqrt(n))))),
           rownames_to_column(
             data.frame(
               lapply(
                 list(cholmean = b_mnchol,
                      alcmean = b_mnalc,
                      cotmean = b_mncot,
                      choliqr = b_iqrchol,
                      alciqr = b_iqralc,
                      cotiqr = b_iqrcot),
                 build_bcistr)),
             var = "label"))
```

We now use `bind_rows` to bind three different lists. The first one is created using `list` with values of the observed means rounded to two digits and converted to strings using `as.character`. The second is also created using `list`. Each of its component is a string returned by *confint.str*, our custom function to join together confidence limits. We calculate the confidence limits by adding to the mean, the appropriate multiple of the standard error of the variable. The appropriate multiple is provided by `qt` which accepts the probability and degrees of freedom. Based on whether `lower.tail = FALSE` or not, `qt` returns the upper tail value or lower tail value. The lower tail value will have a negative sign and hence will be deducted from the mean.

The third list is a data frame. It is built by passing to `row_names_to_column`, a dataframe made from the result returned by `lapply` which passes each component of its first argument to the function specified as its second argument. Here, we give it a list of all the boot objects we prepared earlier and the custom function *build_bcistr*. The result is a dataframe with its column *label* derived from the row names of the dataframe made from `lapply`'s result.

TABLE 14.1
Replication of table 14.1

	Means			Interquartile ranges		
Method	Cholesterol (mmol/l)	Alcohol (g/day)	Cotinine (ng/ml)	Cholesterol (mmol/l)	Alcohol (g/day)	Cotinine (ng/ml)
Observed PE	6.29	26.76	139.46	1.03	30.05	283.75
Observed CI	(6.07, 6.5)	(18.88, 34.65)	(89.08, 189.84)			
Normal	(6.08, 6.5)	(19.23, 34.27)	(90.92, 188.53)	(0.71, 1.32)	(12.11, 48.08)	(184.92, 380.85)
Percentile	(6.08, 6.5)	(19.72, 34.57)	(92.1, 190.24)	(0.75, 1.36)	(17.02, 48.05)	(189, 388)
Basic	(6.08, 6.5)	(18.96, 33.81)	(88.68, 186.82)	(0.69, 1.3)	(12.05, 43.08)	(179.5, 378.5)
BCa	(6.07, 6.49)	(20.21, 35.36)	(94.93, 193.22)	(0.74, 1.35)	(17.45, 51.24)	(219.75, 395)

14.2 Practical issues when bootstrapping

Table 14.2 (page 694)

To prepare table 14.2, we need another custom function as we need to call `boot` with different values for replication.

```
bootCustom <- function(v,f,r) {
  boots <- lapply(r, boot, data = v, statistic = f)
  lapply(boots, build_bcistr)}

c(50, 100, 500, 1000, 5000, 10000, 50000,100000) -> rpts

do.call(rbind,
        bootCustom(v = tbl_2_10$chol,
                   f =  meanCust, r = rpts) ) -> mnchollist
do.call(rbind,
        bootCustom(v = tbl_2_10$alcohol,
                   f =  meanCust, r = rpts) ) -> mnalclist
do.call(rbind,
        bootCustom(v = tbl_2_10$cotinine,
                   f =  meanCust, r = rpts) ) -> mncotlist
do.call(rbind,
        bootCustom(v = tbl_2_10$chol,
                   f =  iqrCust, r = rpts) ) -> iqrchollist
do.call(rbind,
        bootCustom(v = tbl_2_10$alcohol,
                   f =  iqrCust, r = rpts) ) -> iqralclist
do.call(rbind,
        bootCustom(v = tbl_2_10$cotinine,
                   f =  iqrCust, r = rpts) ) -> iqrcotlist
```

The custom function *bootCustom* accepts the variable, the function to apply the variable and a vector of required replications. It uses `lapply` to call `boot` with the variable and the function as many times as there are elements in the replications argument, each time with one value from it. The result is a list of boot objects. Then, this is fed to another `lapply` which passes each of these boot object to *build_bci_str*, the custom function we made in the previous example which will provide a vector of confidence intervals for each boot object strung together as a list. We call *bootCustom* from inside `do.call`. The function `do.call` is some what similar to `lapply`. It accepts a function, which in our case is `rbind`, which it calls with a list, the components of which are fed to the function as its argument. In our case, the list for `do.call` is provided by our *bootCustom*. We call *bootCustom* through `do.call` six times, each time with a different combination of the data variable and the summarising function. The results are saved.

```
rbind.data.frame(cbind(mnchollist,
                       rep = rpts,
                       var = "meanchol"),
                 cbind(mnalclist,
                       rep = rpts,
                       var =  "meanalc"),
                 cbind(mncotlist,
                       rep = rpts,
                       var = "meancot"),
                 cbind(iqrchollist,
                       rep = rpts,
                       var = "iqrchol"),
                 cbind(iqralclist,
                       rep = rpts,
                       var = "iqralc"),
                 cbind(iqrcotlist,
                       rep = rpts,
                       var ="iqrcot")) |>
  pivot_longer(cols = c("Normal","Percentile","Basic","BCa"),
               names_to = "type", values_to = "ci") |>
  pivot_wider(names_from = var, values_from = ci) |>
  mutate(type = factor(type,
                       levels = c("Normal", "Percentile","Basic", "BCa")),
         rep = factor(rep, levels = as.character(rpts))) |>
  arrange(type, rep) |>
  filter(type != "Basic")
```

The saved results are combined column wise with a vector to indicate the replication number and the name of the data variable and then joined into one data frame using `rbind.data.frame`. We use `pivot_longer` to bring all values into one column, simultaneously adding another column to indicate the type of bootstrap interval. Then, we use `pivot_wider` to separate them out into different columns based on the data vector. Thus, the initial format where we had different columns for the different bootstrap intervals is transformed to one with different columns for different data vectors. We `mutate` the data to convert the *type* of bootstrap interval and that of *rep* to factors, so that they sort according to our needs. Finally, we use `filter` to remove the interval type *Basic*.

14.3 Further examples of bootstrapping

Example 14.5 (page 695)

We will now rework the example 14.5.

TABLE 14.2
Replication of table 14.2

		Means			Interquartile ranges		
Method	b	Cholesterol (mmol/l)	Alcohol (g/day)	Cotinine (ng/ml)	Cholesterol (mmol/l)	Alcohol (g/day)	Cotinine (ng/ml)
Normal	50	(6.13, 6.47)	(18.15, 34.71)	(91.38, 190.15)	(0.69, 1.32)	(11.54, 44.98)	(174.7, 386.37)
Normal	100	(6.1, 6.48)	(18.69, 35.62)	(88.37, 187.61)	(0.66, 1.35)	(13.08, 47.47)	(181.55, 375.21)
Normal	500	(6.07, 6.49)	(19.69, 34.38)	(89.58, 192.1)	(0.72, 1.3)	(12.58, 47.46)	(188.27, 382.29)
Normal	1000	(6.09, 6.49)	(18.86, 34.08)	(90.49, 186.69)	(0.7, 1.32)	(11.83, 47.88)	(188.44, 372.14)
Normal	5000	(6.08, 6.5)	(19.23, 34.38)	(90.23, 188.29)	(0.7, 1.32)	(12.15, 48.16)	(189.43, 375.11)
Normal	10000	(6.08, 6.5)	(19.21, 34.36)	(90.97, 187.46)	(0.71, 1.32)	(12.48, 47.98)	(186.57, 378.53)
Normal	50000	(6.08, 6.49)	(19.12, 34.43)	(90.79, 188)	(0.71, 1.32)	(12.33, 47.98)	(185.11, 381.41)
Normal	100000	(6.08, 6.49)	(19.09, 34.41)	(90.81, 187.85)	(0.71, 1.32)	(12.3, 47.91)	(186.16, 379.93)
Percentile	50	(6.04, 6.45)	(20.1, 37.28)	(85.09, 198.6)	(0.73, 1.36)	(14.11, 46.52)	(85.92, 383.62)
Percentile	100	(6.11, 6.52)	(18.11, 36.82)	(88.95, 193.16)	(0.67, 1.37)	(13.8, 45.45)	(194.85, 401.61)
Percentile	500	(6.08, 6.49)	(19.87, 34.4)	(87.6, 191.71)	(0.77, 1.37)	(17.2, 47.77)	(176.72, 387.35)
Percentile	1000	(6.09, 6.49)	(20.04, 34.84)	(93.34, 192.17)	(0.73, 1.37)	(17.08, 48.5)	(208.76, 388)
Percentile	5000	(6.08, 6.5)	(19.59, 34.81)	(91.94, 190.25)	(0.74, 1.36)	(17.1, 48.5)	(189, 393.5)
Percentile	10000	(6.08, 6.5)	(19.45, 34.81)	(92.36, 188.22)	(0.74, 1.36)	(17.15, 47.72)	(189, 391.73)
Percentile	50000	(6.08, 6.49)	(19.51, 34.78)	(92.2, 189.44)	(0.75, 1.36)	(17.03, 48.1)	(189, 388)
Percentile	100000	(6.08, 6.49)	(19.54, 34.8)	(92.66, 189.28)	(0.75, 1.36)	(17.02, 47.9)	(189, 391)
BCa	50	(6.19, 6.46)	(19.88, 34.02)	(90.85, 209.06)	(0.73, 1.36)	(13.58, 45.69)	(83.04, 383.49)
BCa	100	(6.12, 6.52)	(19.16, 39.31)	(90.98, 194.29)	(0.64, 1.35)	(14.46, 45.87)	(219.75, 402.41)
BCa	500	(6.07, 6.48)	(20.46, 35.89)	(93.63, 197.2)	(0.76, 1.36)	(17.51, 49.24)	(189, 393.62)
BCa	1000	(6.09, 6.48)	(20.36, 35.51)	(93.3, 192.03)	(0.71, 1.34)	(17.45, 50.44)	(195.25, 387.75)
BCa	5000	(6.07, 6.49)	(20.4, 36.36)	(93.76, 192.16)	(0.73, 1.35)	(17.54, 52)	(221.75, 400.75)
BCa	10000	(6.08, 6.5)	(20.2, 35.77)	(94.24, 190.57)	(0.73, 1.36)	(17.95, 51.34)	(219.75, 395)
BCa	50000	(6.07, 6.49)	(20.18, 35.83)	(94.62, 192.74)	(0.74, 1.36)	(17.55, 51)	(219.75, 395)
BCa	100000	(6.07, 6.49)	(20.1, 35.7)	(94.85, 191.94)	(0.74, 1.35)	(17.5, 50.65)	(219.75, 395)

```
read_table("K11828 supplements/Datasets/Table 9.8.DAT",
           col_names = c("country", "sugar", "dmft"),
           col_types = cols(country = col_factor(),
                            sugar = col_double(),
                            dmft = col_double())) -> data_14_8
filter(data_14_8, country == 2) -> data_14_8_2

lm(data = data_14_8_2, formula = log(dmft) ~ sugar) -> lm_14_8
lmCust <- function(data, index, formula) {
  coef(lm(formula, data = data[index,]))}

boot(data_14_8_2,
     formula = log(dmft) ~ sugar,
     lmCust,
     R = 2000) -> bs_14_8
boot.ci(bs_14_8, index = 2)
confint(lm_14_8)
```

We need a custom function that will accept data and its index along with other parameters needed for regression. Here, we intend to provide only the formula. Note that when we call `boot`, the formula is given as a named argument because `boot` sends `data` as the first argument and its index as the second. Instead of using `subset` argument of `lm`, we use `filter` to subset the dataframe and provide the filtered dataframe as `data`. We do this as our custom function doesn't accept a `subset` argument. After calling `boot` with the appropriate arguments, the result is stored. It is fed to `boot.ci`, which prints various bootstrap confidence intervals. Our custom function returns the result of `coef`. It will have the values for intercept and sugar. The `boot.ci` needs to be told to select one of those vectors to calculate the confidence interval. We use `index = 2` to say that we want the confidence interval for *sugar*. We may compare the bootstrap confidence intervals with the confidence interval produced by feeding the *lm_14_8* object to `confint`.

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 2000 bootstrap replicates

CALL :
boot.ci(boot.out = bs_14_8, index = 2)

Intervals :
Level      Normal              Basic
95%   ( 0.0123,  0.0302 )   ( 0.0120,  0.0303 )

Level     Percentile            BCa
95%   ( 0.0125,  0.0307 )   ( 0.0125,  0.0309 )
Calculations and Intervals on Original Scale
                2.5 %   97.5 %
(Intercept) -0.182456 0.373469
sugar        0.012005 0.030783
```

