

An R  
COMPANION  
to APPLIED  
REGRESSION

3<sup>rd</sup>  
edition



JOHN FOX  
SANFORD WEISBERG



# **An R Companion to Applied Regression**

**Third Edition**



*To the memory of my parents, Joseph and Diana*

*—J. F.*

*For my teachers, and especially Fred Mosteller, who I think would have  
liked this book*

*—S. W.*



# An R Companion to Applied Regression

Third Edition

**John Fox**

McMaster University

**Sanford Weisberg**

University of Minnesota



Los Angeles | London | New Delhi  
Singapore | Washington DC | Melbourne

Los Angeles

London

New Delhi

Singapore

Washington DC

Melbourne



Copyright © 2019 by SAGE Publications, Inc.

All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the publisher.



FOR INFORMATION:

SAGE Publications, Inc.

**2455 Teller Road**

Thousand Oaks, California 91320

E-mail: [order@sagepub.com](mailto:order@sagepub.com)

SAGE Publications Ltd.

1 Oliver's Yard

**55 City Road**

London, EC1Y 1SP

United Kingdom

SAGE Publications India Pvt. Ltd.

B 1/I 1 Mohan Cooperative Industrial Area Mathura Road, New Delhi 110 044  
India

SAGE Publications Asia-Pacific Pte. Ltd.

**3 Church Street**

#10-04 Samsung Hub

Singapore 049483

ISBN: 978-1-5443-3647-3

Printed in the United States of America This book is printed on acid-free paper.

Acquisitions Editor: Helen Salmon Editorial Assistant: Megan O'Heffernan

Production Editor: Kelly DeRosa Copy Editor: Gillian Dickens

Typesetter: QuADS Prepress (P) Ltd Proofreader: Jen Grubba

Cover Designer: Anthony Paular Marketing Manager: Susannah Goldes

# Contents

## [Preface](#)

### [What Is R?](#)

[Obtaining and Installing R and RStudio](#)

[Installing R on a Windows System](#)

[Installing R on a macOS System](#)

[Installing RStudio](#)

[Installing and Using R Packages](#)

[Optional: Customizing R](#)

[Optional: Installing LATEX](#)

### [Using This Book](#)

[Chapter Synopses](#)

[Typographical Conventions](#)

### [New in the Third Edition](#)

### [The Website for the R Companion](#)

### [Beyond the R Companion](#)

### [Acknowledgments](#)

## [About the Authors](#)

### [1 Getting Started With R and RStudio](#)

#### [1.1 Projects in RStudio](#)

#### [1.2 R Basics](#)

[1.2.1 Interacting With R Through the Console](#)

[1.2.2 Editing R Commands in the Console](#)

[1.2.3 R Functions](#)

[1.2.4 Vectors and Variables](#)

[1.2.5 Nonnumeric Vectors](#)

[1.2.6 Indexing Vectors](#)

[1.2.7 User-Defined Functions](#)

#### [1.3 Fixing Errors and Getting Help](#)

[1.3.1 When Things Go Wrong](#)

[1.3.2 Getting Help and Information](#)

#### [1.4 Organizing Your Work in R and RStudio and Making It Reproducible](#)

[1.4.1 Using the RStudio Editor With R Script Files](#)

[1.4.2 Writing R Markdown Documents](#)

#### [1.5 An Extended Illustration: Duncan's Occupational-Prestige Regression](#)

[1.5.1 Examining the Data](#)

[1.5.2 Regression Analysis](#)

[1.5.3 Regression Diagnostics](#)

[1.6 R Functions for Basic Statistics](#)

[1.7 Generic Functions and Their Methods\\*](#)

## [2 Reading and Manipulating Data](#)

[2.1 Data Input](#)

[2.1.1 Accessing Data From a Package](#)

[2.1.2 Entering a Data Frame Directly](#)

[2.1.3 Reading Data From Plain-Text Files](#)

[2.1.4 Files and Paths](#)

[2.1.5 Exporting or Saving a Data Frame to a File](#)

[2.1.6 Reading and Writing Other File Formats](#)

[2.2 Other Approaches to Reading and Managing Data Sets in R](#)

[2.3 Working With Data Frames](#)

[2.3.1 How the R Interpreter Finds Objects](#)

[2.3.2 Missing Data](#)

[2.3.3 Modifying and Transforming Data](#)

[2.3.4 Binding Rows and Columns](#)

[2.3.5 Aggregating Data Frames](#)

[2.3.6 Merging Data Frames](#)

[2.3.7 Reshaping Data](#)

[2.4 Working With Matrices, Arrays, and Lists](#)

[2.4.1 Matrices](#)

[2.4.2 Arrays](#)

[2.4.3 Lists](#)

[2.4.4 Indexing](#)

[2.5 Dates and Times](#)

[2.6 Character Data](#)

[2.7 Large Data Sets in R\\*](#)

[2.7.1 How Large Is “Large”?](#)

[2.7.2 Reading and Saving Large Data Sets](#)

[2.8 Complementary Reading and References](#)

## [3 Exploring and Transforming Data](#)

[3.1 Examining Distributions](#)

[3.1.1 Histograms](#)

[3.1.2 Density Estimation](#)

[3.1.3 Quantile-Comparison Plots](#)

[3.1.4 Boxplots](#)

[3.2 Examining Relationships](#)

[3.2.1 Scatterplots](#)  
[3.2.2 Parallel Boxplots](#)  
[3.2.3 More on the plot\(\) Function](#)

[3.3 Examining Multivariate Data](#)  
[3.3.1 Three-Dimensional Plots](#)  
[3.3.2 Scatterplot Matrices](#)

[3.4 Transforming Data](#)  
[3.4.1 Logarithms: The Champion of Transformations](#)  
[3.4.2 Power Transformations](#)  
[3.4.3 Transformations and Exploratory Data Analysis](#)  
[3.4.4 Transforming Restricted-Range Variables](#)  
[3.4.5 Other Transformations](#)

[3.5 Point Labeling and Identification](#)  
[3.5.1 The identify\(\) Function](#)  
[3.5.2 Automatic Point Labeling](#)

[3.6 Scatterplot Smoothing](#)  
[3.7 Complementary Reading and References](#)

## [4 Fitting Linear Models](#)

[4.1 The Linear Model](#)  
[4.2 Linear Least-Squares Regression](#)  
[4.2.1 Simple Linear Regression](#)  
[4.2.2 Multiple Linear Regression](#)  
[4.2.3 Standardized Regression Coefficients](#)

[4.3 Predictor Effect Plots](#)  
[4.4 Polynomial Regression and Regression Splines](#)  
[4.4.1 Polynomial Regression](#)  
[4.4.2 Regression Splines\\*](#)

[4.5 Factors in Linear Models](#)  
[4.5.1 A Linear Model With One Factor: One-Way Analysis of Variance](#)  
[4.5.2 Additive Models With Numeric Predictors and Factors](#)

[4.6 Linear Models With Interactions](#)  
[4.6.1 Interactions Between Numeric Predictors and Factors](#)  
[4.6.2 Shortcuts for Writing Linear-Model Formulas](#)  
[4.6.3 Multiple Factors](#)  
[4.6.4 Interactions Between Numeric Predictors\\*](#)

[4.7 More on Factors](#)  
[4.7.1 Dummy Coding](#)  
[4.7.2 Other Factor Codings](#)

[4.7.3 Ordered Factors and Orthogonal-Polynomial Contrasts](#)

[4.7.4 User-Specified Contrasts\\*](#)

[4.7.5 Suppressing the Intercept in a Model With Factors\\*](#)

[4.8 Too Many Regressors\\*](#)

[4.9 The Arguments of the lm\(\) Function](#)

[4.9.1 formula](#)

[4.9.2 data](#)

[4.9.3 subset](#)

[4.9.4 weights](#)

[4.9.5 na.action](#)

[4.9.6 method, model, x, y, qr\\*](#)

[4.9.7 singular.ok\\*](#)

[4.9.8 contrasts](#)

[4.9.9 offset](#)

[4.10 Complementary Reading and References](#)

[5 Coefficient Standard Errors, Confidence Intervals, and Hypothesis Tests](#)

[5.1 Coefficient Standard Errors](#)

[5.1.1 Conventional Standard Errors of Least-Squares Regression Coefficients](#)

[5.1.2 Robust Regression Coefficient Standard Errors](#)

[5.1.3 Using the Bootstrap to Compute Standard Errors](#)

[5.1.4 The Delta Method for Standard Errors of Nonlinear Functions\\*](#)

[5.2 Confidence Intervals](#)

[5.2.1 Wald Confidence Intervals](#)

[5.2.2 Bootstrap Confidence Intervals](#)

[5.2.3 Confidence Regions and Data Ellipses\\*](#)

[5.3 Testing Hypotheses About Regression Coefficients](#)

[5.3.1 Wald Tests](#)

[5.3.2 Likelihood-Ratio Tests and the Analysis of Variance](#)

[5.3.3 Sequential Analysis of Variance](#)

[5.3.4 The Anova\(\) Function](#)

[5.3.5 Testing General Linear Hypotheses\\*](#)

[5.4 Complementary Reading and References](#)

[6 Fitting Generalized Linear Models](#)

[6.1 Review of the Structure of GLMs](#)

[6.2 The glm\(\) Function in R](#)

[6.3 GLMs for Binary Response Data](#)

[6.3.1 Example: Women's Labor Force Participation](#)

[6.3.2 Example: Volunteering for a Psychological Experiment](#)

[6.3.3 Predictor Effect Plots for Logistic Regression](#)

[6.3.4 Analysis of Deviance and Hypothesis Tests for Logistic Regression](#)

[6.3.5 Fitted and Predicted Values](#)

[6.4 Binomial Data](#)

[6.5 Poisson GLMs for Count Data](#)

[6.6 Loglinear Models for Contingency Tables](#)

[6.6.1 Two-Dimensional Tables](#)

[6.6.2 Three-Dimensional Tables](#)

[6.6.3 Sampling Plans for Loglinear Models](#)

[6.6.4 Response Variables](#)

[6.7 Multinomial Response Data](#)

[6.8 Nested Dichotomies](#)

[6.9 The Proportional-Odds Model](#)

[6.9.1 Testing for Proportional Odds](#)

[6.10 Extensions](#)

[6.10.1 More on the Anova \(\) Function](#)

[6.10.2 Gamma Models](#)

[6.10.3 Quasi-Likelihood Estimation](#)

[6.10.4 Overdispersed Binomial and Poisson Models](#)

[6.11 Arguments to glm\(\)](#)

[6.11.1 weights](#)

[6.11.2 start, etastart, mustart](#)

[6.11.3 offset](#)

[6.11.4 control](#)

[6.11.5 model, method, x, y](#)

[6.12 Fitting GLMs by Iterated Weighted Least Squares\\*](#)

[6.13 Complementary Reading and References](#)

## [7 Fitting Mixed-Effects Models](#)

[7.1 Background: The Linear Model Revisited](#)

[7.1.1 The Linear Model in Matrix Form\\*](#)

[7.2 Linear Mixed-Effects Models](#)

[7.2.1 Matrix Form of the Linear Mixed-Effects Model\\*](#)

[7.2.2 An Application to Hierarchical Data](#)

[7.2.3 Wald Tests for Linear Mixed-Effects Models](#)

[7.2.4 Examining the Random Effects: Computing BLUPs](#)

[7.2.5 An Application to Longitudinal Data](#)

[7.2.6 Modeling the Errors](#)

## [7.2.7 Sandwich Standard Errors for Least-Squares Estimates](#)

### [7.3 Generalized Linear Mixed Models](#)

#### [7.3.1 Matrix Form of the GLMM\\*](#)

#### [7.3.2 Example: Minneapolis Police Stops](#)

### [7.4 Complementary Reading](#)

## [8 Regression Diagnostics for Linear, Generalized Linear, and Mixed-Effects Models](#)

### [8.1 Residuals](#)

### [8.2 Basic Diagnostic Plots](#)

#### [8.2.1 Plotting Residuals](#)

#### [8.2.2 Marginal-Model Plots](#)

#### [8.2.3 Added-Variable Plots](#)

#### [8.2.4 Marginal-Conditional Plots](#)

### [8.3 Unusual Data](#)

#### [8.3.1 Outliers and Studentized Residuals](#)

#### [8.3.2 Leverage: Hat-Values](#)

#### [8.3.3 Influence Measures](#)

### [8.4 Transformations After Fitting a Regression Model](#)

#### [8.4.1 Transforming the Response](#)

#### [8.4.2 Predictor Transformations](#)

### [8.5 Nonconstant Error Variance](#)

#### [8.5.1 Testing for Nonconstant Error Variance](#)

### [8.6 Diagnostics for Generalized Linear Models](#)

#### [8.6.1 Residuals and Residual Plots](#)

#### [8.6.2 Influence Measures](#)

#### [8.6.3 Graphical Methods: Added-Variable Plots, Component-Plus-Residual Plots, and Effect Plots With Partial Residuals](#)

### [8.7 Diagnostics for Mixed-Effects Models](#)

#### [8.7.1 Mixed-Model Component-Plus-Residual Plots](#)

#### [8.7.2 Influence Diagnostics for Mixed Models](#)

### [8.8 Collinearity and Variance Inflation Factors](#)

### [8.9 Additional Regression Diagnostics](#)

### [8.10 Complementary Reading and References](#)

## [9 Drawing Graphs](#)

### [9.1 A General Approach to R Graphics](#)

#### [9.1.1 Defining a Coordinate System: plot\(\)](#)

#### [9.1.2 Graphics Parameters: par\(\)](#)

#### [9.1.3 Adding Graphical Elements: axis\(\), points\(\), lines\(\), text\(\), et al.](#)

- [9.1.4 Specifying Colors](#)
- [9.2 Putting It Together: Explaining Local Linear Regression](#)
  - [9.2.1 Finer Control Over Plot Layout](#)
- [9.3 Other R Graphics Packages](#)
  - [9.3.1 The lattice Package](#)
  - [9.3.2 The ggplot2 Package](#)
  - [9.3.3 Maps](#)
  - [9.3.4 Other Notable Graphics Packages](#)
- [9.4 Complementary Reading and References](#)
- [10 An Introduction to R Programming](#)
  - [10.1 Why Learn to Program in R?](#)
  - [10.2 Defining Functions: Preliminary Examples](#)
    - [10.2.1 Lagging a Variable](#)
    - [10.2.2 Creating an Influence Plot](#)
  - [10.3 Working With Matrices\\*](#)
    - [10.3.1 Basic Matrix Arithmetic](#)
    - [10.3.2 Matrix Inversion and the Solution of Linear Simultaneous Equations](#)
    - [10.3.3 Example: Linear Least-Squares Regression](#)
    - [10.3.4 Eigenvalues and Eigenvectors](#)
    - [10.3.5 Miscellaneous Matrix Computations](#)
  - [10.4 Program Control With Conditionals, Loops, and Recursion](#)
    - [10.4.1 Conditionals](#)
    - [10.4.2 Iteration \(Looping\)](#)
    - [10.4.3 Recursion](#)
  - [10.5 Avoiding Loops: apply \(\) and Its Relatives](#)
    - [10.5.1 To Loop or Not to Loop?](#)
  - [10.6 Optimization Problems\\*](#)
    - [10.6.1 Zero-Inflated Poisson Regression](#)
  - [10.7 Monte-Carlo Simulations\\*](#)
    - [10.7.1 Testing Regression Models Using Simulation](#)
  - [10.8 Debugging R Code\\*](#)
  - [10.9 Object-Oriented Programming in R\\*](#)
  - [10.10 Writing Statistical-Modeling Functions in R\\*](#)
  - [10.11 Organizing Code for R Functions](#)
  - [10.12 Complementary Reading and References](#)
- [References](#)
- [Subject Index](#)
- [Data Set Index](#)

[Package Index](#)

[Index of Functions and Operators](#)

Sara Miller McCune founded SAGE Publishing in 1965 to support the dissemination of usable knowledge and educate a global community. SAGE publishes more than 1000 journals and over 800 new books each year, spanning a wide range of subject areas. Our growing selection of library products includes archives, data, case studies and video. SAGE remains majority owned by our founder and after her lifetime will become owned by a charitable trust that secures the company's continued independence.

Los Angeles | London | New Delhi | Singapore | Washington DC | Melbourne

# Preface

This book aims to provide an introduction to the R statistical computing environment (R Core Team, 2018) in the context of applied regression analysis, which is typically studied by social scientists and others in a second course in applied statistics. We assume that the reader is learning or is otherwise familiar with the statistical methods that we describe; thus, this book is a *companion* to a text or course on modern applied regression, such as, but not necessarily, our own *Applied Regression Analysis and Generalized Linear Models*, third edition (Fox, 2016) and *Applied Linear Regression*, fourth edition (Weisberg, 2014). Of course, different texts and courses have somewhat different content, and different readers will have different needs and interests: If you encounter a topic that is unfamiliar or that is not of interest, feel free to skip it or to pass over it lightly. With a caveat concerning the continuity of examples within chapters, the book is designed to let you skip around and study only the sections you need, providing a reference to which you can turn when you encounter an unfamiliar subject.

The *R Companion* is associated with three R packages, all freely and readily available on the Comprehensive R Archive Network (CRAN, see below): The **car** package includes R functions (programs) for performing many tasks related to applied regression analysis, including a variety of regression graphics; the **effects** package is useful for visualizing regression models of various sorts that have been fit to data; and the **carData** package provides convenient access to data sets used in the book. The **car** and **effects** packages are in very wide use, and in preparing this new edition of the *R Companion* we substantially updated both packages. The book was prepared using Version 3.0-1 of the **car** package, Version 3.0-1 of the **carData** package, and Version 4.0-2 of the **effects** package. You can check the NEWS file for each package, accessible, for example, via the R command news(package="car"), for information about newer versions of these packages released after the publication of the book.

This Preface provides a variety of orienting information, including

- An explanation of what R is and where it came from
- Step-by-step instructions for obtaining and installing R, the RStudio interactive development environment, the R packages associated with this book, and some additional optional software
- Suggestions for using the book, including chapter synopses
- A description of what's new in the third edition of the *R Companion*
- Information about resources available on the website associated with the *R Companion*

## What Is R?

R descended from the S programming language, which was developed at Bell Labs by experts in statistical computing, including John Chambers, Richard Becker, and Allan Wilks (see, e.g., Becker, Chambers, & Wilks, 1988, Preface). Like most good software, S has evolved considerably since its origins in the mid-1970s. Bell Labs originally distributed S directly, which eventually morphed into the commercial product S-PLUS.

R is an independent, open-source, and free implementation and extension of the S language, developed by an international team of statisticians, including John Chambers. As described in Ihaka and Gentleman (1996), what evolved into the R Project for Statistical Computing was originated by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand.<sup>1</sup> A key advantage of the R system is that it is free—simply download and install it, as we will describe shortly, and then use it. R has eclipsed its commercial cousin S-PLUS, which is essentially defunct.

1 It probably hasn't escaped your notice that R is the first letter of both Ross and Robert; this isn't a coincidence. It is also the letter before S in the alphabet.