Example 14.6 (page 698)

We may follow the textbook method of extracting correlation coefficient from the model object or use `cor` to calculate it directly.

```
corCust <- function(data, index) {cor(data[index, ]$dmft,
                                      data[index,]$sugar,
                                      method = "pearson")}

boot(data_14_8_2, corCust, R = 2000) -> b_14_6
boot.ci(b_14_6) -> bci_14.6
c("Fisher", "Normal", "Percentile","Basic", "BCa") -> citypes

data.frame(
  rbind(cor.test(data_14_8_2$dmft, data_14_8_2$sugar)$conf.int,
        bci_14.6$normal[2:3],
        bci_14.6$perc[4:5],
        bci_14.6$basic[4:5],
        bci_14.6$bca[4:5])) |>
  mutate(pe = b_14_6$t0,
         labels = factor(citypes,
                         levels = rev(citypes))) -> data_14_8
```

The custom function that we pass to `boot` uses `cor` to calculate the Pearson correlation between the *dmft* and *sugar* components of the data argument after indexing it with the index variable. We pass the boot object to `boot.ci` and store it. We build a data frame from it by using `rbind` to bind together the appropriate element of the result and then passing the resultant matrix to `data.frame`. We add two new columns to indicate the point estimate and the labels for the confidence interval types. We specify the labels as a factor as we want order to be preserved when graphing. We reverse the order of levels as `ggplot` plots the first level near to zero and we want it away from zero like in our text. The Fisher confidence interval is calculated by `cor.test`.

Figure 14.8 (page 698)

```
ggplot(data_14_8, aes(x = pe, y =labels)) +
  geom_pointrange(aes(xmin = X1, xmax = X2)) +
  geom_text(aes(x = 0.75,
                label = paste0(round(pe,2),
                               " (",
                               round(X1,2),
                               ", ",
                               round(X2,2),
                               ")"))) +
  geom_text(aes(x = 0.1,
                label = labels), hjust = "left") +
  xlim(0.05,0.85) +
  xlab(NULL) +
  ylab(NULL) +
  theme_void()
```

We use `ggpplot` along with `geom_pointrange` to plot the confidence intervals. We add the numerical value of confidence interval to the plot using `geom_text`. We use `xlim` to expand the x-axis to accommodate the numerical values.

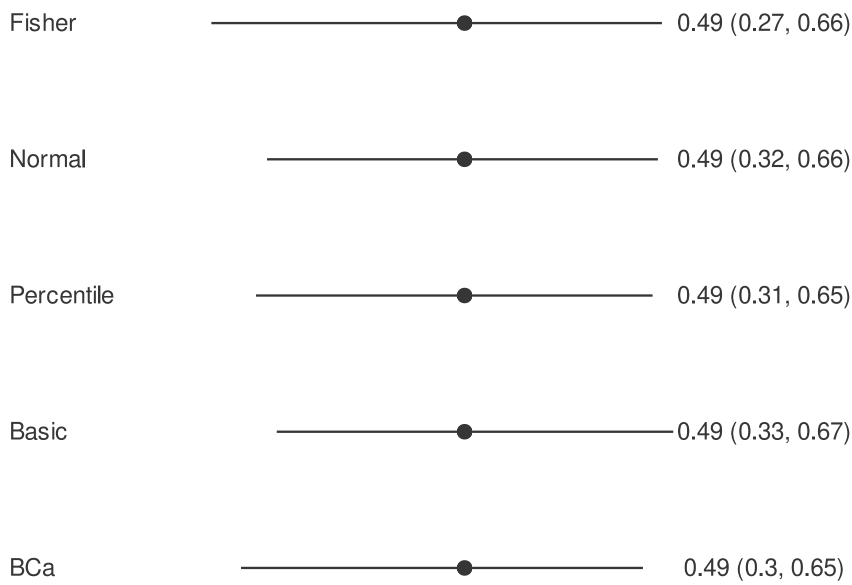


FIGURE 14.7
Replication of figure 14.8

Example 14.7 (page 698)

We now turn to example 14.7.

```
2 -tbl_2_10$chd -> tbl_2_10$chd2
build_bcistr <- function(bo, index = 1){
  b <- boot.ci(bo, index = index)
  c(Normal = confint.str(b$normal[2:3]),
    Percentile = confint.str(b$per[4:5]),
    Basic = confint.str(b$basic[4:5]),
    BCa = confint.str(b$bca[4:5]))}

buildlm <- function(var){
  f <- as.formula(paste(var, "~ chd2"))
  d <- tbl_2_10
  l <- lm(f, data = d)
  b <- boot(d, lmCust, formula = f, R = 2000)
  t <- b$t[,2]
  c(obsest = round(coef(l)["chd2"],3),
    bias = round(mean(t) - coef(l)["chd2"], 4),
    osd = round(summary(l)$coefficients["chd2",2],4),
    bsd = round(sd(t),4),
    obsci = confint.str(confint(l)["chd2",]),
    build_bcistr(b, index = 2))}
```

First, we create a new variable *chd2* from the exiting *chd*. We also modify *build_bcistr* to accept an *index* argument. We require *index* because *lmCust* returns more than one coefficient and so, we need to tell `boot.ci` which among the different vectors we are interested in. We make a new custom function *buildlm*, which accepts a string. This string is used to build the left-hand side of the `lm` formula using `as.formula`. Inside *buildlm* we call `boot` and provide it with this formula to pass on to *lmCust*. We also pass the formula to `lm`. We store the results of `boot` and `lm`. We extract the required information from these objects. We convert confidence intervals from `lm` to strings using *confint.str* and from `boot` using *build_bcistr*.

Table 14.3 (page 699)

```
bind_cols(`labels` = c("Observed estimate", "Bias",
                       "Observed SE","Bootstrap SE",
                       "Observed 95% CI","Normal 95% CI",
                       "Percentile 95% CI","Basic 95% CI",
                       "BCa 95% CI"),
          `Total Cholesterol` =  buildlm("chol"),
          `Systolic BP` = buildlm("sbp"),
          Alcohol = buildlm("alcohol"),
          Cigarettes = buildlm("cig"),
          `Carbon monoxide` = buildlm("co"),
          Cotinine = buildlm("cotinine"))
```

Next, we call *buildlm* multiple times, each time with the name of a different column in *tbl_2_10*. We bind together the result of these calls along with a vector of labels to recreate table 14.3.

Example 14.8 (page 700)

```
logit2r <- function(logit) { (1 + exp(- logit)) ^ -1}

confint.str <- function(x, digits = 2) {paste0("(",
                                               round(x[1],
                                                     digits),
                                               ", " ,
                                               round(x[2],
                                                     digits),
                                               ")")}

lgstCust <- function(data, index) {
  lgst <- glm(deaths ~ smoke,
              data = data[index,],
              family = binomial())
  logit2r(predict(lgst, newdata = data.frame(smoke = 1)))}

read_table("K11828 supplements/Datasets/Example 14.8.dat",
           col_names = c("smoke", "deaths")) -> data_14_8

nrow(filter(data_14_8, smoke == 1)) -> tot_smoke
```

```
nrow(filter(data_14_8,
            deaths == 1,
            smoke == 1)) -> death_smoke
death_smoke / tot_smoke  -> prop_death
sqrt(prop_death * (1-prop_death) / tot_smoke) -> sd_pdeath
glm(deaths ~ smoke,
    family = binomial(link ="log"),
    data = data_14_8) -> bi_14_8
predict(bi_14_8,
        newdata = data.frame(smoke =1),
        se.fit=TRUE) -> prdb_14_8
boot(data_14_8, lgstCust, R = 2000) -> b_14_8

data.frame(Data = c(rep("Observed",3), rep("Bootstrap",3)),
          `Analysis Method` =c("Binomial",
                               "Normal approximation",
                               "Binomial regression",
                               rep("Logistic regression",3)),
          `Bootstrap method` = c(rep("",3), "Normal",
                                  "Percentile", "Basic"),
          `Confidence interval 95%` = c(
             confint.str(binom.test(death_smoke,
                                     tot_smoke)$conf.int,
                         digits = 4),
             confint.str(prop_death +
                           c(-1.96, 1.96) * sd_pdeath,
                         digits = 4),
             confint.str(exp(prdb_14_8$fit[1] + c(-1,1) *
                               (prdb_14_8$se.fit[1] * 1.96)),
                         digits = 4),
             confint.str(boot.ci(b_14_8,
                                   type ="norm")$norm[2:3],
                         digits = 4),
             confint.str(boot.ci(b_14_8,
                                   type ="perc")$perc[4:5],
                         digits = 4),
             confint.str(boot.ci(b_14_8,
                                   type ="basic")$basic[4:5],
                         digits = 4)))
```

We need the *logit2r* function we built in the last chapter to convert logits to risks. We modify *confint.str* to accept the number of digits to round to. The third custom function we define is *lgstCust*, to which `boot` will pass our data frame. As the formula to be used is the same, the formula is defined inside the function and is not fed to it through `boot`.

We calculate the total smokers, the number of death among smoker, its proportion and normal approximation of its confidence interval for the proportion using the formula given in the textbook. The exact confidence interval is calculated using `binom.test`. Binomial regression model is fitted using `glm`. We build a vector of confidence intervals from these and

TABLE 14.3
Replication of table 14.3

	Total Cholesterol (mmol/l)	Systolic BP (mm Hg)	Alcohol (g/day)	Cigarettes (per day)	Carbon monoxide (ppm)	Cotinine (ng/ml)
Observed estimate	0.54	8.874	0.163	4.762	14.851	98.245
Bias	0.0085	−0.0356	−0.0762	−0.0287	−0.3785	0.464
Observed SE	0.2492	4.7566	9.573	4.2432	4.6888	59.4797
Bootstrap SE	0.264	6.0794	8.5253	4.3956	6.5575	60.1421
Observed 95% CI	(0.04, 1.04)	(−0.69, 18.44)	(−19.09, 19.41)	(−3.77, 13.29)	(5.42, 24.28)	(−21.35, 217.84)
Normal 95% CI	(0.01, 1.05)	(−3.01, 20.83)	(−16.47, 16.95)	(−3.82, 13.41)	(2.38, 28.08)	(−20.1, 215.66)
Percentile 95% CI	(−0.02, 1.06)	(−2.24, 21.64)	(−15.87, 17.68)	(−3.47, 13.7)	(2.08, 28.11)	(−24.34, 210.36)
Basic 95% CI	(0.02, 1.1)	(−3.89, 19.99)	(−17.36, 16.19)	(−4.17, 13)	(1.6, 27.62)	(−13.87, 220.83)
BCa 95% CI	(−0.05, 1.02)	(−1.08, 23.43)	(−14.36, 19.87)	(−2.71, 14.9)	(3.18, 30.08)	(−30.59, 207.16)

bind with the identifying text to create table 14.4. Note that, `boot.ci` couldn't calculate the confidence interval by BCa method and so we have omitted it here. That is the reason why we chose not to use *build_bcistr* to build the confidence intervals for the bootstrap values.

TABLE 14.4
Replication of table 14.4

Data	Analysis method	Bootstrap method	95% confidence interval
Observed	Binomial		(0.0149, 0.0309)
Observed	Normal approximation		(0.0143, 0.0295)
Observed	Binomial regression		(0.0154, 0.031)
Bootstrap	Logistic regression	Normal	(0.0141, 0.0297)
Bootstrap	Logistic regression	Percentile	(0.0141, 0.0299)
Bootstrap	Logistic regression	Basic	(0.0139, 0.0297)

Example 14.9 (page 701)

We will now rework the example 14.9.

```
library(survival)
read_table("K11828 supplements/Datasets/Example 10.12.DAT",
           col_names = c("age", "chol", "bmi", "sbp",
                         "smoke", "active", "chd", "survive"),
           col_types = cols( smoke = col_factor(levels = c("1","2","3")),
                             active =col_factor(levels =c("1","2","3")),
                             chd = col_integer())) -> data_14_9

confint.str <- function(x, digits = 2) {
  if (is.null(dim(x)))
    dim(x) <- c(1,2)
  paste0("(",
         round(x[,1],digits),
         ", " ,
         round(x[,2],digits),
         ")")}
```

After importing the relevant data, we modify our custom function `confint.str`. Earlier, we assumed that a vector of length two would be passed to it. Here, we will need to pass a matrix of confidence intervals (returned by `confint`) to it. That will throw errors due to inaccurate subscripting. So, we add a condition using `if` to check if the argument x has the attribute dimension. We do this using `dim` which will return `NULL` if there is no such attribute. Vectors don't have dimension attribute. We check if the return value of `dim()` is `NULL` using `is.null` in which case we assign the dimension `c(1,2)` to x. Thus, we convert any vector to a two dimensional matrix with one row. Rest of the code *confint.str* is not changed. The result will be a string for each row of the matrix passed made by joining together the values in its first and second column.

TABLE 14.5
Replication of table 14.5

	Observed results (sample data)						Bootstrap results	
Labels	Odds.ratio	Odds.ratio.CI	Hazard.ratio	Hazard.ratio.CI	Difference	Ratio	CI.for.difference	CI.for.ratio
Age	1.01725	(0.99, 1.045)	1.01819	(0.992, 1.045)	0.00093554	1.00092	(−0.0007, 0.0025)	(0.9991, 1.003)
Total Cholesterol	1.35944	(1.209, 1.528)	1.33118	(1.195, 1.483)	−0.02825781	0.97921	(−0.0525, −0.0105)	(0.9649, 0.9927)
BMI	1.04254	(0.999, 1.087)	1.03884	(0.998, 1.081)	−0.00369943	0.99645	(−0.0084, −0.0006)	(0.9935, 0.9997)
Systolic BP	1.02060	(1.013, 1.028)	1.02025	(1.013, 1.028)	−0.00034768	0.99966	(−0.0011, 0.0005)	(0.9987, 1.0004)
Smoker	1.38064	(0.851, 2.281)	1.36628	(0.847, 2.204)	−0.01435671	0.98960	(−0.0674, 0.0229)	(0.9556, 1.0176)
Smoker	2.07425	(1.372, 3.249)	2.01348	(1.325, 3.059)	−0.06076845	0.97070	(−0.2168, −0.0119)	(0.9394, 0.996)
Activity	0.82661	(0.584, 1.185)	0.82815	(0.59, 1.163)	0.00153747	1.00186	(−0.0248, 0.0237)	(0.9753, 1.03)
Activity	0.90388	(0.568, 1.424)	0.89605	(0.577, 1.39)	−0.00782695	0.99134	(−0.0545, 0.022)	(0.9542, 1.0283)

```
diffOddsHaz <- function(data,index) {
  lgmdl <- glm(chd ~ age + chol+bmi+sbp+smoke+active,
               data =  data[index,],
               family = binomial())
  cpmdl <- coxph(Surv(survive, chd) ~ age + chol+bmi+sbp+smoke+active,
                 data =  data[index,])
  exp(coef(cpmdl)) - exp(coef(lgmdl))[-1]}

ratioOddsHaz <- function(data,index) {
  lgmdl <- glm(chd ~ age + chol+bmi+sbp+smoke+active,
               data =  data[index,],
               family = binomial())
  cpmdl <- coxph(Surv(survive, chd) ~ age + chol+bmi+sbp+smoke+active,
                 data =  data[index,])
  exp(coef(cpmdl)) / exp(coef(lgmdl))[-1]}
```

We make two more custom functions *diffOddsHaz* and *ratioOddsHaz* which will calculate the difference between the exponentiated coefficients from logistic and Cox proportional hazards model & the ratio between them respectively. These functions need only accept a data and index argument as the formula used for model fitting doesn't change with each call of the function. The intercept term of the logistic model is removed from the coefficients by negative subscripting before the comparisons.

```
boot(data_14_9, diffOddsHaz, 200) -> bdiff_14_9
boot(data_14_9, ratioOddsHaz, 200) -> bratio_14_9
glm(chd ~ age + chol+bmi+sbp+smoke+active,
    data =  data_14_9,
    family = binomial()) -> lg_14_9
coxph(Surv(survive, chd) ~ age + chol+bmi+sbp+smoke+active,
      data =  data_14_9) -> cp_14_9

bciperc <- function(i, b) {
  confint.str(boot.ci(b, index = i, type = "perc")$perc[4:5],
              digits = 4)}

data.frame(Labels = c("Age", "Total Cholesterol", "BMI","Systolic BP",
                      "Smoker", "Smoker","Activity", "Activity"),
  `Odds ratio` = exp(coef(lg_14_9)[-1]),
  `Odds ratio CI` = confint.str(exp(confint(lg_14_9)),digits = 3)[-1],
  `Hazard ratio` = exp(coef(cp_14_9)),
  `Hazard ratio CI` = confint.str(exp(confint(cp_14_9)),digits = 3),
  `Difference`  = exp(coef(cp_14_9)) - exp(coef(lg_14_9)[-1]),
  `Ratio` = exp(coef(cp_14_9)) / exp(coef(lg_14_9)[-1]),
  `CI for difference` = do.call(rbind,
                                (lapply(1:8,
                                        FUN= bciperc,
                                        b = bdiff_14_9 ))),
  `CI for ratio` = do.call(rbind, lapply(1:8,
```

```
                                         FUN= bciperc,
                                         b= bratio_14_9)))
```

We pass these functions to `boot` and store the result. The models are also fitted without bootstrapping. The result of bootstrap will have values for each of the coefficients. We make a custom function *bciperc* which will accept a boot object and an index. We need this function as `boot.ci` will return confidence interval for only one index at a time and we will have eight indexes corresponding to the coefficients. We use `lapply` to pass the values 1 to 8 to these functions and collects the values in a list. We convert the list to a matrix by calling `rbind` on this list using `do.call`. All the relevant columns are bound together using `data.frame` to create the table we want.

Example 14.10 (page 703)

Complex boot straps are supported by `boot`. It accepts a `strata` variable.

```
read_delim("./K11828 supplements/Datasets/Table 7.1.DAT",
           "\t",
           col_names=c("treat", "nvinitial", "nvfinal","vinitial", "vfinal"),
           col_types = cols(
             treat = col_factor(),
             nvinitial = col_number(),
             nvfinal = col_number(),
             vinitial = col_number(),
             vfinal = col_number())) -> data_14_10
mutate(data_14_10,
       nvdiff = nvfinal - nvinitial,
       vdiff = vfinal - vinitial,
       treat = relevel(treat, ref = "2")) -> data_14_10

t.test(nvdiff ~ treat, data = data_14_10) -> ttnv_14_10
t.test(vdiff ~ treat, data = data_14_10) -> ttv_14_10

ttCust <- function(data,index) {
  nvstat <- t.test(nvdiff ~ treat, data = data[index,])
  vstat <- t.test(vdiff ~ treat, data = data[index,])
  return(c(nvstat$estimate[1] - nvstat$estimate[2],
           vstat$estimate[1] - vstat$estimate[2]))}

library(broom)
boot(data_14_10, ttCust, R = 2000,
     strata = data_14_10$treat) -> b_14_10
tidy(b_14_10) -> b_14_10.tidy

data.frame(labels = c("Observed estimate", "Bias",
                      "Observed SE", "Bootstrap SE",
                      "Observed 95% CI", "Normal 95% CI",
                      "Percentile 95% CI", "Basic 95% CI",
                      "BCa 95% CI"),
```

```
        `Nonverbal score` = c(round(b_14_10.tidy[[1,2]],3),
                              round(b_14_10.tidy[[1,3]],5),
                              round(ttnv_14_10$stderr,3),
                              round(b_14_10.tidy[[1,4]],3),
                              confint.str(ttnv_14_10$conf.int,digits = 3),
                              build_bcistr(b_14_10,index =1)),
        `Verbal score` = c(round(b_14_10.tidy[[2,2]],3),
                     round(b_14_10.tidy[[2,3]],5),
                     round( ttv_14_10$stderr,3),
                     round(b_14_10.tidy[[2,4]],3),
                     confint.str(ttv_14_10$conf.int,digits = 3),
                     build_bcistr(b_14_10, index =2)))
```

As with our previous examples, we import data, create a custom function to pass to `boot` and save the result of the test done without booting. We modify the imported data to store the difference in the two IQ scores and to change the reference level of *treat*. The custom function that we defined *ttCust* does two `t.tests` within its body and combines the corresponding statistics into a vector of length two. Thus, we will need to index the result of `boot`. The call to `boot` is different from previous examples in that we specify a `strata`. The `strata` argument should be a vector, which in our case is the *treat* column of the dataframe. The results are combined into a data frame using `data.frame`. We use the dataframe returned by `tidy` (when it is supplied the boot object) to select the observed estimates, bias and bootstrap SEs. We use *build_bcistr* and *confint.str* to convert confidence intervals into strings before joining them to form the table.

TABLE 14.6
Replication of table 14.6

labels	Nonverbal.score	Verbal.score
Observed estimate	2.405	0.506
Bias	−0.01841	−0.01465
Observed SE	1.935	1.392
Bootstrap SE	1.924	1.38
Observed 95% CI	(−1.445, 6.254)	(−2.262, 3.275)
Normal 95% CI	(−1.35, 6.19)	(−2.18, 3.23)
Percentile 95% CI	(−1.33, 6.21)	(−2.26, 3.2)
Basic 95% CI	(−1.4, 6.14)	(−2.19, 3.27)
BCa 95% CI	(−1.35, 6.2)	(−2.11, 3.3)

14.4 Bootstrap hypothesis testing

Example 14.11 (page 704)

We will now rework example 14.11.

```
t.test(chol ~ chd,
       data = tbl_2_10,
       var.equal = FALSE) -> t_14_11
t_14_11
```

As we mentioned in an early chapter, by default R assumes unequal variance between the two samples of a t test. Here, we choose that option explicitly by specifying `var.equal = FALSE`. We store the result. We may print the `t.test` result to confirm that the results agree with the textbook.

```
    Welch Two Sample t-test

data:  chol by chd
t = 2.02, df = 14.7, p-value = 0.062
alternative hypothesis: true difference in means between
      group 1 and group 2 is not equal to 0
95 percent confidence interval:
 -0.030675  1.111654
sample estimates:
mean in group 1 mean in group 2
         6.7082          6.1677
```

```
mean(tbl_2_10$chol[tbl_2_10$chd == "1"]) -> mnchol_1
mean(tbl_2_10$chol[tbl_2_10$chd == "2"]) -> mnchol_2
mutate(tbl_2_10,
       cholt = case_when(chd == "1" ~ chol - mnchol_1,
                         chd == "2" ~ chol - mnchol_2)) -> tbl_2_10

ttCust <- function(data, index) {
  (t.test(cholt ~ chd, data = data[index,]))$statistic}
boot(tbl_2_10, ttCust, R = 2000,
     strata = tbl_2_10$chd) -> b_14_11

prop.table(table(abs(b_14_11$t >= t_14_11$statistic )))
```

We calculate the mean cholesterol levels in the two groups using `mean`. The argument given to `mean` is the *chol* vector values selected by logical subsetting. We `mutate` the dataframe to form a new column by subtracting the appropriate mean from *chol*. Note that calculating mean on this modified values by *chd* group will not show zero, but a very small number. This is due to the approximation used by computers to represent floating point numbers.