R is a *statistical computing environment* that includes an *interpreter* for the R programming language, with which the user-programmer can interact in a conversational manner.<sup>2</sup> R is one of several programming environments used to develop statistical applications; others include Gauss, Julia, Lisp-Stat, Python, and Stata (some of which are described in Stine & Fox, 1996).

2 A *compiler* translates a program written in a programming language (called *source code*) into an independently executable program in machine code. In contrast, an *interpreter* translates and executes a program under the control of the interpreter. Although it is in theory possible to write a compiler for a high-level, interactive language such as R, it is difficult to do so. Compiled programs usually execute more efficiently than interpreted programs. In advanced use, R has facilities for incorporating compiled programs written in Fortran, C, and C++.

If you can master the art of typing commands,<sup>3</sup> a good statistical programming environment allows you to have your cake and eat it too. Routine data analysis is convenient, as it is in statistical packages such as SAS or SPSS,<sup>4</sup> but so are programming and the incorporation of new statistical methods. We believe that R balances these factors especially well:

3 Although there are point-and-click *graphical user interfaces (GUIs)* for R—indeed, one of us wrote the most popular GUI for R, called the R Commander (Fox, 2017) and implemented as the **Rcmdr** package—we believe that users

beyond the basic level are better served by learning to write R commands. 4 Traditional statistical packages are largely oriented toward processing rectangular data sets to produce printed reports, while statistical computing environments like R focus on transforming objects, including rectangular data sets, into other objects, such as statistical models. Statistical programming environments, therefore, provide more flexibility to the user.

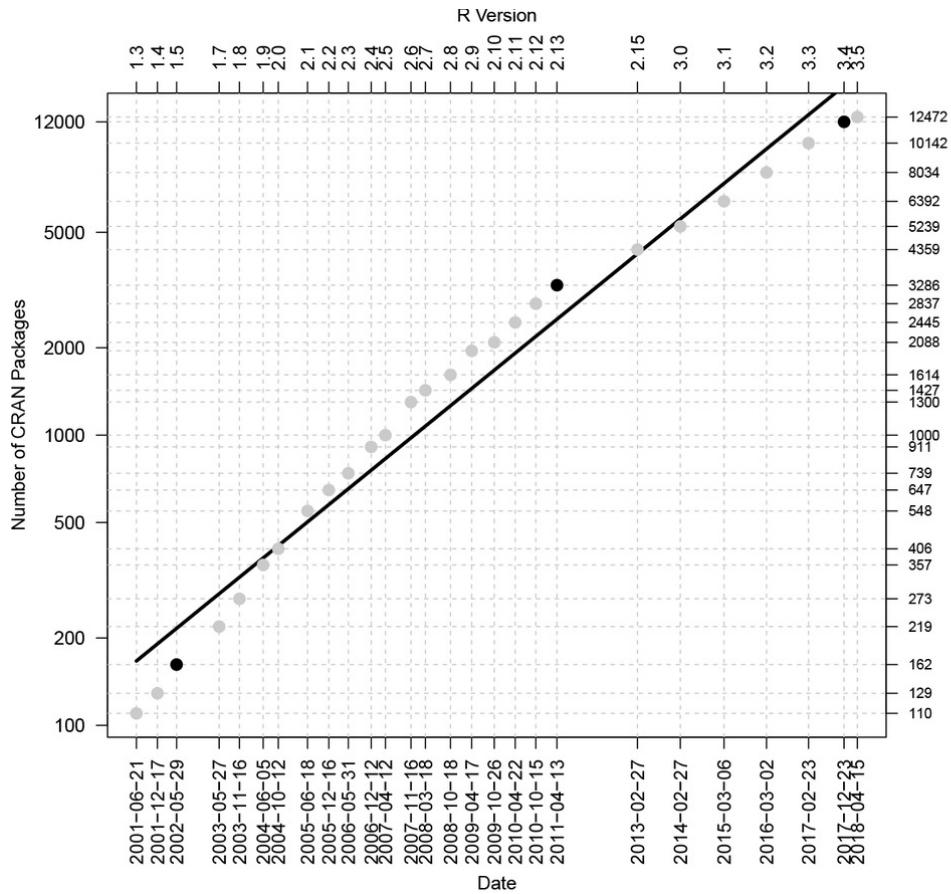
- R is very capable out of the (metaphorical) box, including a wide range of standard statistical applications. Contributed packages, which are generally free and easy to obtain, add to the basic R software, vastly extending the range of routine data analysis both to new general techniques and to specialized methods of interest only to users in particular areas of application.
- Once you get used to it, the R programming language is reasonably easy to use—as easy a programming language as we have encountered—and is finely tuned to the development of statistical applications. We recognize that most readers of this book will be more interested in using existing programs written in R than in writing their own statistical programs. Nevertheless, developing some proficiency in R programming will allow you to work more efficiently, for example, to perform nonstandard data management tasks.
- The S programming language and its descendant R are carefully designed from the point of view of computer science as well as statistics. John Chambers, the principal architect of S, won the 1998 Software System Award of the Association for Computing Machinery (ACM) for the S System. Similarly, in 2010, Robert Gentleman and Ross Ihaka were awarded a prize for R by the Statistical Computing and Statistical Graphics sections of the American Statistical Association.
- The implementation of R is very solid under the hood—incorporating, for example, sound numerical algorithms for statistical computations—and it is regularly updated, currently at least once a year.

One of the great strengths of R is that it allows users and experts in particular areas of statistics to add new capabilities to the software. Not only is it possible to write new programs in R, but it is also convenient to combine related sets of programs, data, and documentation in R *packages*. The first edition of this book, published in 2002, touted the then “more than 100 contributed packages available on the R website, many of them prepared by experts in various areas of applied statistics, such as resampling methods, mixed models, and survival analysis” (Fox, 2002, p. xii). When the second edition of the book was published in 2011, the Comprehensive R Archive Network (abbreviated CRAN and

variously pronounced *see-ran* or *kran*) held more than 2,000 packages. As we write this preface early in 2018, there are more than 12,000 packages on CRAN (see [Figure 1](#), drawn, of course, with R). Other R package archives—most notably the archive of the Bioconductor Project, which develops software for bioinformatics—add more than 2,000 packages to the total. In the statistical literature, new methods are often accompanied by implementations in R; indeed, R has become a kind of *lingua franca* of statistical computing for statisticians and is very widely used in many other disciplines, including the social and behavioral sciences.<sup>5</sup>

5 R packages are contributed to CRAN by their authors, generally without any vetting or refereeing, other than checking that the package can run its own code and examples without generating errors, and can meet a variety of technical requirements for code and documentation. As a consequence, some contributed packages are of higher quality and of greater general interest than others. The Bioconductor package archive checks its packages in a manner similar to (if somewhat more rigorously than) CRAN. There are still other sources of R packages that perform no quality control, most notably the GitHub archive, <https://github.com/>, from which many R packages can be directly installed. The packages that we employ in the *R Companion* are trustworthy and useful, but with other packages, *caveat utilitor*. The *Journal of Statistical Software* (<https://www.jstatsoft.org/>) and the *R Journal* (<https://journal.r-project.org/>), two reputable open-access journals, frequently publish papers on R packages, and these papers and the associated software are refereed.

**Figure 1** The number of packages on CRAN has grown substantially since the first edition of this book was published in 2002 (the black dots correspond roughly to the dates of three editions of the book, the gray dots to other minor releases of R). The vertical axis is on a log scale, so that a linear trend represents exponential growth, and the line on the graph is fit by least squares. It is apparent that, while growth remains robust, the growth rate in the number of CRAN packages has been declining. *Source:* Updated and adapted from Fox (2009a).



## Obtaining and Installing R and RStudio

We assume that you’re working on a single-user computer on which R has not yet been installed and for which you have administrator privileges to install software. To state the obvious, before you can start using R, you have to get it and install it.<sup>6</sup> The good news is that R is free and runs under all commonly available computer operating systems—Windows, macOS, and Linux and Unix—and that precompiled binary distributions of R are available for these systems. The bad news is that R doesn’t run on some locked-down operating systems such as iOS for iPads and iPhones, Android phones or tablets, or ChromeOS

Chromebooks.<sup>7</sup> It is our expectation that most readers of the book will use either the Windows or the macOS implementations of R, and the presentation in the text reflects that assumption. Virtually everything in the text applies equally to Linux and Unix systems (or to using R in a web browser), although the details of installing R vary across specific Linux distributions and Unix systems.

<sup>6</sup> Well, maybe it’s not entirely obvious: Students in a university class or individuals in a company, for example, may have access to R running on an internet server. If you have access to R on the internet, you’ll be able to use it in a web browser and won’t have to install R or RStudio on your own computer,

unless you wish to do so.

<sup>7</sup> In these cases, your only practical recourse is to use R in a web browser, as described in footnote 6. It is apparently possible to install R under Android and ChromeOS but not without substantial difficulty.

The best way to obtain R is by downloading it over the internet from CRAN, at <https://cran.r-project.org/>. It is faster, and better netiquette, to download R from one of the many CRAN mirror sites than from the main CRAN site: Click on the *Mirrors* link near the top left of the CRAN home page and select either the first, *0-Cloud*, mirror (which we recommend), or a mirror near you.

*Warning:* The following instructions are relatively terse and are current as of Version 3.5.0 of R and Version 1.1.456 of RStudio. Some of the details may change, so check for updates on the website for this book (introduced later in the Preface), which may provide more detailed, and potentially more up-to-date, installation instructions. If you encounter difficulties in installing R, there is also troubleshooting information on the website. That said, R installation is generally straightforward, and for most users, the instructions given here should suffice.

## Installing R on a Windows System

Click on the *Download R for Windows* link in the *Download and Install R* section near the top of the CRAN home page. Then click on the *base* link on the *R for Windows* page. Finally, click on *Download R x.y.z for Windows* to download the R Windows installer. *R x.y.z* is the current version of R—for example, R-3.5.0. Here, *x* represents the *major* version of R, *y* the *minor* version, and *z* the *patch* version. New major versions of R appear very infrequently, a new minor version is released each year in the spring, and patch versions are released at irregular intervals, mostly to fix bugs. We recommend that you update your installation of R at least annually, simply downloading and installing the current version.

R installs as a standard Windows application. You may take all the defaults in the installation, although we suggest that you decline the option to *Create a desktop icon*. Once R is installed, you can start it as you would any Windows application, for example, via the Windows start menu. We recommend that you use R through the RStudio interactive development environment, described below.

## Installing R on a macOS System

Click on the *Download R for (Mac) OS X* link in the *Download and Install R* section near the top of the CRAN home page. Click on the *R-x.y.z.pkg* link on the *R for Mac OS X* page to download the R macOS installer. *R x.y.z* is the current version of R—for example, R-3.5.0. Here, *x* represents the *major* version

of R,  $y$  the *minor* version, and  $z$  the *patch* version. New major versions of R appear very infrequently, a new minor version is released each year in the spring, and patch versions are released at irregular intervals, mostly to fix bugs. We recommend that you update your installation of R at least annually, simply downloading and installing the current version.

R installs as a standard macOS application. Just double-click on the downloaded installer package, and follow the on-screen directions. Once R is installed, you can treat it as you would any macOS application. For example, you can keep the R.app program on the macOS dock, from which it can conveniently be launched. We recommend that you use R through the RStudio interactive development environment, described below.

Some R programs, including functions in the **car** package associated with this book, require that the X11 windowing system is installed on your Mac.<sup>8</sup> To install X11, go to the XQuartz website at <https://www.xquartz.org/>, and click on the link *XQuartz-x.y.z.dmg*, where  $x.y.z$  represents the current version of XQuartz. Once it is downloaded, double-click on the XQuartz-x.y.z.dmg disk-image file, and then double-click on XQuartz.pkg to start the XQuartz installer. You may take all of the defaults in the installation. After XQuartz is installed, you must log out of and back into your macOS account—or simply reboot your Mac. *Heads up:* In the future, you should reinstall XQuartz whenever you upgrade macOS to a new major version of the operating system.

<sup>8</sup> To be clear, almost all of the functionality of the **car** package will work even if X11 is *not* installed on your computer.

## Installing RStudio

RStudio is an *interactive development environment (IDE)* for R. Like R, RStudio is open-source software, freely available on the internet for Windows, macOS, Linux, and Unix systems. Regardless of your computing system, the user experience in RStudio is nearly identical.<sup>9</sup>

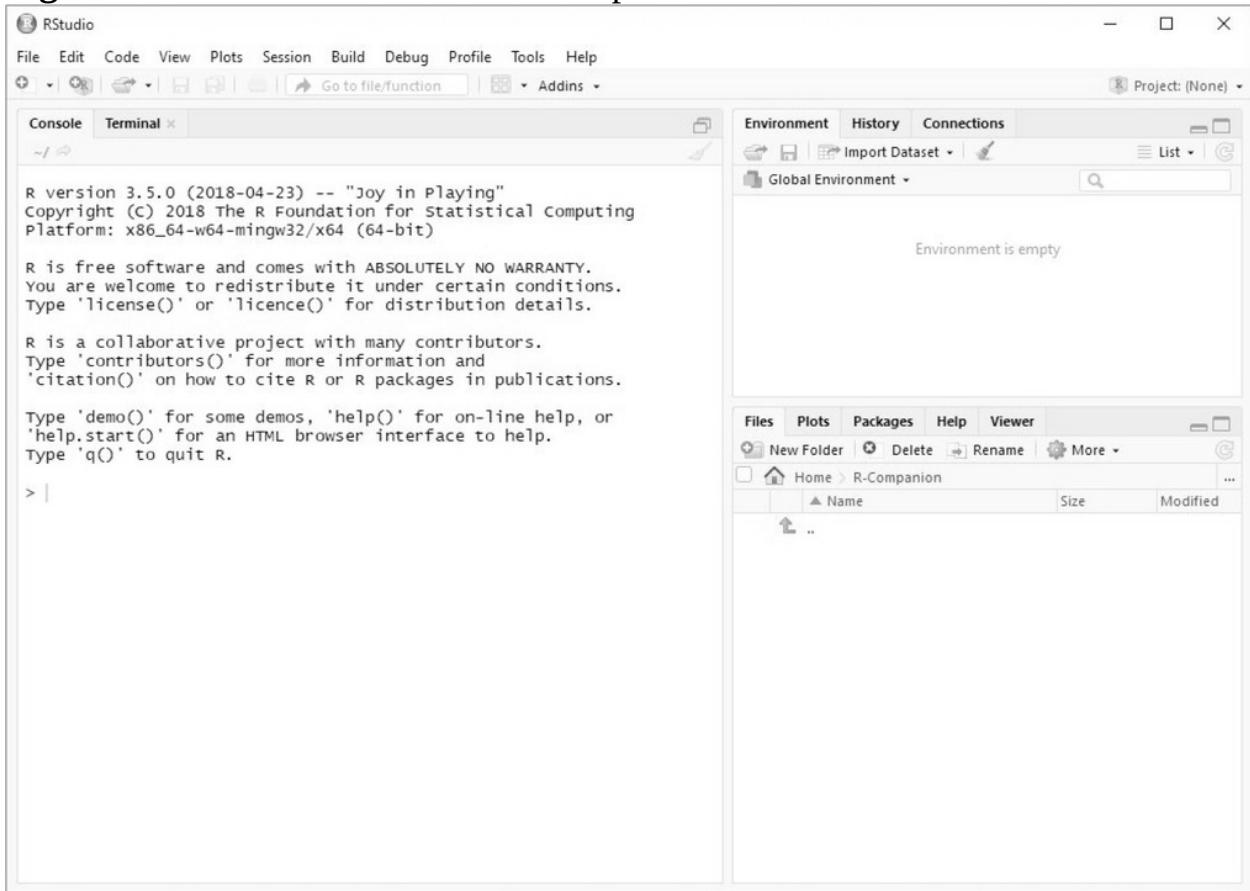
<sup>9</sup> It's also possible to run a server version of RStudio on the internet. In this case, using RStudio remotely via a web browser is, with a few restrictions, similar to using RStudio directly on a computer.

Unlike R, RStudio is developed by a commercial company that also sells R-related products and services. RStudio has excellent facilities for writing R programs and packages, but, more to the point for us, it is also a very convenient environment within which to conduct statistical data analysis in R. We'll explain how to use various facilities in RStudio in the course of this book, chiefly in [Chapter 1](#).

To download RStudio, go to

<https://www.rstudio.com/products/rstudio/download/> and select the installer appropriate to your operating system. The RStudio Windows and macOS installers are entirely standard, and you can accept all of the defaults. You can start RStudio in the normal manner for your operating system: for example, in Windows, click on the start menu and navigate to and select RStudio under *All apps*. On macOS, you can run RStudio from the Launchpad. Because we hope that you'll use it often, we suggest that you pin RStudio to the Windows taskbar or elect to keep it in the macOS dock.

**Figure 2** The RStudio window at startup under Windows.



When you start RStudio, a large window, similar to [Figure 2](#) under Windows or [Figure 3](#) under macOS, opens on your screen.<sup>10</sup> Apart from minor details like font selection and the placement of standard window controls, the RStudio window is much the same for all systems. The RStudio window at startup consists of three *panes*. On the right, the upper of two panes has three tabs, called *Environment*, *History*, and *Connections*, while the lower pane holds several tabs, including *Files*, *Plots*, *Packages*, *Help*, and *Viewer*. Depending on context, other tabs may open during an RStudio session. We'll explain how to use the RStudio interface beginning in [Chapter 1](#).<sup>11</sup> As noted there, your initial

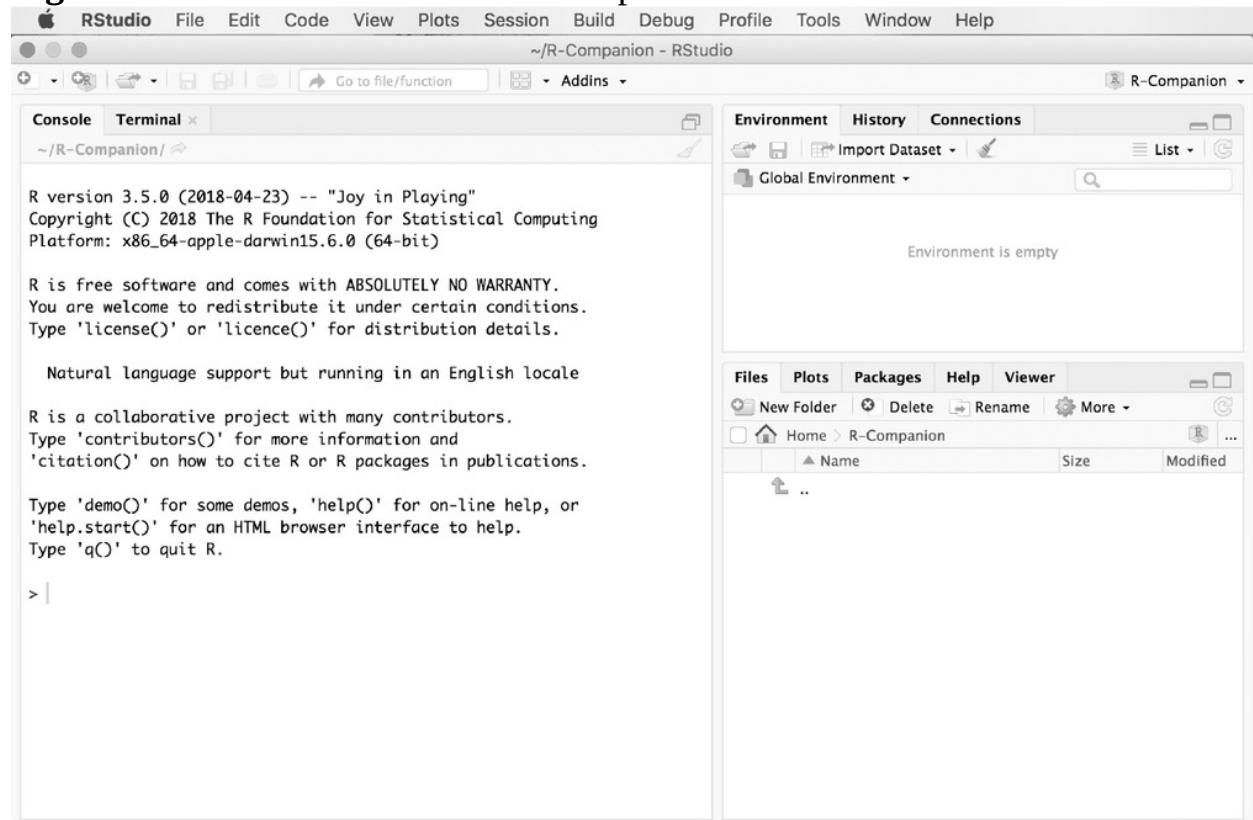
view of the RStudio window may list some file names in the visible *Files* tab. We created and navigated to an empty R-Companion directory.