We define our custom function to accept data, its index and perform the t test on it and return the t statistic. We call this custom function from `boot`, specifying `strata` as the *chd* variable. From the returned values, we calculate the proportion of values above or below the cutoff set by the statistic returned by the `t.test` on the original data.

```
    0     1
0.971 0.029
```

To calculate the robust confidence intervals, we need two more libraries – `lmtest` and `sandwich`.

```
library(lmtest)
library(sandwich)
lm(chol ~ chd, data = tbl_2_10) -> l_14_11
coefci(l_14_11, vcov. = vcovHC(l_14_11, type = "HC3"))
```

First, we fit a linear regression using `lm`. As a second step, we call `coefci` to which we supply the lm object as the first argument. The `vcov.` argument is given the result of `vcovHC` which is fed the lm object. It also accepts various `types`. Here, we explicitly use `HC3`, though it is the default value. We may use `coeftest` in place of `coefci` with the same arguments to perform a t test of the coefficients using the robust standard errors.

```
               2.5 %   97.5 %
(Intercept)  6.2014 8.295930
chd         -1.1011 0.020163
```

Before we leave the topic of bootstrap, it may be noted that `boot` supports parametric bootstrapping as well.

14.5 Permutation tests

Table 14.7 (page 707)

Permutation tests are done using the package `coin`. Remember to install it using `install.packages` as described in chapter 1. First, we will try to reproduce table 14.7.

```
c(0.65,0.85,0.8,0.95,1) -> creat_14_7
factor(c("Women","Women", "Men", "Men", "Men")) -> sex_14_7
library(DescTools)
Permn(sex_14_7) -> permsex
cbind(permsex,
      round(
        apply(permsex, MARGIN = 1,
              function(x) mean(creat_14_7[x == "Women"])),2),
      round(
        apply(permsex, MARGIN = 1,
              function(x) mean(creat_14_7[x == "Men"])),2),
      round(
        abs(
```

```
          apply(permsex, MARGIN = 1,
                function(x) mean(creat_14_7[x == "Women"])) -
            apply(permsex, MARGIN = 1,
                  function(x) mean(creat_14_7[x == "Men"]))),2)) -> tbl_14_7
data.frame(tbl_14_7)
```

We use `Permn` from `DescTools` to create an array of all permutations of the *sex_14.6* vector. We then use `apply` four times, each time by specifying `MARGIN = 1`, to mean row wise, to apply an anonymous function across the values in that row. The anonymous function returns the mean of those values of *creat_14_7* for which the corresponding *sex_14.6* is as specified – either women or men. We bind the values returned by the first two calls to `apply`, a vector of length 10 (one value for each row in *permsex*) column-wise to *permsex*. The third and fourth calls to `apply` are repetitions of the earlier calls, now used to calculate the absolute difference between them, the result of which is also column bound with the others.

TABLE 14.7
Replication of table 14.7

1st value	2nd value	3rd value	4th value	5th value	Female mean	Male mean	Absolute difference
Women	Women	Men	Men	Men	0.75	0.92	0.17
Women	Men	Women	Men	Men	0.73	0.93	0.21
Men	Women	Women	Men	Men	0.82	0.87	0.04
Women	Men	Men	Women	Men	0.8	0.88	0.08
Men	Women	Men	Women	Men	0.9	0.82	0.08
Men	Men	Women	Women	Men	0.88	0.83	0.04
Women	Men	Men	Men	Women	0.82	0.87	0.04
Men	Women	Men	Men	Women	0.92	0.8	0.12
Men	Men	Women	Men	Women	0.9	0.82	0.08
Men	Men	Men	Women	Women	0.98	0.77	0.21

```
as.numeric(tbl_14_7[,8]) -> abdif
abs(mean(creat_14_7[sex_14_7 == "Women"]) -
      mean(creat_14_7[sex_14_7 == "Men"])) -> obsdif
length(abdif[abdif > obsdif]) / length(abdif)
```

We select the last column of the matrix, convert it back to numbers. We calculate the observed difference in mean by using `mean` which is provided with values from *creat_14_7* after subsetting for one of the sexes. Finally, we calculate the number of values in the calculated difference column greater than the observed and divide by the number of permutations.

```
[1] 0.3
```

Rather than calculating by hand, we may use `indepenence_test` from `coin`.

```
library(coin)
independence_test(creat_14_7 ~ sex_14_7,
                  distribution = "exact")
```

The first argument to `independence_test` is a formula, the left-hand side of which is the variable with the value to be compared and the right-hand side is the variable which determines the groups. We use `distribution = "exact"` to instruct the function to not use approximations or resampling.

```
    Exact General Independence Test

data:  creat_14_7 by sex_14_7 (Men, Women)
Z = 1.33, p-value = 0.3
alternative hypothesis: two.sided
```

14.5.1 Montecarlo permutation tests

Section 14.8.1 (page 707)

To use Monte Carlo simulation, we need to change the value of `distribution`.

```
independence_test(creat_14_7 ~ sex_14_7,
                  distribution = approximate(nresample = 100000))
```

We may specify `distribution` as a function as above. We need to do it only if we want to change the default value of its arguments. Otherwise specifying the string "approximate" is sufficient. Here, though we are not changing the default value, we specify `distribution = approximate(nresample = 100000)` to demonstrate how to change the number of MC simulations if needed.

```
    Approximative General Independence Test

data:  creat_14_7 by sex_14_7 (Men, Women)
Z = 1.33, p-value = 0.3
alternative hypothesis: two.sided
```

The `coin` package provides many tests. However, I am not sure of the result they return. We, will try another path.

```
replicate(10000,
          expr = t.test(tbl_2_10$chol ~ sample(tbl_2_10$chd),
```

```
                            var.equal =FALSE)$statistic) -> reptt
prop.table(table(abs(reptt) >= t_14_11$statistic))
```

We use `replicate` to repeat an expression a specified number of times. The first argument to `replicate` is the number of repetitions. Next, the expression that needs to be replicated is provided. In our example, we ask for a t test to be done with the variance assumed to be unequal. The means of cholesterol levels are compared between the groups determined by *chd*. This is similar to the usual `t.test`. The difference lies in the fact that *chd* is sampled without replacement, in effect shuffled, each time. The function `sample` returns a sample of specified size from its first argument. If no numbers are specified, the number of samples is equal to the number in the original argument. By default, `replace = FALSE`. This means that under the default options, we get a random permutation of the original vector. Thus, at each repetition of the test, the cholesterol values are categorised against a random permutation of the observed *chd* outcome variable. Next, we check what proportion of the values produced by repetitions are larger than the statistic calculated in the observed sample.

```
FALSE  TRUE
0.945 0.055
```

A permutation test for the Wilcoxon test is done similarly.

```
wilcox.test(tbl_2_10$chol ~ tbl_2_10$chd) -> w_14_11
replicate (10000,
            expr = wilcox.test(tbl_2_10$chol ~
                                   sample(tbl_2_10$chd))$statistic
            ) -> repwt
prop.table(table(repwt >= w_14_11$statistic))["TRUE"] * 2
```

The proportion of the replicated values greater than the observed test statistic is one sided. So, we need to multiply it with two.

```
  TRUE
0.0488
```

The `wilcox_test` of `coin` may also be used.

```
wilcox_test(chol ~ factor(chd),
             data = tbl_2_10,
             distribution = "approximate")
```

```
    Approximative Wilcoxon-Mann-Whitney Test

data:  chol by factor(chd) (1, 2)
Z = 1.99, p-value = 0.044
```

```
alternative hypothesis: true mu is not equal to 0
```

We proceed to calculate the difference in medians.

```
tapply(tbl_2_10$chol ,  tbl_2_10$chd, FUN = median) |>
  diff() -> diffmed_14_11
replicate (10000,
           expr = diff(tapply(tbl_2_10$chol,
                              sample(tbl_2_10$chd),
                              FUN = median))) -> repmd
prop.table(table(abs(repmd) >= abs(diffmed_14_11)))
```

The function `tapply` applies the function supplied as its `FUN` argument to the categories of values of the first argument as determined by the combination of the other factor argument(s) supplied. Here, we ask it to calculate the median of the cholesterol vector as grouped by the values in the *chd* vector. Thus, we will get two values. We pass the result to `diff` which calculates the difference between the consecutive members of its argument. Thus, we get the difference between the two medians of cholesterol in the two CHD groups.

We use `replicate` to repeat this exercise 10,000 times, with the change that each time a permuted sample of *chd* is provided. Finally, we calculate the proportion of the repetitions in which the absolute value of calculated difference in medians exceed absolute value of the observed difference.

```
 FALSE   TRUE
0.9579 0.0421
```

14.6 Missing values

Example 14.12 (page 711)

We now turn to the topic of missing data.

```
read_table("K11828 supplements/Datasets/Table 14.8.dat",
           col_names = c("sex", "age", "sbp"),
           na = ".") -> tbl_14_8
```

Note the option `na`. It is used to specify which character(s) should be considered as representing missing data. In R, the missing data is marked as `NA`, a logical value.

Let us calculate the mean of *sbp*.

```
mean(tbl_14_8$sbp)
```

```
[1] NA
```

The result may surprise the newcomer. When there are NAs in a data, the result of most functions is NA. We have to specify explicitly that the missing data has to be omitted for calculations.

```
mean(tbl_14_8$sbp, na.rm = TRUE)
```

```
[1] 146.53
```

R also provides many functions like `is.na`, `anyNA`, `na.action` etc. to check if any of the variables are NAs. Equally useful are functions like `Desc` from `DescTools`, `describe` from `Hmisc` and `skim` from `skimr` which will summarise data in more useful ways including distinct values, extreme values… in addition to providing info on NAs.

Here, we see the result of `marginplot` of `VIM`. Remember to install `VIM` using `install.packages` as described in chapter 1. It accepts a data frame or matrix with two columns.

```
library(VIM)
marginplot(tbl_14_8[,c( "age","sbp")],
           col =  c("#111111","#004B73","#713430","#7FBFF5", "#F8A29E"))
```

The `marginplot` produces a visual summary of missing values in the two columns of a data frame. We subset the data frame to select the two columns we want to compare. In addition to a scatter plot, it shows two boxplots each on the margins of both axes. One for the NAs in the other variable and one for which values are available for the other variable. The boxplots are colour coded and the number of NAs are also shown. The `col` argument decides the color palette used. The NAs are also shown in univariate scatter plots. The package also has an `aggr` function which compares all combinations of columns in a data frame for missing values. It has `summary` and `plot` methods too.

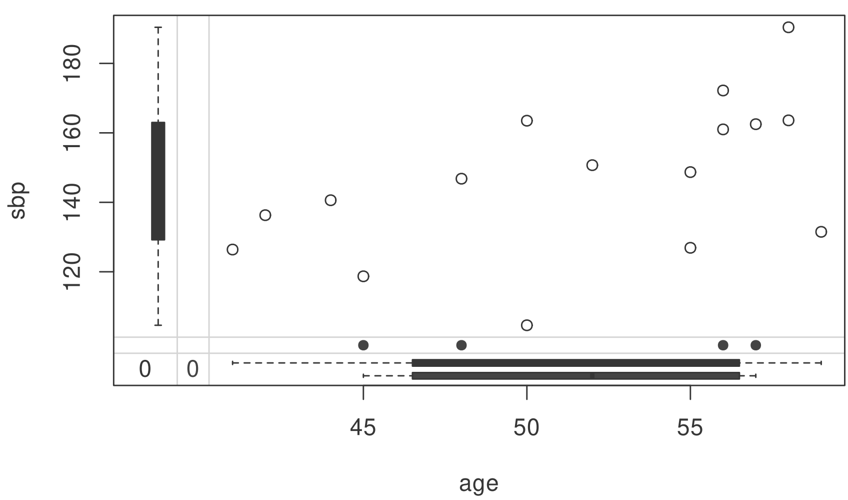


FIGURE 14.8
Marginplot showing the missing values in example 14.12

Section 14.9.3 (page 714)

The function `na.omit` will return complete cases for analysis.

```
anyNA(tbl_14_8)
anyNA(na.omit(tbl_14_8))
```

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

14.7 Naive imputation methods

Section 14.10.1 (page 716)

We may apply mean imputation by subsetting the `NA` values and assigning them new values.

```
tbl_14_8$sbp -> sbp
tbl_14_8$sbp
```

First, we save the systolic blood pressure variable as a new vector to avoid overwriting the original values.

```
sbp[is.na(sbp)] <- mean(sbp, na.rm = TRUE)
sbp
```

```
 [1] 163.5 126.4 150.7 190.4 172.2    NA 136.3 146.8 162.5 161.0 148.7 163.6
[13]    NA 140.6    NA 118.7    NA 104.6 131.5 126.9
 [1] 163.50 126.40 150.70 190.40 172.20 146.53 136.30 146.80 162.50 161.00
[11] 148.70 163.60 146.53 140.60 146.53 118.70 146.53 104.60 131.50 126.90
```

Inspecting the vector after assignment confirms that all the `NA`s have been changed to the mean values. The function `impute` from `Hmisc` can also perform simple imputation allowing the choice of custom functions or random values.

Example 14.14 (page 717)

We will use `regressionImp` to perform conditional mean imputation and regression imputation.

```
regressionImp(sbp ~ age,
              data = subset(tbl_14_8, sex == 1)) -> mimp_14_8
```

The function accepts a regression formula, decides the regression method automatically and imputes missing values for the column on the left-hand side of the formula. A new column is also created to carry a logical indicator to show if the value is imputed or original. Its name is created by joining the name of the original column, "_" and the argument `imp_suffix`, the default for which is "imp". Thus the new column is named `sbp_imp`. If we don't want the indicator column, we may set `imp_var` to `FALSE`.

We pass the data frame with imputed values to `marginplot`.

```
marginplot(mimp_14_8[, c( "age", "sbp","sbp_imp")],
           delimiter = "_imp",
           col =  c("#111111","#004B73","#713430","#7FBFF5", "#F8A29E"))
clip(min(mimp_14_8$age),max(mimp_14_8$age),
     min(mimp_14_8$sbp), max(mimp_14_8$sbp))
abline(coef = coef(lm(sbp ~ age,data = subset(tbl_14_8, sex == 1))),
       lty = 2,
       col = "black")
```

Now, the dataframe supplied to `marginplot` should have the indicator column showing the nature of the values. The `delimiter` option is used to set the string added to the name of the original variable to make the name of the indicator variable. Note that though the value of `imp_suffix` in `regressionImp` is "imp", the value of `delimiter` for `marginplot` is "_imp", with an underscore prefixed to `imp_suffix`. The plot highlights the imputed values as it does for missing values. The `clip` is used to specify the clipping co-ordinates for the plots that follow the command. We intend to draw the regression line over the margin plot. But, we want it to be within the limits of the data. So, we specify x and y coordinates for the clipping rectangle as the maximum and minimum of the two variables. After setting the clipping area, we draw the regression line using `abline`. We provide it with the coefficients returned by the regression as its `coef` argument. Note that `VIM` has `scattMiss` which will plot just the scatterplot if you really don't want the `marginplot`.

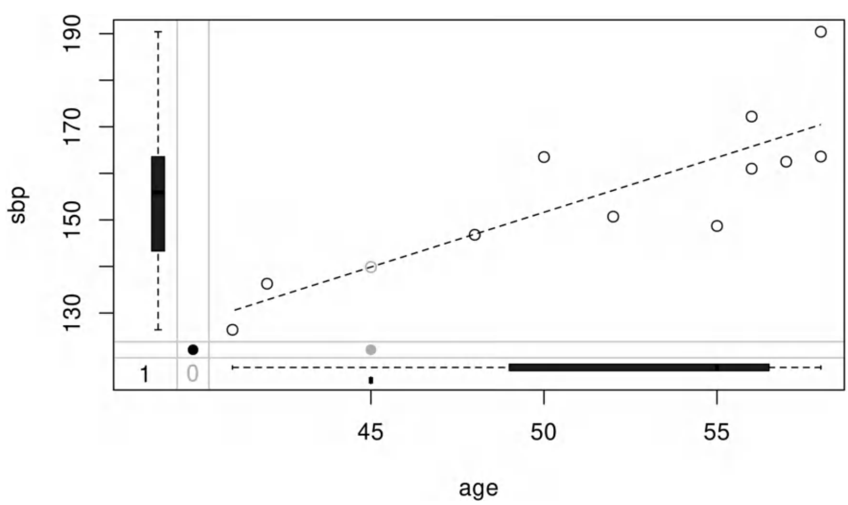


FIGURE 14.9
Replication of figure 14.9

For conditional mean imputation, all we have to do is change the right-hand side of the formula supplied to `regressionImp` to the categorical variable.

```
regressionImp(sbp ~ sex, data = tbl_14_8) -> seximp_14_8
```

For regression imputing based on several conditional variables, all we need do is include all the variable on the right-hand side of the formula supplied to `regressionImp`.

```
regressionImp(sbp ~ age + sex, data = tbl_14_8) -> imp_14_8
```

Section 14.10.3 (page 719)

The hot deck imputation method is achieved by `hotdeck`.

```
hotdeck(tbl_14_8,
        variable = "sbp",
        ord_var = "age",
        domain_var =  "sex")
```

The argument `variable` is used to specify the variables with missing value. The `domain_var` determines the variables used for cross classifications. The variables specified as `ord_var` is used for sorting the data set before imputation.

```
   sex age   sbp sbp_imp
1    1  50 163.5   FALSE
2    1  41 126.4   FALSE
3    1  52 150.7   FALSE
4    1  58 190.4   FALSE
5    1  56 172.2   FALSE
6    1  45 136.3    TRUE
7    1  42 136.3   FALSE
8    1  48 146.8   FALSE
9    1  57 162.5   FALSE
10   1  56 161.0   FALSE
11   1  55 148.7   FALSE
12   1  58 163.6   FALSE
13   2  57 126.9    TRUE
14   2  44 140.6   FALSE
15   2  56 126.9    TRUE
16   2  45 118.7   FALSE
17   2  48 118.7    TRUE
18   2  50 104.6   FALSE
19   2  59 131.5   FALSE
20   2  55 126.9   FALSE
```

The imputed data shows the imputed values.

14.8 Univariate multiple imputation

Section 14.11.2 (721)

We need the package `mice` for multiple imputations. Remember to install it using `install.packages` as described in chapter 1. For the three steps of multiple imputation, it provides three functions.

```
library(mice)
filter(tbl_14_8, sex == "1") -> tblm_14_8
mice(tblm_14_8,
     m = 5,
     method = "norm",
     seed = 123,
     printFlag = FALSE) -> mitblm_14_8
complete(mitblm_14_8, action = "repeated") |>
  select(starts_with("sbp")) |>
  slice(6)
```

The function `mice` does the first step of producing the imputed data sets. It requires the original dataframe with missing values. The number of imputations required is specified through `m`, the default value of which is 5. `mice` also gives many options for the actual imputation method, which we may specify through `method` argument. Here, we selected `"norm"` as we want it to assume a normal distribution with random errors. The function `mice` accepts a random seed which we specify using the argument `seed`. We use `printFlag = FALSE` to suppress the messages that flash on the screen during computation.

We may use `complete` to access the dataframes completed by mice through imputation. The `action` argument determines the structure of the dataframe. Here we choose `"repeated"`. We `select` only those columns, the names of which start with “sbp” using the helper function `starts_with` and then `slice` the row number 6 to show the imputed value for the sixth male.

```
   sbp.1  sbp.2  sbp.3  sbp.4  sbp.5
1 136.82 155.07 146.51 130.31 155.08
```

The function that the package provides for the next step of estimating the quantity required is `with`.

```
with(mitblm_14_8, mean(sbp))
```

We supply the imputed data object returned by `mice` (called a mids object by mice) to `with` along with the expression that needs to be evaluated on each of the data sets. It returns a list with as many components as the number of repetitions used to build the mids object. Each element of the list will be the result returned by the expression supplied to `with`. Thus, we will have a list of five means.

For the final step of combining the results from each of the different data sets into one, we have `pool`. However, it can handle only model objects. So, we cheat it.

```
summary(
  pool(
    with(mitblm_14_8, lm(sbp ~ 1))))
```

With `with`, we use `lm` regressing *sbp* against a constant to get the mean of *sbp*, which becomes the argument for `pool`. `pool` returns many statistics including the averaged estimate from all the different models. Here, we use `summary` to show a restricted set.

```
         term estimate std.error statistic     df     p.value
1 (Intercept)   155.57    5.0789    30.631 9.0256 1.9658e-10
```

Table 14.9 (page 726)

We will now try to recreate table 14.9.

```
mice(tbl_14_8,
     method =  "norm",
     seed = 234,
     printFlag  = FALSE) -> mitbl_14_8
complete(mitbl_14_8,
         action = "repeated",
         include = TRUE) |>
  select(sex.0,age.0,num_range("sbp.", 0:5)) -> tbl_14_9

summarise(tbl_14_9,
          across(num_range("sbp.", 0:5),mean, na.rm = TRUE)) -> mns_14_9
summarise(tbl_14_9,
          across(num_range("sbp.", 0:5),sd, na.rm = TRUE)) -> sd_14_9
c(16,rep(20,5)) -> nr_14_9
sd_14_9 / sqrt(nr_14_9) -> se_14_9
bind_rows(tbl_14_9, mns_14_9, se_14_9)
```

First, we use `mice` to build the imputed dataframes and then bind them together using `complete`. By default, `mice` uses all the columns in the imputer's model. Here, we rely on the default. Now, we select the required columns to get the basic table. The function `num_range` is a helper function to select columns. It selects those columns named by adding the numbers given as its second argument one by one to its first argument. Here, we will get the columns `sbp.1`, `sbp.2` ... `sbp.5`. The columns `sex.0` and `age.0` are the columns of the original data frame that was supplied to `mice`, which we asked `complete` to include in its result by specifying `include = TRUE`. In other words, `complete` returns as many columns as there are in the original dataframe, multiplied by the number of imputations (plus one if you specify `include = TRUE`). These are named by appending to the original column names a period followed by a number, starting with zero for the original columns. We use `across` to apply `summarise` the imputed columns selected using `num_range` to calculate various statistics like mean and sd.