10 Color images of the RStudio window under Windows and macOS appear in the inside front and back covers of this book.

11 The *Connections* tab, which is used to connect R to database-management systems, isn't discussed in this book.

The initial pane on the left of the RStudio window contains *Console* and *Terminal* tabs, which provide you with direct access respectively to the R interpreter and to your computer's operating system. We will have no occasion in this book to interact directly with the operating system, and if you wish, you can close the *Terminal* tab. The *Source* pane, which holds RStudio's built-in programming editor, will appear on the left when it is needed. The *Source* pane contains tabs for the file or files on which you're working.

**Figure 3** The RStudio window at startup under macOS.



The RStudio interface is customizable, and as you become familiar with it, you may well decide to reconfigure RStudio to your preferences, for example, by moving tabs to different panes and changing the placement of the panes. There are two changes to the standard RStudio configuration that we suggest you make immediately: to prevent RStudio from saving the R workspace when you exit and to prevent RStudio from loading a saved R workspace when it starts up.

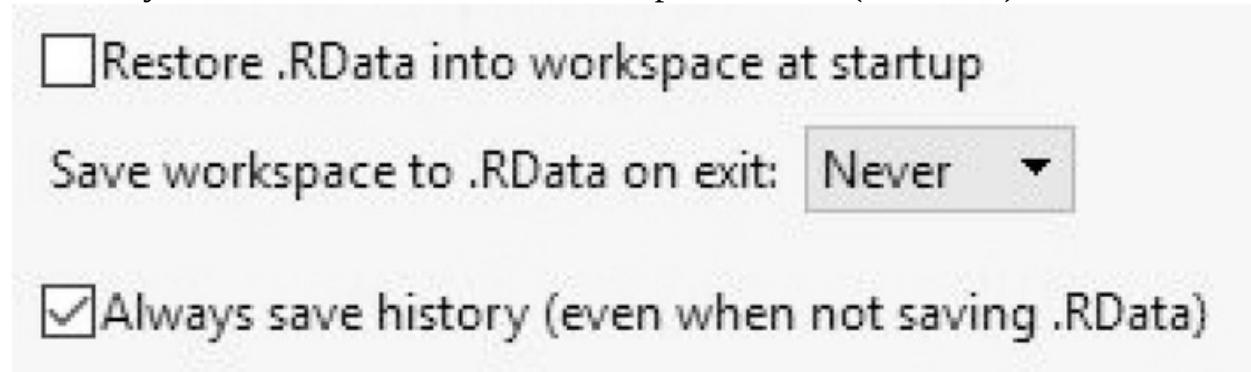
We'll elaborate this point in [Chapter 1](#), when we discuss workflow in R and RStudio.

Select *Tools > Global Options* from the RStudio menus. In the resulting dialog box, as illustrated in the snippet from this dialog in [Figure 4](#), uncheck the box for *Restore .Rdata into workspace at startup*, and select *Never* from the *Save workspace to .RData on exit* drop-down menu. Leave the box for *Always save history (even when not saving .Rdata)* checked.

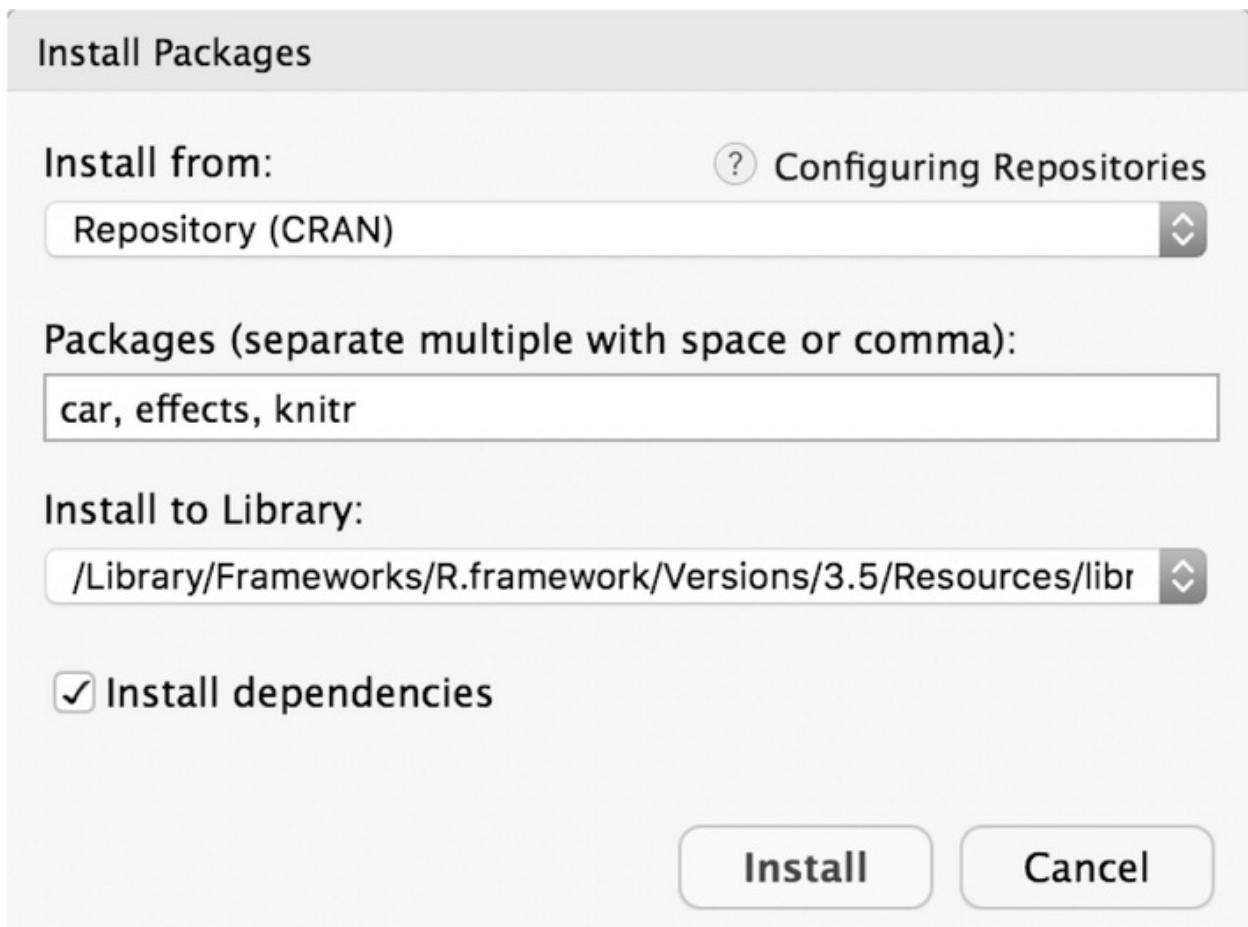
## Installing and Using R Packages

Most of the examples in this book require the **car** and **carData** packages, and many require the **effects** package. To write R Markdown documents, as described in [Chapter 1](#), you'll need the **knitr** package. None of these packages are part of the standard R distribution, but all are easily obtained from CRAN. Some of our examples require still other CRAN packages, such as the **dplyr** package for manipulating data sets, that you may want to install later. You must be connected to the internet to download packages.

**Figure 4** A portion of the *General* screen from the RStudio *Options* dialog, produced by selecting *Tools > Global Options* from the RStudio menus. We recommend that you uncheck the box for restoring the R workspace at startup, and that you elect never to save the workspace on exit (as shown).



**Figure 5** Installing packages using the *Packages* tab from the default download site to the default R library location.



You can see the names of the packages that are part of the standard R distribution by clicking on the RStudio *Packages* tab, and you can install additional packages by clicking on the *Install* button near the top of the *Packages* tab. RStudio will select default values for the *Install from* and *Install to* drop-down lists; you only need to fill in names of the packages to be installed, as shown in [Figure 5](#) for a macOS system, and then click the *Install* button. An alternative to using the RStudio *Install Packages* dialog is to type an equivalent command at the R > command prompt in the *Console* pane:

> *install.packages(c("car", "effects", "knitr"), dependencies=TRUE)*

As with all command-line input, you must hit the Enter or return key when you have finished typing the command. The argument *dependencies=TRUE*, equivalent to the checked *Install dependencies* box in [Figure 5](#), instructs R also to download and install all the packages on which these packages depend, including the **carData** and **rmarkdown** packages.<sup>12</sup>

12 As it turns out, there are several classes of package dependencies, and those in the highest class would be installed along with the directly named packages in any event—including the **carData** and **rmarkdown** packages in the current instance. Even though it can result in downloading and installing more packages

than you really need, specifying dependencies=TRUE is the safest course. We recommend that you regularly update the packages in your package library, either by typing the command update.packages (ask=FALSE) in the *Console* pane or by clicking the *Update* button in the RStudio *Packages* tab.

Installing a package does not make it available for use in a particular R session. When R starts up, it automatically loads a set of standard packages that are part of the R distribution. To access the programs and data in another package, you normally first load the package using the library () command:<sup>13</sup>

13 The name of the library () command is an endless source of largely benign confusion among new users of R. The command loads a *package*, such as the **car** package, which in turn resides in a *library* of packages, much as a book resides in a traditional library. If you want to be among the R cognoscenti, never call a package a “library”!

```
> library ("car")
```

Loading required package: carData

This command also loads the **carData** package, on which the **car** package depends.<sup>14</sup> If you want to use other packages installed in your package library, you need to enter a separate library () command for each. The process of loading packages as you need them will come naturally as you grow more familiar with R.

14 The command library (car) (without the quotes) also works, but we generally prefer enclosing names in quotes—some R commands require quoted names, and so always using quotes avoids having to remember which commands permit unquoted names and which do not.

## Optional: Customizing R

If there are R commands that you wish to execute at the start of each session, you can place these commands in a plain-text file named .Rprofile in your home directory. In Windows, R considers your Documents directory to be your “home directory.” Just create a new text file in RStudio, type in the commands to be executed at the start of each session, and save the file in the proper location, supplying the file name .Rprofile (don’t forget to type the initial period in the file name).<sup>15</sup>

15 In [Chapter 1](#), we’ll explain how to organize your work as RStudio projects, each of which can have its own .Rprofile file.

For example, if you wish to update all of the R packages in your library to the most recent versions each time you start R, then you can place the following command in your .Rprofile file:

```
update.packages (ask=FALSE)
```

## Optional: Installing LATEX

In [Chapter 1](#), we explain how to use R Markdown to create documents that mix R commands with explanatory text. You'll be able to compile these documents to web pages (i.e., HTML files). If you want to be able to compile R Markdown documents to PDF files, you must install LATEX on your computer. LATEX is free, open-source software for typesetting technical documents; for example, this book was written in LATEX. Using LATEX to turn R Markdown documents into PDFs is entirely transparent: RStudio does everything automatically, so you do not need to know how to write LATEX documents.

You can get the MiKTeX LATEX software for Windows systems at <https://miktex.org/download>; select the 64-bit *Basic MiKTeX Installer*. Similarly, you can get MacTeX software for macOS at <http://www.tug.org/mactex>; select *MacTeX Download*. In both cases, installing LATEX is entirely straightforward, and you can take all of the defaults. Be warned that the LATEX installers for both Windows and macOS are well over a gigabyte, and the download may therefore take a long time.

## Using This Book

As its name implies, this book is intended primarily as a *companion* for use with another textbook or textbooks that cover linear models, generalized linear models, and mixed-effects models. For details on the statistical methods, particularly in [Chapters 3 to 8](#), you will need to consult your regression texts. To help you with this task, we provide sections of complementary readings at the end of most chapters, including references to relevant material in Fox (2016) and Weisberg (2014).

While the *R Companion* is not intended as a comprehensive users' manual for R, we anticipate that most students learning regression methods and researchers already familiar with regression but interested in learning to use R will find this book sufficiently thorough for their needs. In addition, a set of manuals in PDF and HTML formats is distributed with R and can be accessed through the *Home* button on the RStudio *Help* tab. These manuals are also available on the R website.<sup>16</sup>

16 R has a substantial user community, which contributes to active and helpful email lists and provides answers to questions about R posted on the StackOverflow@StackOverflow website at

<http://stackoverflow.com/questions/tagged/r/questions/tagged/r>. Links to these resources appear on the R Project website at <https://www.r-project.org/help.html> and in the home screen of the RStudio *Help* tab.

In posing questions on the R email lists and StackOverflow, please try to observe

proper netiquette: Look for answers in the documentation, in frequently-asked-questions (FAQ) lists, and in the email-list and StackOverflow searchable archives before posting a question. Remember that the people who answer your question are volunteering their time. Also, check the posting guide, at [www.r-project.org/posting-guide.html](http://www.r-project.org/posting-guide.html), before posting a question to one of the R email lists.

Various features of R are introduced as they are needed, primarily in the context of detailed, worked-through examples. If you want to locate information about a particular feature, consult the index of functions and operators, or the subject index, at the end of the book; there is also an index of the data sets used in the text.

Occasionally, more demanding material (e.g., requiring a knowledge of matrix algebra or calculus) is marked with an asterisk; this material may be skipped without loss of continuity, as may the footnotes.<sup>17</sup>

17 Footnotes include references to supplementary material (e.g., cross-references to other parts of the text), elaboration of points in the text, and indications of portions of the text that represent (we hope) innocent distortion for the purpose of simplification. The object is to present more complete and correct information without interrupting the flow of the text and without making the main text overly difficult.

Data analysis is a participation sport, and you should try out the examples in the text. Please install R, RStudio, and the **car**, **carData**, and **effects** packages associated with this book before you start to work through the book. As you duplicate the examples in the text, feel free to innovate, experimenting with R commands that do not appear in the examples. Examples are often revisited within a chapter, and so later examples can depend on earlier ones in the same chapter; packages used in a chapter are loaded only once. The examples in *different* chapters are independent of each other. Think of the R code in each chapter as pertaining to a separate R session. To facilitate this process of replication and experimentation, we provide a script of R commands used in each chapter on the website for the book.

## Chapter Synopses

[Chapter 1](#) explains how to interact with the R interpreter and the RStudio interactive development environment, shows you how to organize your work for reproducible research, introduces basic concepts, and provides a variety of examples, including an extended illustration of the use of R in data analysis. The chapter includes a brief presentation of R functions for basic statistical methods.

[Chapter 2](#) shows you how to read data into R from several sources and how to work with data sets. There are also discussions of basic data structures, such as vectors, matrices, arrays, and lists, along with information on handling time, date, and character data, on dealing with large data sets in R, and on the general representation of data in R.

[Chapter 3](#) discusses the exploratory examination and transformation of data, with an emphasis on graphical displays.

[Chapter 4](#) describes the use of R functions for fitting, manipulating, and displaying linear models, including simple- and multiple-regression models, along with more complex linear models with categorical predictors (factors), polynomial regressors, regression splines, and interactions.

[Chapter 5](#) focuses on standard errors, confidence intervals, and hypothesis tests for coefficients in linear models, including applications of robust standard errors, the delta method for nonlinear functions of coefficients, and the bootstrap.

[Chapter 6](#) largely parallels the material developed in [Chapters 4](#) and [5](#), but for generalized linear models (GLMs) in R. Particular attention is paid to GLMs for categorical data and to Poisson and related GLMs for counts.

[Chapter 7](#) develops linear and generalized linear mixed-effects models in R for clustered observations, with applications to hierarchical and longitudinal data.

[Chapter 8](#) describes methods—often called “regression diagnostics”—for determining whether linear models, GLMs, and mixed-effects models adequately describe the data to which they are fit. Many of these methods are implemented in the `car` package associated with this book.

[Chapter 9](#) focuses on customizing statistical graphs in R, describing a step-by-step approach to constructing complex R graphs and diagrams. The chapter also introduces some widely used R graphics packages.

[Chapter 10](#) is a general introduction to basic programming in R, including discussions of function definition, operators and functions for handling matrices, control structures, optimization problems, debugging R programs, Monte-Carlo simulation, object-oriented programming, and writing statistical-modeling functions.

## Typographical Conventions

- Input and output are printed in bold-face slanted and normal-weight upright monospaced (typewriter) fonts, respectively, slightly indented from the left margin—for example, The `>` prompt at the beginning of the input and the `+` prompt, which begins continuation lines when an R command extends over

more than one line (as illustrated in the second example above), are provided by R, not typed by the user. In the remainder of the book we suppress the command and continuation prompts when we show R input.

```
> mean(1:10) # an input line
```

```
[1] 5.5
```

```
> S(model.duncan <- lm(prestige ~ income + education,  
+ data=Duncan)) # command broken across two lines
```

```
Call: lm(formula = prestige ~ income + education, data =  
Duncan)
```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )		
(Intercept)	-6.0647	4.2719	-1.42	0.16		
income	0.5987	0.1197	5.00	1.1e-05 ***		
education	0.5458	0.0983	5.56	1.7e-06 ***		
---						
Signif. codes:	0 '***'	0.001 '**'	0.01 '*'	0.05 '.'	0.1 ' '	1

Residual standard deviation: 13.4 on 42 degrees of freedom

Multiple R-squared: 0.828

F-statistic: 101 on 2 and 42 DF, p-value: <2e-16

AIC	BIC
365.96	373.19

- R input and output are printed as they appear on the computer screen, although we sometimes edit output for brevity or clarity; elided material in computer output is indicated by three widely spaced periods (...).
- Data set names, variable names, the names of R functions and operators, and R expressions that appear in the body of the text are in a monospaced (typewriter) font: Duncan, income, mean (), +, lm (prestige ~ income + education, data=Prestige).
- Names of R functions that appear in the text are followed by parentheses to differentiate them from other kinds of objects, such as R variables: lm () and mean () are functions, and model.duncan and mean are variables.
- The names of R packages are in boldface: **car**.
- Occasionally, generic specifications (to be replaced by particular information, such as a variable name) are given in typewriter italics: mean (*variable-name*).
- Graphical-user-interface elements, such as menus, menu items, and the

names of windows, panes, and tabs, are set in an italic sans-serif font: *File*, *Exit*, *R Console*.