TABLE 14.8
Replication of table 14.9

		Systolic blood pressure (mmHg)					
Sex	Age (yrs)	Observed	Imp1	Imp2	Imp3	Imp4	Imp5
1	50	163.5000	163.5000	163.5000	163.500	163.5000	163.5000
1	41	126.4000	126.4000	126.4000	126.400	126.4000	126.4000
1	52	150.7000	150.7000	150.7000	150.700	150.7000	150.7000
1	58	190.4000	190.4000	190.4000	190.400	190.4000	190.4000
1	56	172.2000	172.2000	172.2000	172.200	172.2000	172.2000
1	45		132.0063	171.4862	130.140	116.9105	132.9174
1	42	136.3000	136.3000	136.3000	136.300	136.3000	136.3000
1	48	146.8000	146.8000	146.8000	146.800	146.8000	146.8000
1	57	162.5000	162.5000	162.5000	162.500	162.5000	162.5000
1	56	161.0000	161.0000	161.0000	161.000	161.0000	161.0000
1	55	148.7000	148.7000	148.7000	148.700	148.7000	148.7000
1	58	163.6000	163.6000	163.6000	163.600	163.6000	163.6000
2	57		134.2187	110.2050	140.296	156.3410	145.8913
2	44	140.6000	140.6000	140.6000	140.600	140.6000	140.6000
2	56		125.6618	118.1960	146.536	139.5365	136.4099
2	45	118.7000	118.7000	118.7000	118.700	118.7000	118.7000
2	48		125.3708	123.8276	124.431	124.5686	117.8033
2	50	104.6000	104.6000	104.6000	104.600	104.6000	104.6000
2	59	131.5000	131.5000	131.5000	131.500	131.5000	131.5000
2	55	126.9000	126.9000	126.9000	126.900	126.9000	126.9000
		146.5250	143.0829	143.4057	144.290	144.0878	143.8711
		5.5296	4.6866	5.2324	4.598	4.7914	4.6765

Example 14.17 (page 725)

We use `pool.scalar` for estimating the imputation summaries.

```
pool.scalar(Q = as.numeric(mns_14_9)[2:6],
            U = as.numeric(se_14_9)[2:6] ^ 2,
            n = nrow(tbl_14_9)) -> mipo_14_9
mipo_14_9
```

The function `pool.scalar` requires the estimate and square of the standard error, calculated for each group, as two vectors, along with the number of observations.

```
$m
[1] 5

$qhat
[1] 143.08 143.41 144.29 144.09 143.87

$u
[1] 21.964 27.378 21.142 22.958 21.869
```

```
$qbar
[1] 143.75

$ubar
[1] 23.062

$b
[1] 0.24603

$t
[1] 23.357

$df
[1] 17.043

$r
[1] 0.012802

$fmi
[1] 0.11116
```

It returns a list with the mean of the estimates named `qbar`, the within variance named as `ubar`, between variance named as `b` and the total variance named as `t`. The relative increase in variance is named `r`, the fraction of information lost due to missingness is named `fmi`. The standard error of the pooled estimate needs to be calculated by hand.

```
sqrt(mipo_14_9$ubar + (mipo_14_9$m+1 *mipo_14_9$b/mipo_14_9$m))
```

```
[1] 5.302
```

The package provides many graphical methods for imputation diagnostics. The functions `stripplot`, `xyplot`, `bwplot` all of which colour codes the imputed data and show their distribution in relation to the original data.

Example 14.18 (page 729)

The default method for imputation in `coin` is `"pmm"`, for predictive mean matching. It however supports many more methods if we want to use them. First, we will see the problem with the assumption of a normal distribution.

```
read_table("K11828 supplements/Datasets/Example 14.18.dat",
           col_names = c("age", "sex", "co", "smoke",
                         "cotinine", "survive", "death"),
           na = ".") -> data_14_18
stripplot(mice(data_14_18,
               method = "norm",
               printFlag = FALSE,
               seed = 345))
```

Here, we import the data and pass it to `mice`. We make sure that we specify `method = "norm"` as an additional argument. The result is supplied to `stripplot`.

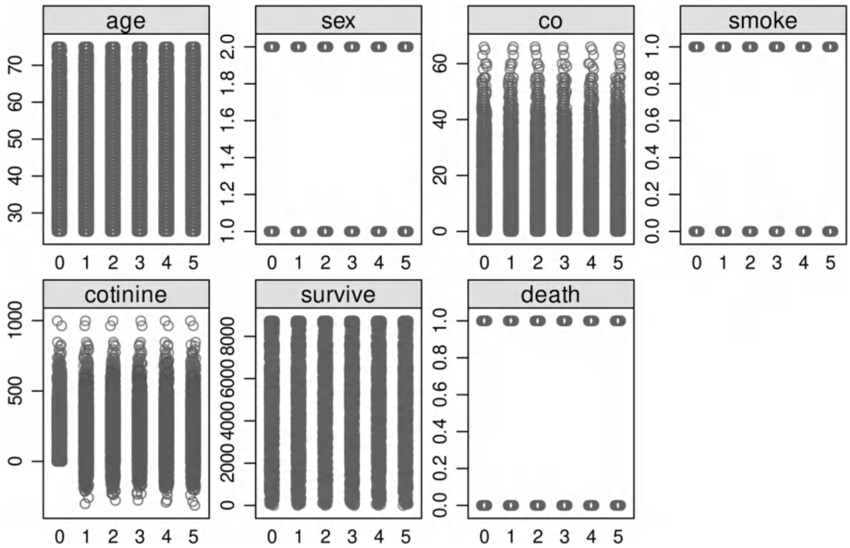


FIGURE 14.10
Imputed values assuming normal distribution visualised using strip plot

We can see that red coloured dots representing *cotinine* value in each of the five imputed samples dips down below zero.

We now pass the data to “mice” without specifying the method.

```
mice(data_14_18, seed = 456, printFlag = FALSE) -> mi_14.18
stripplot(mi_14.18)
```

The strip plot made with this imputed data doesn’t have any value lesser than zero.

```
mi_14.18
```

If we print the mice object, we get to inspect the `method` and `PredictorMatrix` components of the mice object. The `methods` component gives the method used for imputing the missing values of different variables. Here, we see that the method `"pmm"` was used for *cotinine* and no method was used for other variables (because there were no missing values). The `PredictorMatrix` tells us which of the variables were considered to impute the missing values for a given variable. For *cotinine*, we can see that all other variables were considered. `mice` allows us to fine control the imputation process by changing the method for each of the variable via its `method` argument and the variables that will be considered by the methods for each variable via `predictorMatrix` argument. Various methods are supported by `mice` including logistic regression, polytomous regression, bootstrap methods etc.

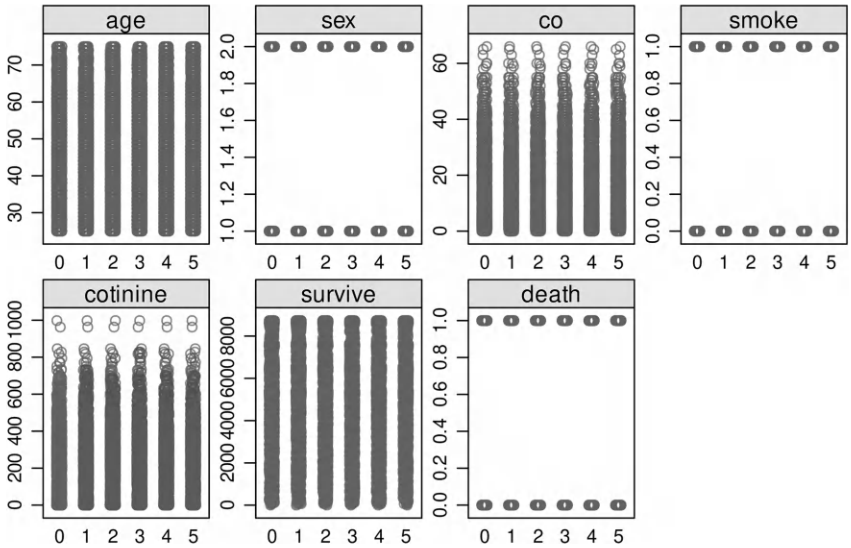


FIGURE 14.11
Imputed values without assumption of normal distribution visualised using strip plot

```
Class: mids
Number of multiple imputations:  5
Imputation methods:
     age      sex       co    smoke cotinine  survive    death
      ""       ""       ""       ""    "pmm"       ""       ""
PredictorMatrix:
         age sex co smoke cotinine survive death
age        0   1  1     1        1       1     1
sex        1   0  1     1        1       1     1
co         1   1  0     1        1       1     1
smoke      1   1  1     0        1       1     1
cotinine   1   1  1     1        0       1     1
survive    1   1  1     1        1       0     1
```

Performing a complete case analysis and analysis with the imputed data follows.

```
library(survival)
coxph(Surv(survive, death) ~ age + sex + cotinine + I(cotinine ^ 2),
      data = data_14_18)
```

```
pool(
  with(mi_14.18,
       coxph(Surv(survive, death) ~age + sex + cotinine + I(cotinine ^ 2))))
```

Note that when we pass the result of a regression analysis with the imputed data using `with`, the result can be passed directly to `pool` rather than calculating mean and variance by hand.

```
Call:
coxph(formula = Surv(survive, death) ~ age + sex + cotinine +
    I(cotinine^2), data = data_14_18)

                      coef   exp(coef)    se(coef)    z      p
age             0.10006397  1.10524162  0.00328031 30.5 <2e-16
sex            -0.48705026  0.61443615  0.06453271 -7.5  4e-14
cotinine        0.00393861  1.00394637  0.00047587  8.3 <2e-16
I(cotinine^2) -0.00000308  0.99999692  0.00000096 -3.2  0.001

Likelihood ratio test=1272  on 4 df, p=<2e-16
n= 4530, number of events= 983
   (858 observations deleted due to missingness)


Class: mipo    m = 5
           term m      estimate       ubar           b           t dfcom
1           age 5  0.0990461196 8.9913e-06 4.3046e-08 9.0429e-06  1198
2           sex 5 -0.4160455262 3.3759e-03 1.9436e-05 3.3992e-03  1198
3      cotinine 5  0.0044824527 1.9169e-07 2.2885e-08 2.1915e-07  1198
4 I(cotinine^2) 5 -0.0000039715 7.7580e-13 1.3415e-13 9.3678e-13  1198
       df       riv    lambda       fmi
1 1177.75 0.0057451 0.0057123 0.0073964
2 1171.42 0.0069087 0.0068613 0.0085525
3  204.85 0.1432647 0.1253119 0.1337286
4  119.15 0.2075067 0.1718472 0.1854064
```

Table 14.11 (page 732)

We will now try to build table 14.11.

```
cotcof <- function(seed, m=5) {
  mice(data_14_18, m= m, seed = seed, printFlag = FALSE) -> midf
  pool(with(midf,
            coxph(Surv(survive, death) ~
                    age + sex + cotinine + I(cotinine ^ 2)))) -> rslt
  rslt$pooled[rslt$pooled$term == "cotinine",]}

lapply(1:10, cotcof) -> cotcof5rslt
lapply(11:20, cotcof, m= 20) -> cotcof20rslt

bind_cols(
  do.call(rbind.data.frame, cotcof5rslt) |>
    transmute(est5 = estimate * 10000,
              se5 = 10^7 * sqrt(ubar + ((m+1) * b/m)),
              re5 = 100 * m/(m + fmi) ),
  do.call(rbind.data.frame, cotcof20rslt) |>
    transmute(est20 = estimate * 10000,
```

```
          se20 = 10^7 * sqrt(ubar + ((m+1) * b/m)),
          re20 = 100 * m/(m + fmi) ))
```

First, we create a custom function which will accept a random seed and then impute our data frame using `mice`, conduct regression as specified, `pool` the results and return the row corresponding to *cotinine*. We call this function repeatedly using `lapply`. Each time it is called, it will get a new seed from the first argument, the sequence from 1 to 10 and 11 to 20, given to `lapply`. The `lapply` is used twice, once with the imputation number set to 5 and once to 20. The results returned by `lapply` are row bound as a dataframe and modified using `transmute` to create new columns and remove all others. This sequence is done for both the `lapply` results and are bound together by `bind_cols`.