- We use a sans-serif font for other names, such as names of operating systems, programming languages, software packages, files, and directories (folders): Windows, R, SAS, Duncan.txt, c:\Program Files\R\R-3.5.0\etc.
- Graphical output from R is shown in many figures scattered through the text; in normal use, graphs appear on the computer screen in the RStudio *Plots* tab, from which they can be saved in several different formats (see the *Export* drop-down menu button at the top of the *Plots* tab).

## New in the Third Edition

We have thoroughly reorganized and rewritten the *R Companion* and have added a variety of new material, most notably a new chapter on mixed-effects models. The *R Companion* serves partly as extended documentation for programs in the **car** and **effects** packages, and we have substantially updated both packages for this new edition of the book, introducing additional capabilities and making the software more consistent and easier to use.

We have de-emphasized statistical programming in the *R Companion* while retaining a general introduction to basic R programming. We think that most R users will benefit from acquiring fundamental programming skills, which, for example, will help reduce their dependence on preprogrammed solutions to what are often idiosyncratic data manipulation problems. We plan a separate volume (a “companion to the *Companion*”) that will take up R programming in much greater detail.

This new edition also provides us with the opportunity to advocate an everyday data analysis workflow that encourages reproducible research. To this end, we suggest that you use RStudio, an interactive development environment for R that allows you to organize and document your work in a simple and intuitive fashion, and then easily to share your results with others. The documents you produce in RStudio typically include R commands, printed output, and statistical graphs, along with free-form explanatory text, permitting you—or anyone else—to understand, reproduce, and extend your work. This process serves to reduce errors, improve efficiency, and promote transparency.

## The Website for the *R Companion*

We maintain a website for the *R Companion* at <https://socialsciences.mcmaster.ca/jfox/Books/Companion/index.html> or, more conveniently, at [tinyurl.com/rcompanion](http://tinyurl.com/rcompanion).<sup>18</sup> The website for the book includes the following materials:

18 If you have difficulty accessing this website, please check the Sage

Publications website at [www.sagepub.com](http://www.sagepub.com) for up-to-date information. Search for “John Fox,” and follow the links to the website for the book.

- Appendices, referred to as “online appendices” in the text, containing brief information on using R for various extensions of regression analysis not considered in the main body of the book, along with some other topics. We have relegated this material to downloadable appendices in an effort to keep the text to a reasonable length. We plan to update the appendices from time to time as new developments warrant.

We currently plan to make the following appendices available, not all of which will be ready when the book is published: Bayesian estimation of regression models; fitting regression models to complex surveys; nonlinear regression; robust and resistant regression; nonparametric regression; time-series regression; Cox regression for survival data; multivariate linear models, including repeated-measures analysis of variance; and multiple imputation of missing data.

- Downloadable scripts of R commands for all the examples in the text
- An R Markdown document for the Duncan example in [Chapter 1](#)
- A few data files employed in the book, exclusive of the data sets in the **carData** package
- Errata and updated information about R

All these resources can be accessed using the `carWeb()` function in the **car** package: After loading the **car** package via the command `library("car")` entered at the R > prompt, enter the command `help("carWeb")` for details.

## Beyond the *R Companion*

There is, of course, much to R beyond the material in this book. The S language is documented in several books by John Chambers and his colleagues: *The New S Language: A Programming Environment for Data Analysis and Graphics* (Becker, Chambers, & Wilks, 1988) and an edited volume, *Statistical Models in S* (Chambers & Hastie, 1992b), describe what came to be known as S3, including the S3 object-oriented programming system, and facilities for specifying and fitting statistical models. Similarly, *Programming With Data* (Chambers, 1998) describes the S4 language and object system, and the system of “reference classes.” The R dialect of S incorporates both S3 and S4, along with reference classes, and so these books remain valuable sources.

Beyond these basic references, there are now so many books that describe the application of R to various areas of statistics that it is impractical to compile a list here, a list that would inevitably be out-of-date by the time this book goes to press. We include complementary readings at the end of most chapters.

## Acknowledgments

We are grateful to a number of individuals who provided valuable assistance in writing this book and its predecessors:

- Several people have made contributions to the **car** package that accompanies the book; they are acknowledged in the package itself—see `help(package="car")`.
- Michael Friendly and three unusually diligent (and, at the time, anonymous) reviewers, Jeff Gill, J. Scott Long, and Bill Jacoby (who also commented on a draft of the second edition), made many excellent suggestions for revising the first edition of the book, as did eight anonymous reviewers of the second edition and 14 anonymous reviewers of the third edition.
- Three editors at SAGE, responsible respectively for the first, second, and third editions of the book, were invariably helpful and supportive: C. Deborah Laughton, Vicki Knight, and Helen Salmon.
- The second edition of the book was written in LATEX using live R code compiled with the wonderful Sweave document preparation system. We are grateful to Fritz Leisch (Leisch, 2002) for Sweave. For the third edition, we moved to the similar but somewhat more powerful **knitr** package for R, and we are similarly grateful to Yihui Xie (Xie, 2015, 2018) for developing **knitr**.
- Finally, we wish to express our gratitude to the developers of R and RStudio, and to those who have contributed the R software used in the book, for the wonderful resource that they have created with their collaborative and, in many instances, selfless efforts.

# About the Authors

## **John Fox**

is Professor Emeritus of Sociology at McMaster University in Hamilton, Ontario, Canada, where he was previously the Senator William McMaster Professor of Social Statistics. Prior to coming to McMaster, he was Professor of Sociology and Coordinator of the Statistical Consulting Service at York University in Toronto. Professor Fox is the author of many articles and books on applied statistics, including *Applied Regression Analysis* and *Generalized Linear Models*, Third Edition (Sage, 2016). He is an elected member of the R Foundation, an associate editor of the *Journal of Statistical Software*, a prior editor of R News and its successor the R Journal, and a prior editor of the Sage Quantitative Applications in the Social Sciences monograph series.

## **Sanford Weisberg**

is Professor Emeritus of statistics at the University of Minnesota. He has also served as the director of the University's Statistical Consulting Service, and has worked with hundreds of social scientists and others on the statistical aspects of their research. He earned a BA in statistics from the University of California, Berkeley, and a PhD, also in statistics, from Harvard University, under the direction of Frederick Mosteller. The author of more than 60 articles in a variety of areas, his methodology research has primarily been in regression analysis, including graphical methods, diagnostics, and computing. He is a fellow of the American Statistical Association and former Chair of its Statistical Computing Section. He is the author or coauthor of several books and monographs, including the widely used textbook *Applied Linear Regression*, which has been in print for almost 40 years.

# 1 Getting Started With R and RStudio

John Fox & Sanford Weisberg

This chapter shows you how to conduct statistical data analysis using R as the primary data analysis tool and RStudio as the primary tool for organizing your data analysis workflow and for producing reports. The goal is to conduct *reproducible research*, in which the finished product not only summarizes your findings but also contains all of the instructions and data needed to replicate your work. Consequently, you, or another skilled person, can easily understand what you have done and, if necessary, reproduce it. If you are a student using R for assignments, you will simultaneously analyze data using R, make your R commands available to your instructor, and write up your findings using R Markdown in RStudio. If you are a researcher or a data analyst, you can follow the same general process, preparing documents that contain reproducible results ready for distribution to others.

- We begin the chapter in [Section 1.1](#) with a discussion of RStudio *projects*, which are a simple, yet powerful, means of organizing your work that takes advantage of the design of RStudio.
- We then introduce a variety of basic features of R in [Section 1.2](#), showing you how to use the *Console* in RStudio to interact directly with the R interpreter; how to call R functions to perform computations; how to work with vectors, which are one-dimensional arrays of numbers, character strings, or logical values; how to define variables; and how to define functions.
- In [Section 1.3](#), we explain how to locate and fix errors and how to get help with R.
- Direct interaction with R is occasionally useful, but in [Section 1.4](#) we show you how to work more effectively by using the RStudio editor to create scripts of R commands, making it easier to correct mistakes and to save a permanent record of your work. We then outline a more sophisticated and powerful approach to reproducible research, combining R commands with largely free-form explanatory material in an R Markdown document, which you can easily convert into a neatly typeset report.
- Although the focus of the *R Companion* is on using R for regression modeling, [Section 1.6](#) enumerates some generally useful R functions for basic statistical methods.
- Finally, [Section 1.7](#) explains how so-called *generic functions* in R are able to adapt their behavior to different kinds of data, so that, for example, the `summary()` function produces very different reports for a data set and for a

linear regression model.

By the end of the chapter, you should be able to start working efficiently in R and RStudio.

We know that many readers are in the habit of beginning a book at [Chapter 1](#), skipping the Preface. The Preface to this book, however, includes information about installing R and RStudio on your computer, along with the **car**, **effects**, and **car-Data** packages, which are associated with the *R Companion to Applied Regression* and are necessary for many of the examples in the text. We suggest that you rework the examples as you go along, because data analysis is best learned by doing, not simply by reading. Moreover, the Preface includes information on the typographical and other conventions that we use in the text. So, if you haven't yet read the Preface, please back up and do so now!

## 1.1 Projects in RStudio

Projects are a means of organizing your work so that both you and RStudio can keep track of all the files relevant to a particular activity. For example, a student may want to use R in several different projects, such as a statistics course, a sociology course, and a thesis research project. The data and data analysis files for each of these activities typically differ from the files associated with the other activities. Similarly, researchers and data analysts generally engage in several distinct research projects, both simultaneously and over time. Using RStudio projects to organize your work will keep the files for distinct activities separate from each other.

If you return to a project after some time has elapsed, you can therefore be sure that all the files in the project directory are relevant to the work you want to reproduce or continue, and if you have the discipline to put all files that concern a project in the project directory, everything you need should be easy to find. By organizing your work in separate project directories, you are on your way to conducting fully reproducible research.

Your first task is to create a new project directory. You can keep a project directory on your hard disk or on a flash drive. Some cloud services, like Dropbox, can also be used for saving a project directory.<sup>1</sup> To create a project, select *File > New Project* from the RStudio menus. In the resulting sequence of dialog boxes, successively select *Existing Directory* or *New Directory*, depending on whether or not the project directory already exists; navigate by pressing the *Browse...* button to the location where you wish to create the project; and, for a new directory, supply the name of the directory to be created. We assume here that you enter the name R-Companion for the project directory, which will contain files relevant to this book. You can use the R-Companion

project as you work through this and other chapters of the *R Companion*.

[1](#) At present, Google Drive appears to be incompatible with RStudio projects. Creating the R-Companion project changes the RStudio window, which was depicted in its original state in Figures 2 and 3 in the Preface (pages xix and xx), in two ways: First, the name of the project you created is shown in the upper-right corner of the window. Clicking on the project name displays a drop-down menu that allows you to navigate to other projects you have created or to create a new project. The second change is in the *Files* tab, located in the lower-right pane, which lists the files in your project directory. If you just created the R-Companion project in a new directory, you will only see one file, *R-Companion.Rproj*, which RStudio uses to administer the project.

Although we don't need them quite yet, we will add a few files to the R-Companion project, typical of the kinds of files you might find in a project directory. Assuming that you are connected to the internet and have previously installed the **car** package, type the following two commands at the > command prompt in the *Console* pane,[2](#) remembering to press the Enter or return key after each command:

```
library ("car")
```

Loading required package: carData

```
carWeb (setup=TRUE)
```

The first of these commands loads the **car** package.[3](#) When **car** is loaded, the **car-Data** package is automatically loaded as well, as reflected in the message printed by the library () command. In subsequent chapters, we suppress package-startup messages to conserve space. The second command calls a function called **carWeb ()** in the **car** package to initialize your project directory by downloading several files from the website for this book to the R-Companion project directory. As shown in [Figure 1.1](#), information about the downloaded files is displayed in the *Console*, and the files are now listed in the *Files* tab. We will use these files in this chapter and in a few succeeding chapters.

**Figure 1.1** RStudio window for the R-companion project after adding the files used in this and subsequent chapters.

The screenshot shows the RStudio interface with the following details:

- Console pane:** Displays R code and its output. It shows multiple URL downloads from <https://socialsciences.mcmaster.ca/jfox/Books/Companion/setup/>, including files like chap-4.R, chap-5.R, chap-6.R, chap-7.R, chap-8.R, chap-9.R, and chap-10.R.
- Environment pane:** Shows the Global Environment, which is currently empty.
- Files pane:** Shows the contents of the R-Companion project directory. The files listed are:
 

Name	Size	Modified
R-Companion.Rproj	218 B	Apr 25, 20
Duncan.txt	1.1 KB	Apr 25, 20
Duncan.csv	1.1 KB	Apr 25, 20
Duncan.xlsx	10.3 KB	Apr 25, 20
Duncan.Rmd	2.8 KB	Apr 25, 20
Hamlet.txt	1.5 KB	Apr 25, 20
RMardownTest.Rmd	923 B	Apr 25, 20
zipmod.R	2.1 KB	Apr 25, 20
zipmodBugged.R	1.3 KB	Apr 25, 20
zipmod-generic.R	6.1 KB	Apr 25, 20
chap-1.R	4 KB	Apr 25, 20
chap-2.R	12.5 KB	Apr 25, 20

**2** As we explained in the Preface, we don't show the command prompt when we display R input in the text.

**3** If you haven't installed the **car** package, or if the library ("car") command produces an error, then please (re)read the Preface to get going.

Files in an RStudio project typically include plain-text data files, usually of file type .txt or .csv, files of R commands, of file type .R or .r, and R Markdown files, of file type .Rmd, among others. In a complex project, you may wish to create subdirectories of the main project directory, for example, a Data subdirectory for data files or a Reports subdirectory for R Markdown files.

RStudio starts in a "default" working directory whenever you choose not to use a project. You can reset the default directory, which is initially set to your Documents directory in Windows or your home directory in macOS, by selecting *Tools > Global Options* from the RStudio menus, and then clicking the *Browse...* button on the *General* tab, navigating to the desired location in your file system. We find that we almost never work in the default directory, however, because creating a new RStudio project for each problem is very easy and because putting unrelated files in the same directory is a recipe for confusion.

## 1.2 R Basics

### 1.2.1 Interacting With R Through the Console

Data analysis in R typically proceeds as an interactive dialogue with the interpreter, which may be accessed directly in the RStudio *Console* pane. We can type an R command at the *> prompt* in the *Console* (which is not shown in the R input displayed below) and press the Enter or return key. The interpreter responds by executing the command and, as appropriate, returning a result, producing graphical output, or sending output to a file or device.

The R language includes the usual arithmetic operators:

- + addition
- subtraction
- \* multiplication
- / division
- ^ or \*\* exponentiation

Here are some simple examples of arithmetic in R:

```
2 + 3 # addition  
[1] 5  
2 - 3 # subtraction  
[1] -1  
2*3 # multiplication  
[1] 6  
2/3 # division  
[1] 0.666667  
2^3 # exponentiation  
[1] 8
```

Output lines are preceded by [1]. When the printed output consists of many values (a “vector”: see [Section 1.2.4](#)) spread over several lines, each line begins with the index number of the first element in that line; an example will appear shortly. After the interpreter executes a command and returns a value, it waits for the next command, as indicated by the > prompt. The pound sign or hash mark (#) signifies a *comment*, and text to the right of # is ignored by the interpreter. We often take advantage of this feature to insert explanatory text to the right of commands, as in the examples above.

Several arithmetic operations may be combined to build up complex expressions:

```
4^2 - 3*2  
[1] 10
```

In the usual mathematical notation, this command is  $4^2 - 3 \times 2$ . R uses standard conventions for precedence of mathematical operators. So, for example,

exponentiation takes place before multiplication, which takes place before subtraction. If two operations have equal precedence, such as addition and subtraction, then they are evaluated from left to right:

```
1 - 6 + 4
```

```
[1] -1
```

You can always explicitly specify the order of evaluation of an expression by using parentheses; thus, the expression  $4^2 - 3^2$  is equivalent to

```
(4^2) - (3^2)
```

```
[1] 10
```

and

```
(4 + 3)^2
```

```
[1] 49
```

is different from

```
4 + 3^2
```

```
[1] 13
```

Although spaces are not required to separate the elements of an arithmetic expression, judicious use of spaces can help clarify the meaning of the expression. Compare the following commands, for example:

```
-2--3
```

```
[1] 1
```

```
-2 - -3
```

```
[1] 1
```

Placing spaces around operators usually makes expressions more readable, as in the preceding examples, and some style standards for R code suggest that they *always* be used. We feel, however, that readability of commands is generally improved by putting spaces around the binary arithmetic operators + and - but not usually around \*, /, or ^.

Interacting directly with the R interpreter by typing at the command prompt is, for a variety of reasons, a poor way to organize your work. In [Section 1.4](#), we'll show you how to work more effectively using scripts of R commands and, even better, dynamic R Markdown documents.

## 1.2.2 Editing R Commands in the Console

- The arrow keys on your keyboard are useful for navigating among commands previously entered into the *Console*: Use the ↑ and ↓ keys to move up and down in the the command history.
- You can also access the command history in the RStudio *History* tab: Double-clicking on a command in the *History* tab transfers the command to the > prompt in the *Console*.

- The command shown after the > prompt can either be re-executed or edited. Use the → and ← keys to move the cursor within the command after the >. Use the backspace or delete key to erase a character. Typed characters will be inserted at the cursor.
- You can also move the cursor with the mouse, left-clicking at the desired point.

### 1.2.3 R Functions

In addition to the common arithmetic operators, the packages in the standard R distribution include hundreds of functions (programs) for mathematical operations, for manipulating data, for statistical data analysis, for making graphs, for working with files, and for other purposes. Function *arguments* are values passed to functions, and these are specified within parentheses after the function name. For example, to calculate the natural log of 100, that is,  $\log_e(100)$  or  $\ln(100)$ , we use the `log()` function:<sup>4</sup>

```
log (100)
```

```
[1] 4.6052
```

[4](#) Here's a quick review of logarithms ("logs"), which play an important role in statistical data analysis (see, e.g., [Section 3.4.1](#)):

- The log of a positive number  $x$  to the base  $b$  (where  $b$  is also a positive number), written  $\log_b x$ , is the exponent to which  $b$  must be raised to produce  $x$ . That is, if  $y = \log_b x$ , then  $b^y = x$ .
- Thus, for example,  $\log_{10} 100 = 2$  because  $10^2 = 100$ ;  $\log_{10} 0.01 = -2$  because  $10^{-2} = 1/10^2 = 0.01$ ;  $\log_2 8 = 3$  because  $2^3 = 8$ ; and  $\log_2 1/8 = -3$  because  $2^{-3} = 1/8$ .
- So-called *natural logs* use the base  $e \approx 2.71828$ .
- Thus, for example,  $\log_e e = 1$  because  $e^1 = e$ .
- Logs to the bases 2 and 10 are often used in data analysis, because powers of 2 and 10 are familiar. Logs to the base 10 are called "common logs."
- Regardless of the base  $b$ ,  $\log_b 1 = 0$ , because  $b^0 = 1$ .
- Regardless of the base, the log of zero is undefined (or taken as  $\log 0 = -\infty$ ), and the logs of negative numbers are undefined.

To compute the log of 100 to the base 10, we specify

```
log (100, base=10)
```

```
[1] 2
```

We could equivalently use the specialized `log10()` function:

```
log10(100) # equivalent
```

```
[1] 2
```

Arguments to R functions may be specified in the order in which they occur in the function definition, or by the name of the argument followed by = (the equals sign) and a value. In the command `log(100, base=10)`, the value 100 is implicitly matched to the first argument of the `log` function. The second argument in the function call, `base=10`, explicitly matches the value 10 to the argument `base`. Arguments specified by name need not appear in a function call in the same order that they appear in the function definition.

Different arguments are separated by commas, and, for clarity, we prefer to leave a space after each comma, although these spaces are not required. Some stylistic standards for R code recommend placing spaces around = in assigning values to arguments, but we usually find it clearer not to insert extra spaces here.

Function-argument names may be abbreviated, as long as the abbreviation is unique; thus, the previous example may be rendered more compactly as

```
log(100, b=10)
```

```
[1] 2
```

because the `log()` function has only one argument beginning with the letter “b.” We generally prefer *not* to abbreviate function arguments because abbreviation promotes unclarity.

This example begs a question, however: How do we know what the arguments to the `log()` function are? To obtain information about a function, use the `help()` function or, synonymously, the `? help` operator. For example,

```
help("log")
```

```
?log
```

Either of these equivalent commands opens the R *help page* for the `log()` function and some closely associated functions, such as the exponential function,  $\exp(x) = e^x$ , in the RStudio *Help* tab. [Figure 1.2](#) shows the resulting help page in abbreviated form, where three widely separated dots (...) mean that we have elided some information, a convention that we’ll occasionally use to abbreviate R output as well.

**Figure 1.2** Abbreviated documentation displayed by the command `help("log")`. The ellipses (...) represent elided lines, and the underscored text under “See Also” indicates hyperlinks to other help pages—click on a link to go to the corresponding help page. The symbol -Inf in the “Value” section represents minus infinity ( $-\infty$ ), and NaN means “not a number.”

`log {base}`

R Documentation

## Logarithms and Exponentials

### Description

`log()` computes logarithms, by default natural logarithms, `log10()` computes common (i.e., base 10) logarithms, and `log2()` computes binary (i.e., base 2) logarithms. The general form `log(x, base)` computes logarithms with base `base`.

...

`exp()` computes the exponential function.

...

### Usage

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)
. . .
exp(x)
. . .
```

### Arguments

`x`: a numeric or complex vector.

`base`: a positive or complex number: the base with respect to which logarithms are computed. Defaults to  $e = \exp(1)$ .

### Details

...

### Value

A vector of the same length as `x` containing the transformed values. `log(0)` gives `-Inf`, and negative values give `NaN`.

...

### See Also

[Trig](#), [sqrt\(\)](#), [Arithmetic](#).

### Examples

```
log(exp(3))
log10(1e7) # = 7
. . .
```

The `log()` help page is more or less typical of help pages for R functions in both standard R packages and in contributed packages obtained from CRAN. The *Description* section of the help page provides a brief, general description of the documented functions; the *Usage* section shows each documented function, its arguments, and argument default values (for arguments that have defaults, see below); the *Arguments* section explains each argument; the *Details* section (suppressed in [Figure 1.2](#)) elaborates the description of the documented functions; the *Value* section describes the value returned by each documented function; the *See Also* section includes hyperlinked references to other help pages; and the *Examples* section illustrates the use of the documented functions. There may be other sections as well; for example, help pages for functions documented in contributed CRAN packages typically have an *Author* section. A novel feature of the R help system is the facility it provides to execute most examples in the help pages via the `example()` command:

```
example("log")
log> log(exp(3))
[1] 3
log> log10(1e7) # = 7
[1] 7
...

```

The number  $1\text{e}7$  in the second example is given in *scientific notation* and represents  $1 \times 10^7 = 10$  million. Scientific notation may also be used in R output to represent very large or very small numbers.

A quick way to determine the arguments of an R function is to use the `args()` function:<sup>5</sup>

```
args("log")
function(x, base = exp(1))
NULL
```

[5](#) Disregard the `NULL` value returned by `args()`.

Because `base` is the second argument of the `log()` function, to compute  $\log_{10} 100$ , we can also type

```
log(100, 10)
[1] 2
```

specifying both arguments to the function (i.e., `x` and `base`) by position.

An argument to a function may have a *default* value that is used if the argument is not explicitly specified in a function call. Defaults are shown in the function documentation and in the output of `args()`. For example, the `base` argument to the `log()` function defaults to `exp(1)` or  $e^1 \approx 2.71828$ , the base of the natural

logarithms.

## 1.2.4 Vectors and Variables

R would not be very convenient to use if we always had to compute one value at a time. The arithmetic operators, and most R functions, can operate on more complex data structures than individual numbers. The simplest of these data structures is a numeric vector, or one-dimensional array of numbers.<sup>6</sup> An individual number in R is really a vector with a single element.

<sup>6</sup> We refer here to vectors informally as one-dimensional “arrays” using that term loosely, because *arrays* in R are a distinct data structure (described in [Section 2.4](#)).

A simple way to construct a vector is with the `c()` function, which combines its elements:

**`c(1, 2, 3, 4)`**

[1] 1 2 3 4

Many other functions also return vectors as results. For example, the *sequence operator*: generates consecutive whole numbers, while the *sequence function* `seq()` does much the same thing but more flexibly:

**`1:4 # integer sequence`**

[1] 1 2 3 4

**`4:1 # descending`**

[1] 4 3 2 1

**`-1:2 # negative to positive`**

[1] -1 0 1 2

**`seq(1, 4) # equivalent to 1:4`**

[1] 1 2 3 4

**`seq(2, 8, by=2) # specify interval between elements`**

[1] 2 4 6 8

**`seq(0, 1, by=0.1) # noninteger sequence`**

[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

**`seq(0, 1, length=11) # specify number of elements`**

[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

The standard arithmetic operators and functions extend to vectors in a natural manner on an elementwise basis:

**`c(1, 2, 3, 4)/2`**

[1] 0.5 1.0 1.5 2.0

**`c(1, 2, 3, 4)/c(4, 3, 2, 1)`**

[1] 0.25000 0.66667 1.50000 4.00000

**`log(c(0.1, 1, 10, 100), base=10)`**

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

If the operands are of different lengths, then the shorter of the two is extended by repetition, as in  $c(1, 2, 3, 4)/2$  above, where the 2 in the denominator is effectively repeated four times. If the length of the longer operand is *not* a multiple of the length of the shorter one, then a warning message is printed, but the interpreter proceeds with the operation, *recycling* the elements of the shorter operand:

```
c(1, 2, 3, 4) + c(4, 3) # no warning
```

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

```
c(1, 2, 3, 4) + c(4, 3, 2) # produces warning
```

```
[1] 5 5 5 8
```

Warning message:

In  $c(1, 2, 3, 4) + c(4, 3, 2)$ :

longer object length is not a multiple of shorter object length

R would be of little practical use if we were unable to save the results returned by functions to use them in further computation. A value is saved by *assigning* it to a *variable*, as in the following example, which assigns the vector  $c(1, 2, 3, 4)$  to the variable x:

```
x <- c(1, 2, 3, 4) # assignment x # print
```

```
[1] 1 2 3 4
```

The left-pointing arrow ( $<-$ ) is the *assignment operator*; it is composed of the two characters < (less than) and - (dash or minus), with no intervening blanks, and is usually read as *gets*: “The variable x *gets* the value  $c(1, 2, 3, 4)$ .” The equals sign (=) may also be used for assignment in place of the arrow ( $<-$ ), except inside a function call, where = is used exclusively to specify arguments by name. We generally recommend the use of the arrow for assignment.<sup>7</sup>

<sup>7</sup> R also permits a right-pointing arrow for assignment, as in  $2 + 3 \rightarrow x$ , but its use is uncommon.

As the preceding example illustrates, when the leftmost operation in a command is an assignment, nothing is printed. Typing the name of a variable as in the second command above is equivalent to typing the command `print(x)` and causes the value of x to be printed.

Variable and function names in R are composed of letters (a–z, A–Z), numerals (0–9), periods (.), and underscores (\_), and they may be arbitrarily long. In particular, the symbols # and - should not appear in variable or function names. The first character in a name must be a letter or a period, but variable names beginning with a period are reserved by convention for special purposes.<sup>8</sup> Names in R are case sensitive: So, for example, x and X are distinct variables. Using

descriptive names, for example, totalIncome rather than x2, is almost always a good idea.

[8](#) *Nonstandard names* may also be used in a variety of contexts, including assignments, by enclosing the names in back-ticks, or in single or double quotes (e.g., "given name" <- "John" ). Nonstandard names can lead to unanticipated problems, however, and in almost all circumstances are best avoided.

R commands using variables simply substitute the values of the variables for their names:

```
x/2 # equivalent to c(1, 2, 3, 4)/2  
[1] 0.5 1.0 1.5 2.0  
(y <- sqrt(x))  
[1] 1.0000 1.4142 1.7321 2.0000
```

In the last example, `sqrt()` is the square-root function, and thus `sqrt(x)` is equivalent to  $x^{0.5}$ . To obtain printed output without having to type the name of the variable `y` as a separate command, we enclose the command in parentheses so that the assignment is no longer the leftmost operation. We will use this trick regularly to make our R code more compact.

Variables in R are dynamically defined, meaning that we need not tell the interpreter in advance how many values `x` will hold or whether it contains integers, real numbers, character strings, or something else. Moreover, if we wish, we may freely *overwrite* (i.e., redefine) an existing variable, here, `x`:

```
(x <- rnorm(100)) # 100 standard-normal random numbers  
[1] 0.58552882 0.70946602 -0.10930331 -0.45349717 0.60588746  
[6] -1.81795597 0.63009855 -0.27618411 -0.28415974 -0.91932200  
[11] -0.11624781 1.81731204 0.37062786 0.52021646 -0.75053199  
. . .  
[91] -0.96390148 -0.85508251 1.88694694 -0.39181937 -0.98063295  
[96] 0.68733210 -0.50504352 2.15771982 -0.59979756 -0.69454669
```

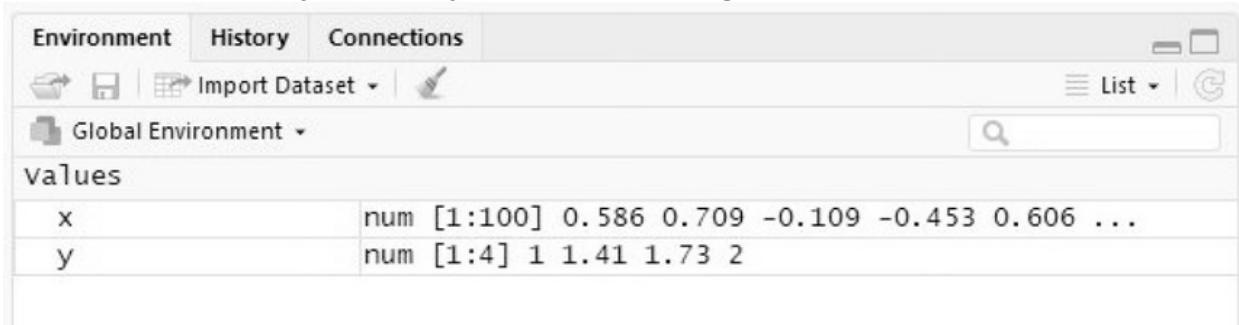
The `rnorm()` function generates standard-normal random numbers,[9](#) in this case, 100 of them. Two additional arguments of `rnorm()`, `mean` and `sd`, which are not used in this example, allow us to sample values from a normal distribution with arbitrary mean and standard deviation; the defaults are `mean=0` and `sd=1`, and because we did not specify these arguments, the defaults were used (for details, see `help("rnorm")`). When a vector prints on more than one line, as in the last example, the index number of the leading element of each line is shown in square brackets; thus, the first value in the second line of output is the sixth element of the vector `x`.

[9](#) Because the values are sampled randomly, when you enter this command

you'll get a different result from ours. Random-number generation in R, including how to make the results of random simulations reproducible, is discussed in [Section 10.7](#).

[Figure 1.3](#) shows the contents of the RStudio *Environment* tab after `x` has been overwritten. The *Environment* tab displays a brief summary of objects defined in the *global environment*, currently the numeric vector `x` with 100 values, the first five of which are shown, and the numeric vector `y`. The global environment, also called the user *workspace*, is the region of your computer's memory that holds objects created at the R command prompt. Overwriting `x` with a new value did not change the value of `y`.

**Figure 1.3** The *Environment* tab shows the name, and an abbreviated version of the value, for all objects that you define in the global environment.

A screenshot of the RStudio interface showing the Environment tab. The tab bar at the top includes Environment, History, and Connections. Below the tabs is a toolbar with icons for file operations like Open, Save, and Import Dataset, and a search bar labeled 'List'. Underneath the toolbar, a dropdown menu shows 'Global Environment'. The main area is titled 'values' and lists two objects: 'x' and 'y'. The 'x' object is described as a numeric vector of length 100, with the first few values being 0.586, 0.709, -0.109, -0.453, and 0.606 followed by ellipses. The 'y' object is described as a numeric vector of length 4, with the first few values being 1, 1.41, 1.73, and 2.

The `summary()` function is an example of a *generic function*: How it behaves depends on its argument. Applied to the numeric vector `x` of 100 numbers sampled from the standard-normal distribution, we get

**`summary(x)`**

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-2.380	-0.590	0.484	0.245	0.900	2.477

In this case, `summary(x)` prints the minimum and maximum values of its argument, along with the mean, median, and first and third quartiles (but, curiously, not the standard deviation). Applied to another kind of object, to a data set or a regression model, for example, `summary()` produces different information.<sup>10</sup>

<sup>10</sup> The `args()` and `help()` functions may not be very helpful with generic functions. See [Section 1.7](#) for an explanation of how generic functions in R work.

## 1.2.5 Nonnumeric Vectors

Vectors may also contain nonnumeric values. For example, the command

```
(words <- c ("To", "be", "or", "not", "to", "be"))
```

```
[1] "To" "be" "or" "not" "to" "be"
```

defines a *character vector* whose elements are *character strings*. Many R functions work with character data. For example, we may call `paste()` to turn the vector words into a single character string:

```
paste (words, collapse= " ")
```

```
[1] "To be or not to be"
```

The very useful `paste()` function pastes character strings together; the `collapse` argument, as its name implies, collapses the character vector into a single string, separating the elements by the character or characters between the quotation marks, in this case one blank space.<sup>11</sup>

<sup>11</sup> The `paste()` function is discussed along with other functions for manipulating character data in [Section 2.6](#).

A *logical vector* consists of elements that are either TRUE or FALSE:

```
(logical.values <- c (TRUE, TRUE, FALSE, TRUE))
```

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

The symbols T and F may also be used as logical values, but while TRUE and FALSE are *reserved symbols* in R,<sup>12</sup> T and F are not, an omission that we regard as a design flaw in the language.<sup>13</sup> For example, you can perniciously assign `T <- FALSE` and `F <- TRUE` (*Socrates was executed for less!*). For this reason, we suggest that you avoid the symbols T and F in your R code and that you also avoid using T and F as variable names.

<sup>12</sup> For the full set of reserved symbols in R, see `help ("Reserved")`.

<sup>13</sup> It would, however, be difficult to make T and F reserved symbols now, because doing so would break some existing R code.

There are R functions and operators for working with logical vectors. For example, the `!` (“not”) operator negates a logical vector:

```
!logical.values
```

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

If we use logical values in arithmetic, R treats FALSE as if it were zero and TRUE as if it were 1:

```
sum (logical.values)
```

```
[1] 3
```

```
sum (!logical.values)
```

```
[1] 1
```

If we create a vector mixing character strings, logical values, and numbers, we produce a vector of character strings:

```
c ("A", FALSE, 3.0)
```

```
[1] "A" "FALSE" "3"
```

A vector of mixed numbers and logical values is treated as numeric, with FALSE becoming zero and TRUE becoming 1:

```
c(10, FALSE, -6.5, TRUE)
```

```
[1] 10.0 0.0 -6.5 1.0
```

These examples illustrate *coercion*: In the first case, we say that the logical and numeric values are *coerced* to character values; in the second case, the logical values are coerced to numbers. In general, coercion in R takes place naturally, and is designed to lose as little information as possible.

## 1.2.6 Indexing Vectors

If we wish to access, perhaps to print, only one of the elements of a vector, we can specify the index of the element within square brackets. For example, `x[12]` is the 12th element of the vector `x`:

```
x[12] # 12th element
```

```
[1] 1.8173
```

```
words[2] # second element
```

```
[1] "be"
```

```
logical.values[3] # third element
```

```
[1] FALSE
```

We may also specify a vector of indices:

```
x[6:15] # elements 6 through 15
```

```
[1] -1.81796 0.63010 -0.27618 -0.28416 -0.91932 -0.11625  
[7] 1.81731 0.37063 0.52022 -0.75053
```

```
x[c(1, 3, 5)] # 1st, 3rd, 5th elements
```

```
[1] 0.58553 -0.10930 0.60589
```

*Negative* indices cause the corresponding values of the vector to be *omitted*:

```
x[-(11:100)] # omit elements 11 through 100
```

```
[1] 0.58553 0.70947 -0.10930 -0.45350 0.60589 -1.81796  
[7] 0.63010 -0.27618 -0.28416 -0.91932
```

The parentheses around 11:100 serve to avoid generating numbers from  $-11$  to  $100$ , which would result in an error. (Try it!) In this case,  $x[-(11:100)]$  is just a convoluted way of obtaining  $x[1:10]$ , but negative indexing can be very useful in statistical data analysis, for example, to delete outliers from a computation.

Indexing a vector by a logical vector of the same length selects the elements with TRUE indices; for example,

```
v <- 1:4
```

```
v[c (TRUE, FALSE, FALSE, TRUE)]
```

```
[1] 1 4
```

Logical values frequently arise through the use of *relational operators*, all of which are *vectorized*, which means that they apply on an elementwise basis to vectors:

```
== equals  
!= not equals  
<= less than or equals  
< less than  
> greater than  
>= greater than or equals
```

The double-equals sign ( $==$ ) is used for testing equality, because  $=$  is reserved for specifying function arguments and for assignment. Using  $=$  where  $==$  is intended is a common mistake and can result either in a syntax error or, worse, in an inadvertent assignment, so be careful!

Logical values may also be used in conjunction with the *logical operators*:

```
& and (vectorized)  
&& and (for single left and right operands)  
| or (vectorized)  
|| or (for single left and right operands)  
! not (vectorized)
```

Here are some simple examples:

**1 == 2**

[1] FALSE

**1 != 2**

[1] TRUE

**1 <= 2**

[1] TRUE

**1 < 1:3**

[1] FALSE TRUE TRUE

**3:1 > 1:3**

[1] TRUE FALSE FALSE

**3:1 >= 1:3**

[1] TRUE TRUE FALSE

**TRUE & c (TRUE, FALSE)**

[1] TRUE FALSE

**c (TRUE, FALSE, FALSE) | c (TRUE, TRUE, FALSE)**

[1] TRUE TRUE FALSE

**TRUE && FALSE**

[1] FALSE

**TRUE || FALSE**

```
[1] TRUE
```

The *unvectorized* versions of the *and* (`&&`) and *or* (`||`) operators, included in the table, are primarily useful for writing R programs (see [Chapter 10](#)) and are not appropriate for indexing vectors.

An extended example illustrates the use of the comparison and logical operators in indexing:

```
(z <- x[1:10]) # first 10 elements of x
[1] 0.58553 0.70947 -0.10930 -0.45350 0.60589 -1.81796
[7] 0.63010 -0.27618 -0.28416 -0.91932
z < -0.5 # is each element less than -0.5?
[1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE
z > 0.5 # is each element greater than 0.5
[1] TRUE TRUE FALSE FALSE TRUE FALSE TRUE FALSE FALSE FALSE
z < -0.5 | z > 0.5 # < and > are of higher precedence than |
[1] TRUE TRUE FALSE FALSE TRUE TRUE TRUE FALSE FALSE TRUE
abs(z) > 0.5 # absolute value, equivalent to last expression
[1] TRUE TRUE FALSE FALSE TRUE TRUE TRUE FALSE FALSE TRUE
z[abs(z) > 0.5] # values of z for which |z| > 0.5
[1] 0.58553 0.70947 0.60589 -1.81796 0.63010 -0.91932
z[!(abs(z) > 0.5)] # values z for which |z| <= 0.5
[1] -0.10930 -0.45350 -0.27618 -0.28416
```

The `abs()` function returns the absolute value of its argument. The last of these commands uses the `!` operator to negate the logical values produced by `abs(z) > 0.5` and thus selects the numbers for which the condition is FALSE.

A couple of pointers about using the logical and relational operators:

- We need to be careful in typing `z < -0.5`; although most spaces in R

commands are optional, the space after `<` is crucial: `z <-0.5` would *assign* the value 0.5 to `z`. Even when the spaces are not *required* around operators, they usually help to clarify R commands.

- Logical operators have lower precedence than relational operators, and so `z < -0.5 | z > 0.5` is equivalent to `(z < -0.5) | (z > 0.5)`. When in doubt, parenthesize!

## 1.2.7 User-Defined Functions

As you probably guessed, R includes functions for calculating many common statistical summaries, such as the mean of a numeric vector:

**`mean (x)`**

[1] 0.2452

Recall that, as shown in the RStudio *Environment* tab, `x` is a vector of 100 standard-normal random numbers, and so this result is the mean of those 100 values.

Were there no `mean ()` function, we could nevertheless have calculated the mean straightforwardly using the built-in functions `sum ()` and `length ()`:

**`sum (x)/length (x)`**

[1] 0.2452

where the `length ()` function returns the number of elements in its argument. To do this repeatedly every time we need to compute the mean of a numeric vector would be inconvenient, and so in the absence of the standard R `mean ()` function, we could define our own `mean` function:<sup>14</sup>

**`myMean <- function (x){`**

**`sum (x)/length (x)`**

}

[14](#) When an R command like the definition of the myMean () function extends across more than one line, as it is entered into the *Console*, the > prompt changes to +, indicating the continuation of the command. As explained in the Preface, we don't show the command and continuation prompts in R input displayed in the text. The R interpreter recognizes that a command is to be continued in this manner if it's syntactically incomplete, for example, if there's an opening brace, {, unmatched by a closing brace, }.

- We define a function using the function function ().[15](#) The arguments to function (), here just x, are the *formal arguments* (also called *dummy arguments*) of the function to be defined, myMean (). An *actual argument* will appear in place of the formal argument when the function myMean () is called in an R command.
- The remainder of the function definition is an R expression, enclosed in curly braces, specifying the *body* of the function. Typically, the function body is a *compound expression* consisting of several R commands, each on a separate line (or, less commonly, separated by semicolons). The *value returned* by the function is then the value of the last command in the function body.[16](#) In our simple myMean () function, however, the function body consists of a single command.[17](#)
- The rules for naming functions are the same as for naming variables. We avoided using the name mean because we did not wish to *replace* the standard mean () function, which is a generic function with greater utility than our simple version. For example, mean () has the additional argument na.rm, which tells the function what to do if some of the elements of x are missing.[18](#)

If we had chosen to name our function mean (), then our mean () function, which resides in the global environment, would *shadow* or *mask* the standard mean () function (see [Section 2.3](#)), and calling mean () at the R command prompt would invoke our mean () function rather than the standard one. We could in this circumstance restore use of the standard mean () function simply by deleting our function from the global environment, via the command remove ("mean"). You cannot, however, delete a standard R function or a function in a package you have loaded.

- In naming functions, we prefer using *camel case*, as in myMean (), to

separate words in a function name, rather than separating words by periods: for example, `my.mean()`. Periods in function names play a special role in object-oriented programming in R (see [Section 1.7](#)), and using periods in the names of ordinary functions consequently invites confusion.<sup>19</sup> Some R users prefer to use *snake case*, which employs underscores to separate words in function and variable names: for example, `my_mean()`.

- Functions, such as `myMean()`, that you create in the global environment are listed in the *Functions* section of the RStudio *Environment* tab.
- We will introduce additional information about writing R functions as required and take up the topic more systematically in [Chapter 10](#).

[15](#) We could not resist writing that sentence! Actually, however, `function()` is a *special form*, not a true function, because its arguments (here, the formal argument `x`) are not evaluated. The distinction is technical, and it will do no harm to think of `function()` as a function that returns a function as its result.

[16](#) It is also possible to terminate function execution and explicitly return a value using `return()`. See [Chapter 10](#) for more information about writing R functions.

[17](#) When the function body comprises a single R command, it's not necessary to enclose the body in braces; thus, we *could* have defined the function more compactly as `myMean <- function(x) sum(x)/length(x)`.

[18](#) [Section 2.3.2](#) explains how missing data are represented and handled in R.

[19](#) Nevertheless, largely for historical reasons, many R functions have periods in their names, including standard functions such as `install.packages()`.

User-defined functions in R are employed in exactly the same way as the standard R functions. Indeed, most of the standard functions in R are themselves written in the R language.<sup>20</sup> Proceeding with the `myMean()` function,

**`myMean(x)`**

```
[1] 0.2452
```

**`y # from sqrt(c(1, 2, 3, 4))`**

```
[1] 1.0000 1.4142 1.7321 2.0000
```

***myMean (y)***

[1] 1.5366

***myMean (1:100)***

[1] 50.5

***myMean (sqrt (1:100))***

[1] 6.7146

[20](#) Some of the standard R functions are *primitives*, in the sense that they are defined in code written in the lower-level languages C and Fortran.

As these examples illustrate, there is no necessary correspondence between the name of the formal argument x of the function myMean () and an actual argument to the function. Function arguments are evaluated by the interpreter, and it is the *value* of the actual argument that is passed to the function, not its name. In the last example, the function call sqrt (1:100) must first be evaluated, and then the result is used as the argument to myMean ().

Function arguments, along with any variables that are defined within a function, are *local* to the function and exist only while the function executes. These *local variables* are distinct from *global variables* of the same names residing in the global environment, which, as we have seen, are listed in the RStudio

*Environment* tab.[21](#) For example, the last call to myMean () passed the value of sqrt (1:100) (i.e., the square roots of the integers from 1 to 100) to the formal argument x, but this argument is local to the function myMean () and thus did not alter the contents of the global variable x, as you can confirm by examining the *Environment* tab.

[21](#) In more advanced use of RStudio, you can pause a function while it is executing to examine variables in its local environment; see [Section 10.8](#).

Our function myMean () is used in the same way as standard R functions, and we can consequently use it in defining other functions. For example, the R function sd () can compute the standard deviation of a numeric vector. Here's a simple substitute, mySD (), which uses myMean ():

```
mySD <- function (x){  
  sqrt (sum ((x - myMean (x))^2)/(length (x) - 1))  
}  
  
mySD (1:100)  
[1] 29.011  
  
sd (1:100) # check  
[1] 29.011
```

## 1.3 Fixing Errors and Getting Help

### 1.3.1 When Things Go Wrong

Errors can result from bugs in computer software, but much more commonly, they are the fault of the user. No one is perfect, and it is impossible to use a computer without making mistakes. Part of the craft of computing is *debugging*, that is, finding and fixing errors.

- Although it never hurts to be careful, do not worry too much about generating errors. An advantage of working in an interactive system is that you can proceed step by step, fixing mistakes as you go. R is also unusually forgiving in that it is designed to restore the workspace to its previous state when a command results in an error.
- If you are unsure whether a command is properly formulated or will do what you intend, try it out. You can often debug a command by trying it on a scaled-down problem with an obvious answer. If the answer that you get differs from the one that you expected, focus your attention on the nature of the difference. Similarly, reworking examples from this book, from R help pages, or from textbooks or journal articles can help convince you that a program is working properly.<sup>22</sup>
- When you do generate an error, don't panic! Read the error or warning message carefully. Although some R error messages are cryptic, others are informative, and it is often possible to figure out the source of the error

from the message. Some of the most common errors are merely typing mistakes. For example, when the interpreter tells you that an object is not found, suspect a typing error that inadvertently produced the name of a nonexistent object, or that you forgot to load a package or to define a variable or a function used in the offending command.

- The source of an error may be subtle, particularly because an R command can generate a sequence (or *stack*) of function calls of one function by another, and the error message may originate deep within this sequence. The trace-back () function, called with no arguments, provides information about the sequence of function calls leading up to an error.

To create a simple example, we'll use the mySD () function for computing the standard deviation, defined in [Section 1.2.7](#); to remind ourselves of the definition of this function, and of myMean (), which mySD () calls, we can print the functions by typing their names (without the parentheses that would signal a function *call*), as for any R objects:<sup>23</sup>

***mySD***

```
function (x){  
  sqrt (sum ((x - myMean (x))^2)/(length (x) - 1))  
}
```

***myMean***

```
function (x){  
  sum (x)/length (x)  
}  
  
<bytecode: 0x00000001cbbdb28>
```

We deliberately produce an error by foolishly calling mySD () with a nonnumeric argument:

### ***letters***

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"  
[16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

### ***mySD (letters)***

Error in sum (x): invalid 'type' (character) of argument

The built-in variable letters contains the lowercase letters, and of course, calculating the standard deviation of character data makes no sense.

Although the source of the problem is obvious, the error occurs in the sum () function, not directly in mySD ().<sup>24</sup> The traceback () function, called after the offending command, shows the sequence of function calls culminating in the error:

### ***traceback ()***

```
2: myMean (x) at #2
```

```
1: mySD (letters)
```

Here, “at #2” refers to the second line of the definition of mySD (), where myMean () is called.

- Not all mistakes generate error messages. Indeed, the ones that do not are more pernicious, because you may fail to notice them. Always check your output for reasonableness, and investigate suspicious results. It’s also generally a bad idea to ignore *warnings*, even though, unlike errors, they don’t prevent the completion of a computation.
- If you need to interrupt the execution of a command, you may do so by pressing the Esc (escape) key or by using the mouse to press the *Stop* button at the upper right of the RStudio *Console* pane.<sup>25</sup>

<sup>25</sup> The *Stop* button only appears *during* a computation.

<sup>22</sup> Sometimes, however, testing may convince you that the published results are

wrong, but that is another story.

[23](#) The “bytecode” message below the listing of myMean () indicates that R has translated the function into a form that executes more quickly.

[24](#) Were it programmed more carefully, mySD () would perform sanity checks on its argument and report a more informative error message when, as here, the argument is nonsensical.

### 1.3.2 Getting Help and Information

What should you do if the information provided by a call to the help () function is insufficient or if you don’t know the name of the function that you want to use? You may not even know whether a function to perform a specific task *exists* in the standard R distribution or in one of the contributed packages on CRAN. This is not an insignificant problem, for there are hundreds of functions in the standard R packages and many thousands of functions in the more than 12,000 packages on CRAN.

Although there is no completely adequate solution to this problem, there are several R resources beyond help () and ? that can be of assistance:

- The apropos () command searches for currently accessible objects whose names contain a particular character string.[26](#) For example,

```
apropos ("log")
```

```
....  
[11] "is.logical"           "log"  
[13] "log10"                 "log1p"  
[15] "log2"                  "logb"  
[17] "Logic"                 "logical"  
....
```

[26](#) The apropos () function can also search for character patterns called *regular*

*expressions* (which are discussed in [Section 2.6](#)).

If we're looking for a function to compute logs, this command turns up some relevant results (e.g., `log`, `log10`) and many irrelevant ones.

- Casting a broader net, the `help.search()` command examines the titles and certain other fields in the help pages of all R packages installed in your library, showing the results in the RStudio *Help* tab (which also has its own search box, at the top right of the tab). For example, try the command `help.search("loglinear")` to find functions related to loglinear models (discussed in [Section 6.6](#)). The `??` operator is a synonym for `help.search()`, for example, `??loglinear`.
- If you have an active internet connection, you can search even more broadly with the `RSiteSearch()` function. For example, to look in all standard and CRAN packages, even those not installed on your system, for functions related to loglinear models, you can issue the command

***RSiteSearch ("loglinear", restrict="functions")***

The results appear in a web browser. See `help("RSiteSearch")` for details.

- The CRAN *task views* are documents that describe resources in R for applications in specific areas, such as Bayesian statistics, econometrics, psychometrics, social statistics, and spatial statistics. There are (at the time of writing) more than 30 task views, available via the command `carWeb("taskviews")` (using the `carWeb()` function from the `car` package), directly by pointing your web browser at <https://cran.r-project.org/web/views/>, or in the home screen of the RStudio *Help* tab.
- The command `help(package="package-name")`, for example, `help(package="car")`, displays information about an installed package in the RStudio *Help* tab, including a hyperlinked index of help topics documented in the package.
- Some packages contain *vignettes*, discursive documents describing the use of the package. To find out what vignettes are available in the packages installed on your system, enter the command `vignette()`. The command `vignette(package="package-name")` displays the vignettes available in a particular installed package, and the command `vignette("vignette-name")` or `vignette("vignette-name", package="package-name")` opens a specific vignette.
- R and RStudio have a number of resources and references available, both locally and on the internet. An index to the key resources can be obtained on the

RStudio *Help* tab by clicking on the *Home* icon in the toolbar at the top of the tab.

- Help on R is available on the internet from many other sources. A Google search for R residualPlot, for example, will lead to a page at <https://www.rdocumentation.org> for the residualPlot () function in the **car** package. The web page <https://www.r-project.org/help.html> has several suggestions for getting help with R, including information on the R email lists and the StackOverflow@StackOverflow website discussed in the Preface to this book. Also see <http://search.r-project.org/>.

## 1.4 Organizing Your Work in R and RStudio and Making It Reproducible

If you organize your data analysis workflow carefully, you'll be able to understand what you did even after some time has elapsed, to reproduce and extend your work with a minimum of effort, and to communicate your research clearly and precisely. RStudio has powerful tools to help you organize your work in R and to make it reproducible. Working in RStudio is surprisingly simple, and it is our goal in this section to help you get started. As you become familiar with RStudio, you may wish to consult the RStudio documentation, conveniently accessible from the *Help* tab, as described at the end of the preceding section, to learn to do more.

We showed in [Section 1.2.1](#) how to interact with the R interpreter by typing commands directly in the *Console*, but that's not generally an effective way to work with R. Although commands typed into the *Console* can be recovered from the *History* tab and by the up-and down-arrow keys at the command prompt, no permanent and complete record of your work is retained. That said, it sometimes makes sense to type *particular* commands directly at the R command prompt—for example, `help()` and `install.packages()` commands—but doing so more generally is not an effective way to organize your work.

We'll describe two better strategies here: writing annotated scripts of R commands and writing R Markdown documents.<sup>27</sup>

<sup>27</sup> RStudio is also capable of creating and working with other kinds of documents. For example, we wrote this book in LATEX using RStudio and the **knitr** package (Xie, 2015, 2018), which support the creation of LATEX

documents that incorporate R code, in the same manner as R Markdown supports the incorporation of R code in a Markdown document.

If you have not already done so, we suggest that you now create an RStudio project named R-Companion, as described in [Section 1.1](#). Doing so will give you access to several files used both in this section and elsewhere in the book.

### 1.4.1 Using the RStudio Editor With R Script Files

An *R script* is a plain-text document containing a sequence of R commands. Rather than typing commands directly at the R command prompt, you enter commands into the script and execute them from the script, either command by command, several selected commands at once, or the whole script as a unit. Saving the script in a file creates a permanent record of your work that you can use to reproduce, correct, and extend your data analysis.

RStudio makes it easy to create, work with, and manage R scripts. A new script is created by selecting *File > New File > R Script* from the RStudio menus. The first time that you do this, a new source-editor pane will open on the top left of the RStudio window, and the *Console* will shrink to create space for the new pane. You should now have one tab, called *Untitled1*, in the source pane. When you create a new script in this manner, you should normally save the script in your project directory, using *File > Save*, and giving the script a file name ending in the file type .R. RStudio saves R script files as plain-text files.

RStudio recognizes files with extension .R or .r as R scripts and handles these files accordingly, for example by highlighting the syntax of R code (e.g., comments are colored green by default), by automatically indenting continuation lines of multiline R commands as you enter them, and by providing tools for executing R commands.<sup>28</sup> In a subsequent session, you can reopen an existing R script file in several ways: The simplest method is to click on the file's name in the RStudio *Files* tab. Alternatively, you can use the *File > Open File* or *File > Recent Files* menu. Finally, files that were open in tabs in the RStudio source-editor pane at the end of a session are reopened automatically in a subsequent session.

<sup>28</sup> You can customize the RStudio editor in the *Appearance* tab on *Tools > Global Profile*

If you liberally sprinkle your R scripts with explanatory comments and notes, you'll be in a better position to understand what you did when you revisit a script at some future date. Scripts provide a simple and generally effective, if crude, means of making your work reproducible.

The file chap-1.R that you downloaded to your project folder in [Section 1.1](#) includes all the R commands used in [Chapter 1](#) of the *R Companion*. [Figure 1.4](#) shows this script open in the RStudio source-editor pane, along with the *Console* pane. As with most editors, you can click the mouse anywhere you like and start typing. You can erase text with the Delete or Backspace key. Common editing gestures are supported, like double-clicking to select a word, triple-clicking to select a whole line, and left-clicking and dragging the mouse across text to select it. In [Figure 1.4](#), we clicked and dragged to select five lines, and then pressed the *Run* button at the top right of the source pane. The selected commands were sent to the R interpreter, as if they had been typed directly at the R command prompt; both the input commands and associated printed output appear in the *Console*, as shown in [Figure 1.4](#).

**Figure 1.4** The left-hand panes of the RStudio window. The script files chap-1.R and Untitled1 are in the source-editor pane (at the top). We highlighted several commands in chap-1.R by left-clicking and dragging the mouse over them and caused these commands to be executed by pressing the *Run* button in the toolbar at the top right of the editor tab for the file. The commands and resulting output appear in the R *Console* pane (at the bottom).

The screenshot shows the RStudio interface. At the top, there are two tabs: 'Untitled1' and 'chap-1.R'. The 'chap-1.R' tab is active, showing an R script with code demonstrating basic arithmetic operations. Lines 1 through 14 are comments about the script, while lines 15 through 20 show the results of running the arithmetic code. Below the script editor is a 'Run' toolbar with various icons. The main workspace below the editor contains a 'Console' tab which is currently selected, showing the R command-line output for each arithmetic operation.