TABLE 14.9
Replication of table 14.11

M = 5			M = 20		
Estimate[a]	SE[b]	Relative efficiency (%)	Estimate[a]	SE[b]	Relative efficiency (%)
45.157	4717.3	97.407	44.445	4514.9	99.566
45.665	4592.9	98.438	44.704	4495.5	99.653
44.190	4538.3	98.906	44.531	4578.7	99.470
44.087	4670.0	96.974	44.874	4600.3	99.522
44.755	4890.7	95.569	45.156	4754.4	99.297
44.686	4800.0	96.419	44.574	4608.7	99.434
46.594	4905.8	95.954	44.211	4546.3	99.506
43.618	4591.2	97.624	44.627	4555.8	99.576
44.649	4536.6	98.123	44.822	4515.3	99.645
44.021	4664.0	96.988	44.537	4611.0	99.471

[a] Results multiplied by ten thousand
[b] Results multiplied by ten million

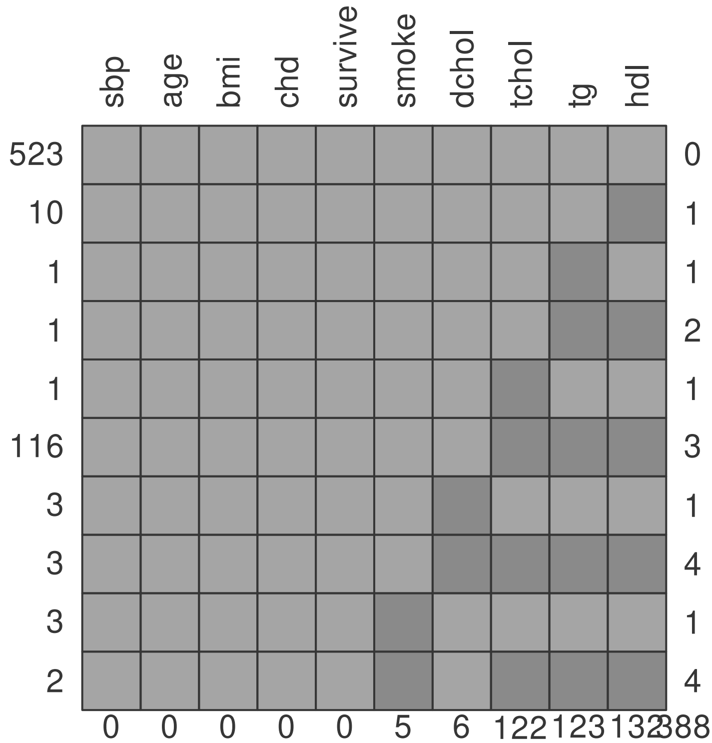
14.9 Multivariate multiple imputation

Example 14.20 (page 737)

We now turn to example 14.20

```
read_table("K11828 supplements/Datasets/Example 14.20.dat",
           col_names = c("sbp", "age", "bmi", "chd","survive",
                         "dchol","hdl","tg", "tchol", "smoke"),
           na = ".") -> data_14_20
md.pattern(data_14_20, rotate.names = TRUE)
```

First, we inspect the missing pattern using `md.pattern`. We supply `rotate.names = TRUE` so that the labels are vertical – to avoid overlapping. This graph contains all the information that is shown in Output 14.8. In addition to the plot which is similar to that obtained by plotting the result of `aggr` of `VIM`, `md.pattern` also returns the numbers as a matrix. We may save that, say, if we want to calculate percentages of missing.




```
    sbp age bmi chd survive smoke dchol tchol  tg hdl
523   1   1   1   1       1     1     1     1   1   1   0
10    1   1   1   1       1     1     1     1   1   0   1
1     1   1   1   1       1     1     1     1   0   1   1
1     1   1   1   1       1     1     1     1   0   0   2
1     1   1   1   1       1     1     1     0   1   1   1
116   1   1   1   1       1     1     1     0   0   0   3
3     1   1   1   1       1     1     0     1   1   1   1
3     1   1   1   1       1     1     0     0   0   0   4
3     1   1   1   1       1     0     1     1   1   1   1
2     1   1   1   1       1     0     1     0   0   0   4
      0   0   0   0       0     5     6   122 123 132 388
```

We may use `cor` to calculate and display the correlation between different variables.

```
cor(data_14_20[, c("tchol", "hdl", "tg","dchol","age", "sbp", "bmi")],
    use = "pairwise.complete.obs",
    method = "spearman") |>
```

```
  data.frame()  |>
  rownames_to_column()
```

We have specified `use = "pairwise.complete.obs"` to ask `cor` to compute correlation using only those cases where the variables under consideration both have non NA values. The `method = "spearman"` instructs `cor` to use the Spearman correlation coefficient.

```
        tchol         hdl       tg     dchol     age         sbp      bmi
tchol 1.00000  0.29368473  0.41298 0.1096478 0.49391 0.31182119  0.19573
hdl   0.29368  1.00000000 -0.32649 0.0073132 0.10273 0.00063938 -0.18563
tg    0.41298 -0.32649061  1.00000 0.1162353 0.39421 0.31810453  0.37888
dchol 0.10965  0.00731321  0.11624 1.0000000 0.10772 0.09092713  0.09005
age   0.49391  0.10273042  0.39421 0.1077201 1.00000 0.50507877  0.22124
sbp   0.31182  0.00063938  0.31810 0.0909271 0.50508 1.00000000  0.29136
bmi   0.19573 -0.18563472  0.37888 0.0900502 0.22124 0.29136238  1.00000
```

The function `aggregate` can be used to calculate the mean cholesterol, aggregating the values according to *chd* or *smoke*.

```
aggregate(tchol ~ chd, FUN = mean, data = data_14_20)
aggregate(tchol ~ smoke, FUN = mean, data = data_14_20)
```

```
  chd  tchol
1   0 6.3250
2   1 7.2732
```

```
  smoke  tchol
1     0 6.3230
2     1 6.6664
```

We will now try to do the actual imputation.

```
mutate(data_14_20, ltg = log(tg)) -> data_14_20
mice(select(data_14_20, -c("tg")),
     printFlag = FALSE,
     m = 20,
     seed = 567,
     maxit = 0,
     method = "norm") -> mi_14_20
mi_14_20$post -> pst
trimupper <- function(x) {ifelse(round(x) > 1,1,round(x))}
trimlower <- function(x) {ifelse(x < 0 ,0,x)}


"imp[[j]][,i] <- trimlower(imp[[j]][,i])" -> pst["dchol"]
```

```
"imp[[j]][,i] <- trimlower(imp[[j]][,i])" -> pst["hdl"]
"imp[[j]][,i] <- trimlower(imp[[j]][,i])" -> pst["tchol"]
"imp[[j]][,i] <- trimupper(imp[[j]][,i])" -> pst["smoke"]

mice(select(data_14_20, -c("tg")),
     printFlag = FALSE,
     m = 20,
     seed = 678,
     maxit = 20,
     method = "norm",
     post = pst) -> mi_14_20
```

First, we `mutate` the data to log transform the triglyceride value. Next, we use `mice` to do a mock imputation. We use the arguments we need, but keep `maxit`, the number of iterations zero. This is stored for the next step, where we extract the `post` object of the result given by `mice` after the mock imputation. The argument `post` decides what post processing is done on the imputed values. Though, by default it is a simple character vector with empty strings for all its elements, it is easier to get it from a mock imputation and modify rather than make it by hand from scratch. Next, we define two custom functions. One will convert its argument to zero if it is less than zero, otherwise it will return the original value. The second function, following the textbook, is to convert the imputed values for *smoke* to zero or one, by rounding all values to the nearest integer. However, I doubt that this method is fool proof. It can round to nearest negative numbers too. Any way, we will take the path laid out in the textbook.

Next, we change the value of the four elements of *pst*. It is changed to a string that assigns `imp[[j]][,i]` the value returned by one of our custom functions when supplied the original value of `imp[[j]][,i]`. The `imp[[j]][,i]` is the jth imputation, ith column. This string is executed to obtain new values from the imputed value. The name `post` refers to post processing.

Now, we do the actual imputation. This time increasing the `miter` to 20, and specifying the value of `post` as *pst*.

To obtain a trace plot, we need to supply the mice object to `plot`.

```
plot(mi_14_20, tchol ~ .it | .ms, layout = c(1,2))
```

By default, trace plots of mean and sd of imputed values for all the variables will be plotted. We can specify a formula to change this behaviour. Here, we ask to plot the *tchol* against `.it` in two panels – one for means and one for standard deviation. The `.it` and `.ms` are two variables made available by the function. The `.it` identifies the iteration and `.ms` identifies the value as mean or standard deviation. Each line plots the value for one imputation as it varies over the iterations we have specified. The option `layout = c(1,2)` is used to tell that we want the panels arranged in one column and two rows.

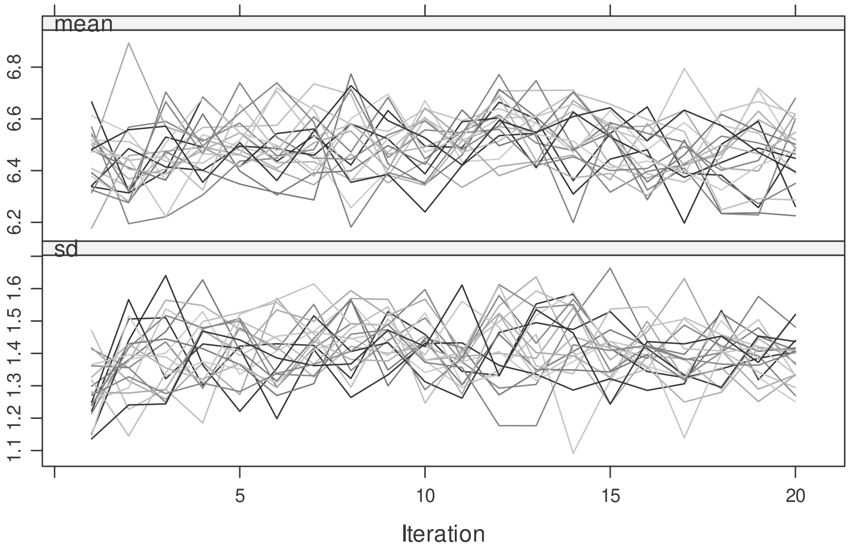


FIGURE 14.12
Replication of figure 14.12

We can use `densityplot` to plot kernel density plots.

```
densityplot(mi_14_20, ~ tchol,
            xlab = "Total cholesterol (mmol/l)",
            ylab = "Kernel density")
```

We pass the mice object to `densityplot` and specify the details via a formula. For `densityplot`, the left-hand side of the formula is empty. The variables for which we want to plot density is specified on the right-hand side. Here, we need only *tchol*. We get density for each of the 20 imputations overlaid on top of the density for the observed values. If we want each of the curves on a different panel, we can add `| . imp` to the formula.

While R does provide an `acf` function to plot correlograms, it does not handle mice objects directly. Also, I couldn't understand which value is used for plotting the correlogram. If it is the mean of values averaged over each imputation, then we can use the code below to get the correlogram.

```
acf(colMeans(mi_14_20$imp$tchol),
    main = NA)
```

Here, we are using the imputed values of *tchol* extracted from the `.imp` component of the mice object. It is structured as a dataframe with one column for each of the imputations, with as many rows as the number of missing values. The `colMeans` returns a vector with the calculated mean for all columns in the data frame supplied to it. Thus, we are passing a numerical vector to `acf` which calculates the correlation between lagged values and plots them.