```

1 ##-----##  

2 ## An R Companion to Applied Regression, 3rd Edition ##  

3 ## J. Fox and S. Weisberg, Sage Publications ##  

4 ## Script for Chapter 1 ##  

5 ##-----##  

6  

7 library("car")  

8 carweb(setup=TRUE)  

9  

10 2 + 3 # addition  

11 2 - 3 # subtraction  

12 2*3 # multiplication  

13 2/3 # division  

14 2^3 # exponentiation|  

15  

16 4^2 - 3^2  

17  

18 1 - 6 + 4  

19  

20 (4^2) - (3^2)  

14:23 (Top Level) ⇣ R Script ⇣

```

```

Console Terminal ×
~/
> 2 + 3 # addition
[1] 5
> 2 - 3 # subtraction
[1] -1
> 2*3 # multiplication
[1] 6
> 2/3 # division
[1] 0.6666667
> 2^3 # exponentiation
[1] 8
>

```

Some controls in the RStudio editor depend on the type of file that you are editing. After a while, you will likely find these context-dependent changes to be intuitive. For .R files, important editing operations are located in the RStudio *File* and *Edit* menus, and in the toolbar at the top of the file tab, as shown in [Figure 1.4](#).

- To save a file, use the *Save* or *Save as* items in the *File* menu, click on the disk image in the main toolbar at the top of the source-editor pane, or click on the disk image in the toolbar at the top of the document tab.
- To find a text string, or to find and replace text, either click on the spyglass in the toolbar at the top of the document tab, or select *Edit > Find* from the RStudio menus. Finding and replacing text works as it does in most editors.
- To run selected code, click on the *Run* button at the right of the toolbar above the document tab. To run a single line of code, put the cursor in that line and press *Run*. You can also choose *Code > Run Selected Line (s)* from

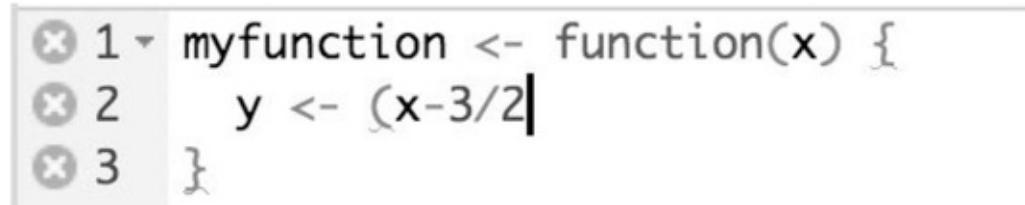
the RStudio menus.

- If *Source on Save* at the top left of the document tab is checked, then whenever you save the script file, all of its contents are sent to the R interpreter (i.e., the file is “sourced”). Similarly, the *Source* button at the top right of the document tab sends all of the commands in the script to the interpreter but without saving the file. The *Source* button is actually a menu; click on its inverted triangle to see the various options.

All of these actions have keyboard equivalents that you can discover from the menu items or (in most cases) by hovering your mouse over the toolbar icons. Two key-combinations that we use frequently are Ctrl-F (on Windows) or command-F (on macOS), which opens the find/replace tool, and Ctrl-Enter or command-return, which runs the currently selected command or commands. You can open a window with all of RStudio’s many keyboard equivalents via the key-combination Alt-Shift-K (or option-shift-K on a Mac).

The RStudio editor has sophisticated tools for writing R code, including tools that are very helpful for finding and correcting errors, such as bugs in user-defined functions.<sup>29</sup> For example, the snippet from the source editor shown in [Figure 1.5](#) contains code defining a function that will not run correctly, as indicated by the (red) x-marks to the left of the numbers of the offending lines. Hovering the mouse over one of the x-marks reveals the problem, an unmatched opening parenthesis. Of course, RStudio can’t tell *where* you want the missing closing parenthesis to go. When you insert the matching parenthesis, the error marks go away.

**Figure 1.5** Snippet from the RStudio source error showing the definition of a function that contains unmatched parentheses. The error is flagged by the x-marks to the left of the lines in the function definition.



```
1 > myfunction <- function(x) {
2 >   y <- (x-3/2|
3 > }
```

<sup>29</sup> RStudio also incorporates tools for debugging R programs; see [Section 10.8](#).

The editor and, for that matter, the *Console* automatically insert and check for matching parentheses, square brackets, curly braces, and quotation marks. The automatic insertion of delimiters can be unnerving at first, but most users

eventually find this behavior helpful.<sup>30</sup>

<sup>30</sup> If automatic insertion of delimiters annoys you, you can turn it off in the *General* tab of the *Tools > Global Options* dialog. Indeed, many features of the RStudio editor can be customized in the *General*, *Code*, and *Appearance* tabs of this dialog, so if you encounter an editor behavior that you dislike, you can probably change it.

## 1.4.2 Writing R Markdown Documents

R Markdown documents take script files to a new level by allowing you to mix R commands with explanatory text. Think of an R Markdown document as an R script on steroids. Your R Markdown source document is compiled into an output report evaluating the R commands in the source document to produce easily reproducible results in an aesthetically pleasing form. We believe that for most readers of this book, R Markdown is the most effective means of using R for statistical data analysis.

Markdown, on which R Markdown is based, is a simple, punningly named, *text-markup* language, with simple conventions for adding the main features of a typeset document, such as a title, author, date, sections, bulleted and numbered lists, choices of fonts, and so on. R Markdown enhances Markdown by allowing you to incorporate *code chunks* of R commands into the document. You can press the *Knit* button in the toolbar at the top left of the tab containing an R Markdown document tab to compile the document into a typeset report. When an R Markdown document is compiled, the R commands in the document are run in an independent R session, and the commands and associated output appear in the report that is produced—just as the commands and output would have appeared in the *Console* or the *Plots* tab if you had typed the R commands in the document into the *Console*. As we will explain, it is also possible to display R text output and graphical output associated with a code chunk *directly* in the R Markdown source document.

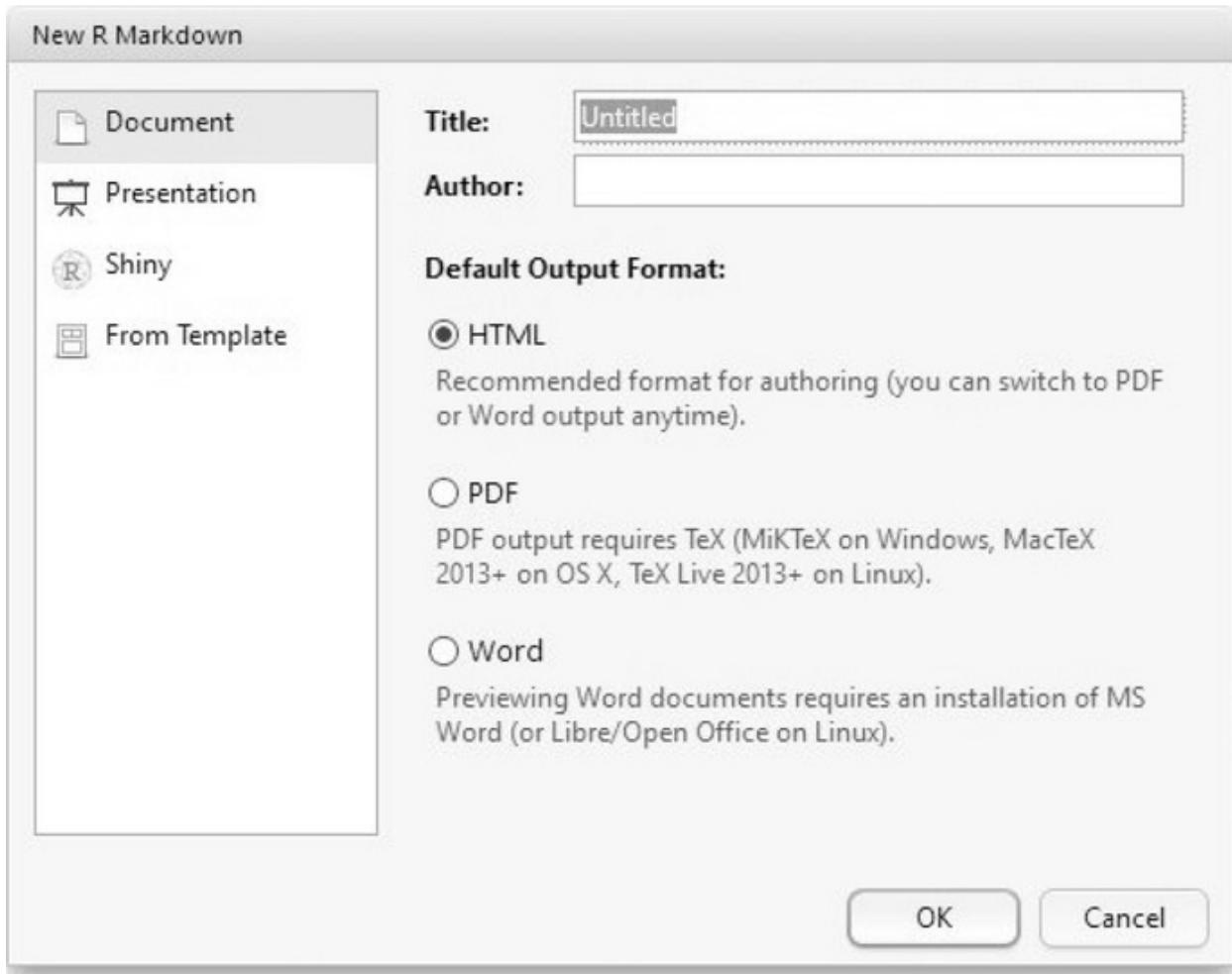
R Markdown is sufficiently simple that most users will be able to produce attractively typeset reports using only the instructions we supply here. If you want to learn more about R Markdown, you can select *Help > Markdown Quick Reference* from the RStudio menu, which will open the *Markdown Quick Reference* in the RStudio *Help* tab. This document summarizes many of the

formatting commands available in Markdown and has a link to more detailed information on the RStudio website. The R Markdown “cheatsheet,” from *Help > Cheatsheets*, provides similar information in a compact form that is suitable for printing. Finally, for a book-length treatment of R Markdown and other kinds of dynamic R documents, consult Xie (2015).<sup>31</sup>

<sup>31</sup> Xie (2015) is written by the author of the **knitr** package and a coauthor of the **rmarkdown** package; these packages are the mechanism that RStudio uses to compile R Markdown documents into reports.

You can create a new R Markdown document by selecting *File > New File > R Markdown...* from the RStudio menus, producing the dialog box shown in [Figure 1.6](#). Fill in the *Title* field in the dialog with Test of RMarkdown and the *Author* field with your name. Leave the other selections at their defaults, creating a document that will produce an HTML (web-page) report.<sup>32</sup>

**Figure 1.6** The *New R Markdown* dialog box, produced by selecting *File > New File > R Markdown* from the RStudio menus.



[32](#) If you want to produce PDF output, you must first install LATEX on your computer; instructions for downloading and installing LATEX are given in the Preface. Once LATEX is installed, RStudio will find and use it. Another option is to produce a Word document. We discourage you from doing so, however, because you may be tempted to edit the Word document directly, rather than modifying the R Markdown document that produced it, thus losing the ability to reproduce your work.

A skeleton R Markdown source document opens in a tab in the RStudio source-editor pane, as shown in [Figure 1.7](#). We recommend that you immediately rename and save this document as a file of type .Rmd, say RMarkdownTest.Rmd. The source document consists of three components: a *header*, comprising the first six lines; ordinary free-form discursive text, intermixed with simple Markdown formatting instructions; and *code chunks* of commands to be executed by the R interpreter when the source document is compiled into a report.

**Figure 1.7** The skeleton R Markdown file produced by the menu selection *File > New File > R Markdown....*

The screenshot shows the RStudio interface with an R Markdown file open. The title bar says "Untitled1\*". The toolbar includes icons for file operations, ABC, search, Knit, Insert, Run, and other utilities. The code editor contains the following R Markdown code:

```
1 ---  
2 title: "Test of RMarkdown"  
3 author: "John Fox and Sanford Weisberg"  
4 date: "3/24/2017"  
5 output: html_document  
6 ---  
7  
8 ```{r setup, include=FALSE}  
9 knitr::opts_chunk$set(echo = TRUE)  
10 ````  
11  
12 ## R Markdown  
13  
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring  
HTML, PDF, and MS Word documents. For more details on using R Markdown see  
http://rmarkdown.rstudio.com.  
15  
16 When you click the **Knit** button a document will be generated that includes both  
content as well as the output of any embedded R code chunks within the document. You can  
embed an R code chunk like this:  
17  
18 ```{r cars}  
19 summary(cars)  
20 ````  
21  
22 ## Including Plots  
23  
24 You can also embed plots, for example:  
25  
26 ```{r pressure, echo=FALSE}  
27 plot(pressure)  
28 ````  
29  
30 Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of  
the R code that generated the plot.  
31
```

The status bar at the bottom shows "3:30" and "Test of RMarkdown".

The document header is filled in automatically and includes title, author, date, and output format fields. You can edit the header as you can edit any other part of the R Markdown document. Consult the R Markdown documentation for more information on the document header.

The text component of the document can include simple Markdown formatting instructions. For example, any line that begins with a single # is a major section heading, ## is a secondary section heading, and so on. The *Markdown Quick Reference* lists formatting markup for italics, bold face, bulleted and numbered lists, and so on. Text surrounded by back-ticks, for example, `echo = FALSE`, is set in a typewriter font, suitable for representing computer code or possibly

variable names.

R code chunks appear in gray boxes in an R Markdown document and are demarcated by the line `r` at the top of the chunk and the line ` (three back-ticks) at the bottom. The initial line may also include *chunk options*. You can name a chunk—for example, setup in the first chunk in the sample document and cars in the second chunk—or supply other options, such as echo=FALSE, which suppresses printing the commands in a block in the compiled report, showing only the output from the commands, or include=FALSE, which suppresses printing both the commands and associated output. If multiple chunk options are supplied, they must be separated by commas. For more information about R Markdown chunk options, see the webpage at <https://yihui.name/knitr/options/>, or click the gear icon at the top right of a code chunk and follow the link to *Chunk options*.

You can insert a new code chunk in your R Markdown document manually by typing the initial line `r` and terminal line ` or by positioning the cursor at the start of a line and pressing the *Insert* button near the top right of the document tab and selecting *R* from the resulting drop-down list. Click on the gear icon at the top right of a gray code chunk to set some chunk options. The other two icons at the top right of the code chunk allow you to run either all the code chunks *before* the current chunk (the down-pointing triangle) or to run the *current* code chunk (the right-pointing triangle). Hover the mouse cursor over each icon for a “tool tip” indicating the function of the icon.

The body of each code chunk consists of commands to be executed by the R interpreter, along with comments beginning with #, which the interpreter will ignore. The various chunk options tell R Markdown how to treat the R input and output in the HTML document that it produces. Almost any R commands may be included in a code chunk.<sup>33</sup>

<sup>33</sup> It is best, however, to avoid commands that require direct user intervention; examples of such commands include identify (), for interactive point identification in a plot, and file.choose () for interactive file selection.

In addition to code chunks of R commands, you can include *in-line* executable R code, between `r and another back-tick, `. For example, a sentence in your R Markdown document that reads

The number of combinations of six items taken two at a time is `r choose(6, 2)`.

would be typeset by evaluating the R command `choose(6, 2)` and replacing the command with its value, 15—the number of ways of choosing two objects from among six objects, computed by the `choose()` function: The number of combinations of six items taken two at a time is 15.

Moreover, if your document assigns values to the variables `n <- 6` and `k <- 2` in a previous code chunk, you could replace `choose(6, 2)` with `choose(n, k)`, and R would substitute the values of these variables correctly into the command.

Turning `RMarkdownTest.Rmd` into a typeset report can be accomplished effortlessly by pressing the *Knit* button at the top left of the source document tab.<sup>34</sup>

<sup>34</sup> This procedure is essentially effortless only if there are no errors in an R Markdown document. Errors in code chunks typically produce R error messages, and fixing these errors is no different from debugging a script (see [Section 1.3.1](#)). You should, however, debug the code chunks interactively *before* compiling the document, as we will explain presently. Markdown errors, in contrast, may cause the document to fail to compile or may result in undesired formatting in the typeset report. Particularly if the document fails to compile, you should examine the output in the *R Markdown* tab, which appears in the lower-right pane in RStudio alongside the *Console* and may contain informative error messages.

The resulting report, `RMarkdownTest.html`, is displayed in RStudio’s built-in web-page viewer and is automatically saved in your project directory. R code in the report appears in gray boxes, and associated text output appears in white boxes immediately below the command or commands that produced it. The `R >` command prompt is suppressed, and each line of output is preceded by two R comment characters, `##`. The rationale for formatting commands and output in this manner is that commands can then be copied and pasted directly into R scripts or at the R command prompt without having to remove extraneous `>`s, while output preceded by `##` will be treated as R comments. If you prefer to include the command prompts and suppress the hash-marks in the compiled document, you can alter the first code chunk in the document to read

```
```{r setup, include=FALSE, eval=FALSE}

knitr::opts_chunk$set (echo=TRUE, prompt=TRUE, comment="")

```

```

When you develop an R script, you normally execute commands as you write them, correcting and elaborating the commands in a natural manner. R Markdown source documents support a similar style of work: After you type a new command into a code chunk, you can execute it by pressing the *Run* button at the top of the document tab and choosing *Run Selected Line (s)* from the dropdown list, pressing the key-combination Control-Enter (as you would for a script), or selecting *Code > Run Selected Line (s)* from the RStudio menus. The command is executed in the R *Console*, and any output from the command—both text and graphics—appears as well in the source-editor tab for the R Markdown document. This in-document output is independent of the web-page report that's produced when you compile the whole document. You can, consequently, maximize the editor pane, covering the *Console*, and still see the output. We recommend this approach to developing R Markdown documents interactively.

You can also execute an entire code chunk by selecting *Run Current Chunk* from the *Run* button list or via *Code > Run Current Chunk*. Alternatively, and most conveniently, each code chunk has small icons at the upper right of the chunk; the right-pointing arrow runs the code in the chunk, displaying output immediately below. See [Figure 1.8](#) for an example.

**Figure 1.8** Part of the illustrative R Markdown document (displayed in its entirety in [Figure 1.7](#)) as it appears in RStudio. The in-document output was produced by pressing the arrow at the top right of each code chunk to run the code in the chunk.