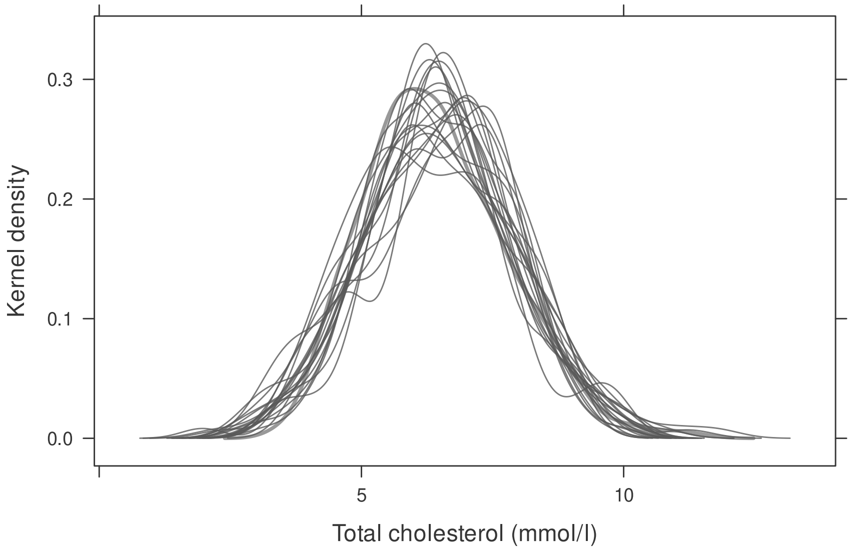


FIGURE 14.13
Replication of figure 14.14

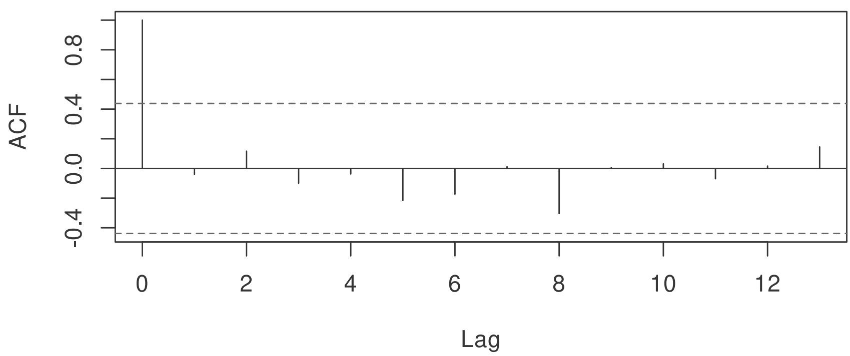


FIGURE 14.14
Replication of figure 14.13

Table 14.12 (page 740)

We will now try to build table 14.12.

```
with(mi_14_20,
     coxph(Surv(survive, chd) ~age + I(sbp/10)+ bmi + smoke + tchol)) |>
  pool() -> rslt_mi_14_20

bind_rows(tidy(rslt_mi_14_20,conf.int = TRUE,exponentiate = TRUE) |>
            mutate(identifier = "impute"),
          tidy(coxph(Surv(survive, chd) ~age + I(sbp/10)+ bmi+smoke+tchol,
                     data = data_14_20),
               conf.int = TRUE,
```

```
              exponentiate = TRUE) |>
            mutate(identifier = "complete")) |>
  select(c("term", "estimate","conf.low","conf.high",
           "p.value","identifier")) |>
  pivot_wider(names_from = identifier,
              values_from = c("estimate", "conf.low","conf.high", "p.value"),
              names_sep = "\n",
              names_vary = "slowest")
```

We pool the result of regression and store it. We row bind the result of the pooled result and the result from complete case analysis after using `tidy` to convert the result to a dataframe. We tell `tidy` to exponentiate the estimates and also to provide a confidence interval. We also add an identifier column before binding the results. From the combined results, we select the required columns and convert it to wide format. We specify `name_sep` as `"\n"` so that the column headings formed from the values in *identifier* and the column names specified in `values_from` are joined together using a newline to ensure that it is not too wide to print. The argument `names_vary` is used to determine the order in which the column names resulting from widening are combined. We want the columns from one type of analysis to be together rather than be separated by the columns from the other analysis. So, we opt for `"slowest"`.

TABLE 14.10
Replication of table 14.12

	Complete cases				Multiple imputation			
Variable (units)	Estimate	95% CI high	95% CI low	p	Estimate	95% CI high	95% CI low	p
age	1.0937	1.05805	1.1306	0.00000080377	1.0940	1.05283	1.1367	0.0000043824
I(sbp/10)	1.0495	0.94893	1.1608	0.34224621589	1.0468	0.93264	1.1750	0.4372858342
bmi	1.0765	1.02238	1.1335	0.00569220828	1.0768	1.01115	1.1467	0.0211342578
smoke	1.5292	0.96225	2.4303	0.07174524459	1.5199	0.88815	2.6010	0.1266961785
tchol	1.2396	1.02010	1.5062	0.03140511451	1.2469	1.02129	1.5222	0.0302470823

Example 14.22 (page 746)

We now turn to example 14.22.

```
data_14_20 |>
  mutate(bchol = ifelse(tchol >= 6.5, TRUE,FALSE)) |>
  select(-c("tchol", "tg")) -> data_14_22
mice(data_14_22,
     printFlag = FALSE,
     m = 20,
     maxit = 0)$method -> mthd
"logreg" -> mthd["bchol"]

mice(data_14_22,
     printFlag = FALSE,
```

```
     m = 20,
     seed = 789,
     maxit = 20,
     method = mthd) -> mi_14_22_lg

mice(data_14_22,
     printFlag = FALSE,
     m = 20,
     seed = 890,
     maxit = 20,
     method = "norm") -> mi_14_22_nr

with(mi_14_22_lg,
     coxph(Surv(survive, chd) ~age + I(sbp/10)+ bmi + smoke + bchol)) |>
  pool() -> rslt_mi_14_22_lg

with(mi_14_22_nr,
     coxph(Surv(survive, chd) ~age + I(sbp/10)+ bmi + smoke + bchol)) |>
  pool() -> rslt_mi_14_22_nr

bind_cols(Method = c("Complete cases",
                     "Data Augmentation",
                     "Chained equations"),
          bind_rows(
            filter(
              tidy(
                coxph(Surv(survive, chd) ~age+I(sbp/10)+ bmi+smoke+bchol,
                      data = data_14_22),
                conf.int = TRUE),
              term == "bcholTRUE"),
            filter(tidy(rslt_mi_14_22_nr,
                        conf.int = TRUE),
                   term == "bchol"),
            filter(tidy(rslt_mi_14_22_lg,
                        conf.int = TRUE),
                   term == "bchol")) |>
            select(c("estimate", "std.error","conf.low",
                     "conf.high","p.value")))
```

First, we `mutate` the data to create the new binary cholesterol value. Then, we run a mock imputation and select the `method` component from it. We modify it to make the value for *bchol* to `"logreg"` to mean logistic regression. Now, we impute values supplying the modified *mthd* as `method`. We also perform another imputation where the `method` is `"norm"` for all variables. We `pool` the result for both the imputations and save it. We pass these results to `tidy`, asking it to give the confidence intervals, select the row for *bchol* and row bind them with the corresponding row from complete case analysis. We also add a column to identify the methods.

```
# A tibble: 3 x 6
```

```
  Method             estimate std.error conf.low conf.high p.value
  <chr>                 <dbl>     <dbl>    <dbl>     <dbl>   <dbl>
1 Complete cases        0.612     0.306   0.0128      1.21  0.0453
2 Data Augmentation     0.598     0.329  -0.0709      1.27  0.0780
3 Chained equations     0.549     0.293  -0.0392      1.14  0.0667
```

14.10 Recap

14.10.1 Commands introduced in this chapter

- boot::boot
- ggplot2::geom_bar
- ggplot2::guide_axis
- ggplot2::scale_x_binned
- ggplot2::geom_qq
- boot::boot.ci
- base::as.character
- base::do.call
- base::rbind.data.frame
- ggplot2::theme_void
- stats::as.formula
- base::is.null
- base::dim
- readr::read_delim
- base::return
- lmtest::coefci
- sandwich::vcovHC
- DescTools::Permn
- coin::independence_test
- coin::approximate
- base::replicate
- coin::wilcox_test
- VIM::marginplot
- base::anyNA
- stats::na.omit
- VIM::regressionImp
- graphics::clip
- graphics::abline
- VIM::hotdeck
- mice::mice
- mice::complete
- dplyr::starts_with
- dplyr::slice
- mice::with
- dplyr::num_range
- mice::pool.scalar
- mice::stripplot
- mice::md.pattern
- graphics::densityplot
- stats::acf
- mice::pool

Final Words

In this final chapter, I want to give you some tips to hone your R skill.

Literate programming

Many a times in this book, you will come across situations where I used code to refer to small piece of information that could have been copied and pasted. When we copy and paste, there are chances of error like leaving out a digit. The real problem surfaces when situations demand us to revise or repeat calculations after correcting errors in the original data. How can we be sure that we changed the values at all places? It is not easy. The answer to this problem is literate programming. Literate programming refers to mixing code within the prose we write. One of R's solution is by means of an Rmd file, to mean R markdown file. We write a plain text file with certain markups to indicate formatting or more importantly code. We use a package `knitr`, which will run the code and replace it with its result in the final output. If we develop the habit of referring all results using code, we need not worry about changing it at any place; all we need is rerun the Rmd file using `knitr`. I am not going to give you further details as there are excellent tutorials out there to cover this easy topic. A favourite of mine is the free online tutorial called "knitr in a knutshell" available at https://kbroman.org/knitr_knutshell/. I strongly recommend you to go through this tutorial as support for literate programming is one of the best part of R and you need to understand it clearly to achieve efficiency.

Presentation

Throughout this book, the focus has been on reproducing the results, not exactly on presenting them. When we adapt literate programming, we need to take care about presentation using R code as well. Though we have discussed graphs in some detail, we have just touched upon tables. Of the many packages available to format tables, I have introduced `tinytable`. However, I have not gone into the details. The user guide for the package available at https://vincentarelbundock.github.io/tinytable/#tutorial is the source to master the package. There are many other packages for the same purpose including `flextable`, `kable`, `kableExtra` and `huxtable`. While I was preparing this book, I was using `flextable`, but changed to `tinytable` at the very end as the LaTeX code generated by it was rather bulky. At the time of writing this book, `tinytable` version was still under 1. But, it was very usable and I recommend it over `flextable` especially if the output you want is LaTeX.

The package `broom` which we used many times is another package helpful for presentation. It converts many results, especially of modelling, to dataframes which we may format using `tinytable`.

R

While we have covered a lot of R, I have not gone through in conventional order. I would suggest you to go through the short "An introduction to R" that comes with R and also available at https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf. In under 100 pages it gives a succinct introduction to R. While we have covered most of the topics included in it, reading it should help you organise your R knowledge.

The `tidyverse` is a relatively new alternative to the base R methods. In our book, we have used both base R and `tidyverse` methods. The book "R for Data Science" available at https://r4ds.hadley.nz/ is an excellent source to consolidate what you have learned here about tidyverse.

Help

We saw about the inbuilt help. Most packages also have vignettes, which teach you the typical use of the packages, how the various functions that the package provide are related to each other. The command `vignette` lists all the vignettes from all the installed packages. It also allows you to restrict to one vignette or package if you specify them. To know about a package, especially a new one, you need to check its vignette. When you know the package, but want details of a specific function, then you need to use the `?` function.

When you cannot find the answers to your questions from the above sources, there are many online communities to turn to. The best, I think, are the Stack Exchange sites Stack Overflow (https://stackoverflow.com/) and Cross Validated (https://stats.stackexchange.com/).

Advanced R

Hands-On Programming with R: Write Your Own Functions and Simulations available at https://rstudio-education.github.io/hopr/, Advanced R available at https://adv-r.hadley.nz/ and R Packages: Organize, Test, Document, and Share Your Code available at https://r-pkgs.org/ are some of the books for you to consider when you want to take your R skills to the next level.

Recap

Command introduced in this chapter

utils::vignette

Command Index

Note: The index is sorted by the command names though the command names are prefixed with the package names.

Subject Index

Note: Topics related to R programming are shown in **bold**.