```

12 ## R Markdown
13
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS
Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.
15
16 When you click the **Knit** button a document will be generated that includes both content as well as
the output of any embedded R code chunks within the document. You can embed an R code chunk like this:
17
18 ````{r cars}
19 summary(cars)
20 ````

      speed          dist
Min.   : 4.0   Min.   : 2.00
1st Qu.:12.0  1st Qu.: 26.00
Median :15.0  Median : 36.00
Mean    :15.4  Mean    : 42.98
3rd Qu.:19.0  3rd Qu.: 56.00
Max.   :25.0  Max.   :120.00

21
22 ## Including Plots
23
24 You can also embed plots, for example:
25
26 ````{r pressure, echo=FALSE}
27 plot(pressure)
28 ````



```

## 1.5 An Extended Illustration: Duncan's Occupational-Prestige Regression

In this section, we use standard R functions along with functions in the **car** package in a typical linear regression problem. An R Markdown document that repeats all the R commands, but with minimal explanatory text, is in the file `Duncan.Rmd` that you downloaded to your R-Companion project in [Section 1.1](#).<sup>35</sup> Following the instructions in [Section 1.4.2](#), you can typeset this document by clicking on `Duncan.Rmd` in the RStudio *Files* tab to open the document in the

source editor and then clicking the *Knit* button in the toolbar at the top of the document tab.

[35](#) If you choose not to create an R-Companion project, you can still obtain this file by the command carWeb (file="Duncan.Rmd"), which downloads the file to your RStudio working directory.

The data for this example are in the **carData** package, which is automatically loaded when you load the **car** package, as we've done earlier in this chapter. Working with your own data almost always requires the preliminary work of importing the data from an external source, such as an Excel spreadsheet or a plain-text data file. After the data are imported, you usually must perform various data management tasks to prepare the data for analysis. Importing and managing data in R are the subject of the next chapter.

The Duncan data set that we use in this section was originally analyzed by the sociologist Otis Dudley Duncan (1961). The first 10 lines of the Duncan *data frame* are printed by the head () command:

```
head(Duncan, n=10)
```

|            |      | type | income | education | prestige |
|------------|------|------|--------|-----------|----------|
| accountant | prof | 62   | 86     | 82        |          |
| pilot      | prof | 72   | 76     | 83        |          |
| architect  | prof | 75   | 92     | 90        |          |
| author     | prof | 55   | 90     | 76        |          |
| chemist    | prof | 64   | 86     | 90        |          |
| minister   | prof | 21   | 84     | 87        |          |
| professor  | prof | 64   | 93     | 93        |          |
| dentist    | prof | 80   | 100    | 90        |          |
| reporter   | wc   | 67   | 87     | 52        |          |
| engineer   | prof | 72   | 86     | 88        |          |

```
dim(Duncan)
```

```
[1] 45 4
```

A data frame is the standard form of a rectangular data set in R, a two-dimensional array of data with columns corresponding to variables and rows corresponding to cases or observations (see [Section 2.3](#)). The first line of the output shows the names for the variables, type, income, education, and prestige. Each subsequent line contains data for one case. The cases are occupations, and the first entry in each line is the name of an occupation, generally called a *row name*. There is no variable in the data frame corresponding to the row names, but you can access the row names by the command `row.names(Duncan)` or `rownames(Duncan)`. The `dim()` (“dimensions”) command reveals that the Duncan data frame has 45 rows (cases) and four columns (variables). Because this is a small data set, you could print it in full in the *Console* simply by entering its name, `Duncan`, at the R command prompt. You can also open the data frame in a spreadsheet-like tab in the RStudio source-editor pane by entering the command `View(Duncan)`.

[36](#) The `View()` function shouldn’t be used in an R Markdown document,

however, because it creates output outside the *Console* and *Plots* tab. Although the RStudio data set viewer *looks* like a spreadsheet, you can't edit a data frame in the viewer.

The definitions of the variables in Duncan's data set are given by help ("Duncan") and are as follows:

- type: Type of occupation, prof (professional and managerial), wc (white-collar), or bc (blue-collar)
- income: Percentage of occupational incumbents in the 1950 U.S. Census who earned \$3,500 or more per year (about \$36,000 in 2017 U.S. dollars)
- education: Percentage of occupational incumbents in 1950 who were high school graduates (which, were we cynical, we would say is roughly equivalent to a PhD in 2017)
- prestige: Percentage of respondents in a social survey who rated the occupation as "good" or better in prestige

The variable type has character strings for values rather than numbers, and is called a *factor* with the three *levels* or categories "prof", "wc", and "bc".<sup>37</sup> The other variables are numeric. Duncan used a linear least-squares regression of prestige on income and education to predict the prestige of occupations for which the income and educational scores were known from the U.S. Census but for which there were no direct prestige ratings. He did not use occupational type in his analysis.

[37](#) For efficiency of storage, values of a factor are actually coded numerically as integers, and the corresponding level names are recorded as an *attribute* of the factor. See [Chapter 2](#) and [Section 4.7](#) for more on working with factors.

This is a small data frame in an era of "big data," but, for several reasons, we think that it makes a nice preliminary example:

- Duncan's use of multiple regression to analyze the data was unusual at the time in sociology, and thus his analysis is of historical interest.
- Duncan's methodology—using a regression for a subset of occupations to impute prestige scores for all occupations—is still used to create occupational socioeconomic status scales and consequently is not just of historical interest.
- The story of Duncan's regression analysis is in part a cautionary tale,

reminding us to check that our statistical models adequately summarize the data at hand.

The generic summary () function has a *method* that is appropriate for data frames. As described in [Section 1.7](#), generic functions adapt their behavior to their arguments. Thus, a function such as summary () may be used appropriately with many different kinds of objects. This ability to reuse the same generic function for many similar purposes is one of the strengths of R. When applied to the Duncan data frame, summary () produces the following output:

**summary (Duncan)**

| type    | income       | education     | prestige     |
|---------|--------------|---------------|--------------|
| bc :21  | Min. : 7.0   | Min. : 7.0    | Min. : 3.0   |
| prof:18 | 1st Qu.:21.0 | 1st Qu.: 26.0 | 1st Qu.:16.0 |
| wc : 6  | Median :42.0 | Median : 45.0 | Median :41.0 |
|         | Mean :41.9   | Mean : 52.6   | Mean :47.7   |
|         | 3rd Qu.:64.0 | 3rd Qu.: 84.0 | 3rd Qu.:81.0 |
|         | Max. :81.0   | Max. :100.0   | Max. :97.0   |

The function counts the number of cases in each level of the factor type and reports the minimum, maximum, median, mean, and the first and third quartiles for each numeric variable, income, education, and prestige.

### 1.5.1 Examining the Data

A sensible place to start any data analysis, including a regression analysis, is to visualize the data using a variety of graphical displays. [Figure 1.9](#), for example, shows a histogram for the response variable prestige, produced by a call to the hist () function:

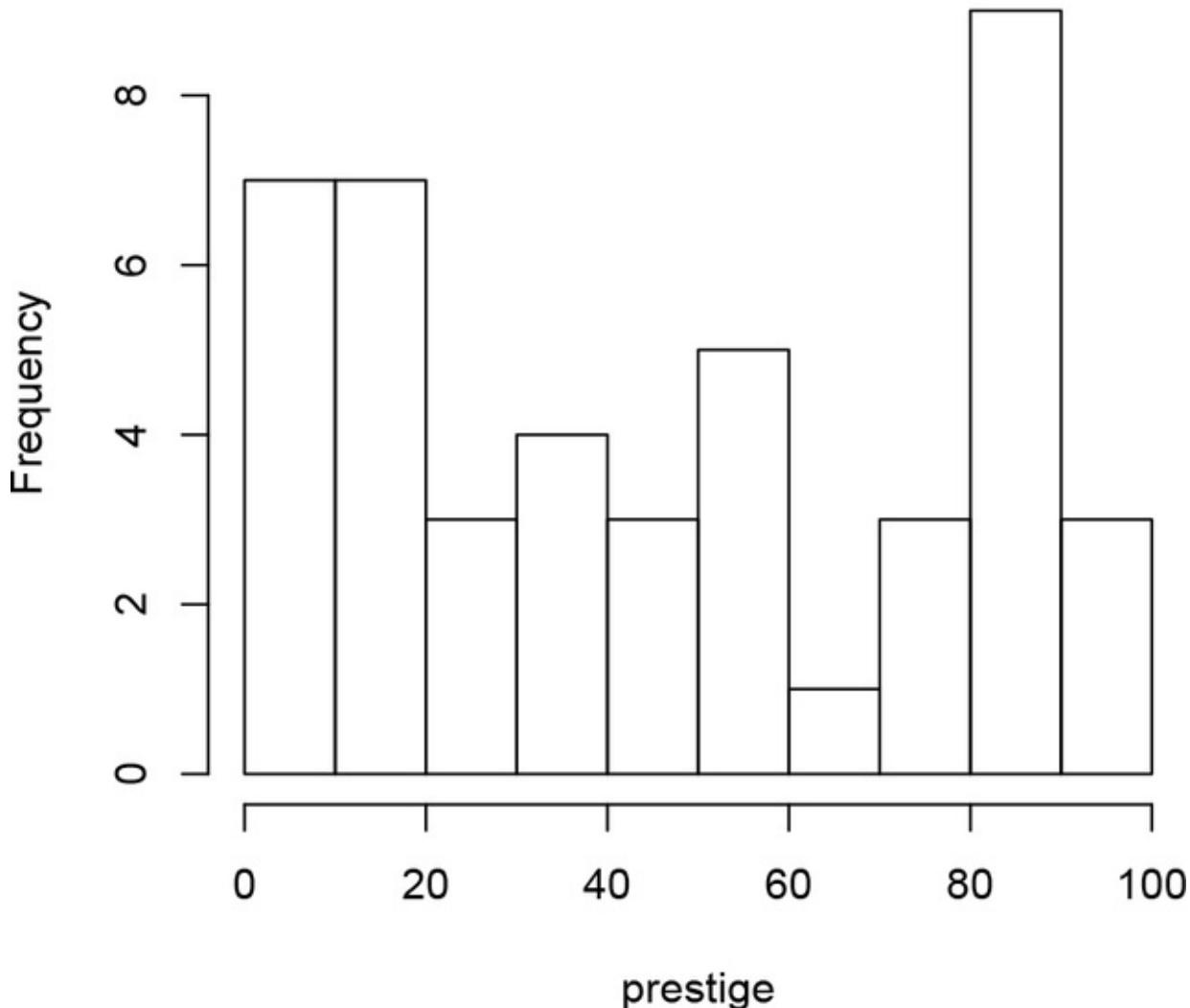
**with (Duncan, hist (prestige))**

The with () function makes the prestige variable in the Duncan data frame available to hist ().<sup>38</sup> The hist () function doesn't return a visible value in the R console but rather is used for the *side effect* of drawing a graph, in this case a

histogram.<sup>39</sup> Entering the `hist()` command from an R script displays the histogram in the RStudio *Plots* tab; entering the command in an R Markdown document displays the graph directly in the document and in the HTML report compiled from the R Markdown document.

**Figure 1.9** Histogram for prestige in Duncan's data frame.

### Histogram of prestige



<sup>38</sup> The general format of a call to `with()` is with *(data-frame, R-command)*, where *R-command* could be a *compound expression* enclosed in braces, { }, and comprising several commands, each command on its own line or separated from other commands by semicolons. We discuss managing and using data in data

frames in [Chapter 2](#).

[39](#) Like all R functions, `hist()` *does* return a result; in this case, however, the result is *invisible* and is a list containing the information necessary to draw the histogram. To render the result visible, put parentheses around the command: `(with (Duncan, hist (prestige)))`. Lists are discussed in [Section 2.4](#).

The distribution of prestige appears to be bimodal, with cases stacking up near the boundaries, as many occupations are either low prestige, near the lower boundary, or high prestige, near the upper boundary, with relatively fewer occupations in the middle bins of the histogram. Because prestige is a percentage, this behavior is not altogether unexpected. Variables such as this often need to be transformed, perhaps with a logit (log-odds) or similar transformation, as discussed in [Section 3.4](#). Transforming prestige turns out to be unnecessary in this problem.

Before fitting a regression model to the data, we should also examine the distributions of the predictors education and income, along with the relationship between prestige and each predictor, and the relationship between the two predictors.[40](#) The `scatterplotMatrix()` function in the **car** package associated with this book allows us to conveniently examine these distributions and relationships.[41](#)

```
scatterplotMatrix (~ prestige + education + income, id=list (n=3),  
data=Duncan)
```

[40](#) We will ignore the additional predictor type, which, as we mentioned, didn't figure in Duncan's analysis of the data.

[41](#) Because we previously loaded the **car** package for access to the Duncan data frame, we do not need to do so again.

The `scatterplotMatrix()` function uses a *one-sided formula* to specify the variables to appear in the graph, where we read the formula `~ prestige + education + income` as "plot prestige and education and income." The `data` argument tells `scatterplotMatrix()` where to find the variables.[42](#) The argument `id=list (n=3)` tells `scatterplotMatrix()` to identify the three most unusual points in

each panel.<sup>43</sup> We added this argument after examining a preliminary plot of the data. Using a script or typing the `scatterplotMatrix()` command directly into the *Console* causes the graph to be shown in the RStudio *Plots* tab. You can view a larger version of the graph in its own window by pressing the *Zoom* button at the top of the *Plots* tab. As explained in [Section 1.4.2](#), if you’re working in an R Markdown document, you can display the graph directly in the source document.

[42](#) We’ll encounter formulas again when we specify a regression model for Duncan’s data (in [Section 1.5.2](#)), and the topic will be developed in detail in [Chapter 4](#) on linear models, particularly in [Section 4.9.1](#).

[43](#) Point identification in the `car` package is explained in [Section 3.5](#).

The scatterplot matrix for prestige, education, and income appears in [Figure 1.10](#). The variable names in the diagonal panels label the axes. The scatterplot in the upper-right-hand corner, for example, has income on the horizontal axis and prestige on the vertical axis. By default, *nonparametric density estimates*, using an adaptive-kernel estimator, appear in the diagonal panels, with a *rug-plot* (“one-dimensional scatterplot”) at the bottom of each panel, showing the location of the data values for the corresponding variable.<sup>44</sup> There are several lines on each scatterplot:

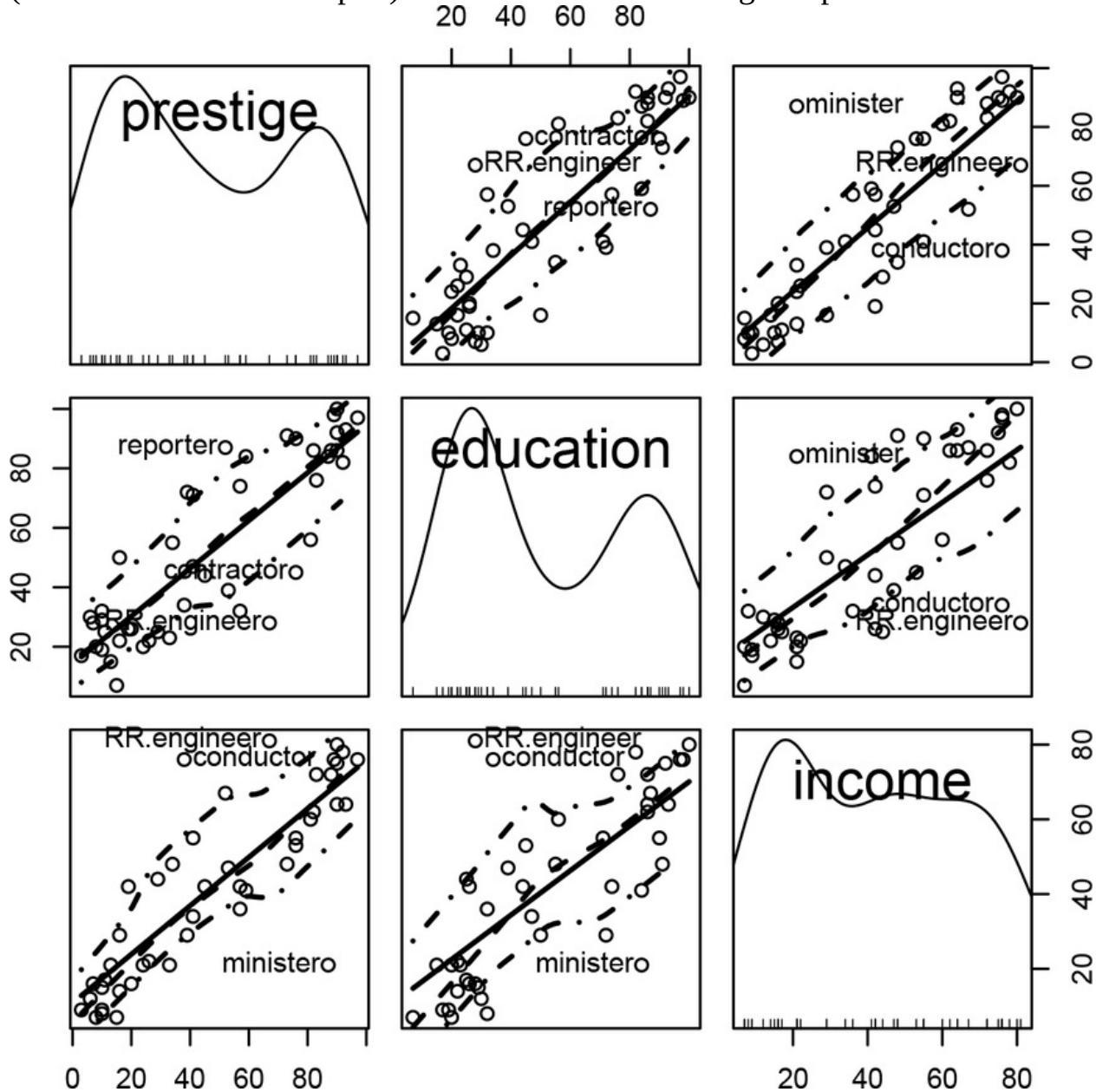
[45](#) Scatterplot smoothers in the `car` package are discussed in [Section 3.2.1](#).

- The solid line shows the marginal linear least-squares fit for the regression of the vertical-axis variable ( $y$ ) on the horizontal-axis variable ( $x$ ), ignoring the other variables.
- The central broken line is a nonparametric regression smooth, which traces how the average value of  $y$  changes as  $x$  changes without making strong assumptions about the form of the relationship between the two variables.<sup>45</sup>
- The outer broken lines represent smooths of the conditional variation of the  $y$  values given  $x$  in each panel, like running quartiles.

[44](#) Other choices are available for the diagonal panels, including histograms. We discuss `scatterplot-Matrix()` and other graphics functions in the `car` package for exploring data in [Section 3.3](#).

**Figure 1.10** Scatterplot matrix for prestige, education, and income in Duncan’s data, identifying the three most unusual points in each panel. Nonparametric

density estimates for the variables appear in the diagonal panels, with a rug-plot (one-dimensional scatterplot) at the bottom of each diagonal panel.



Like prestige, education appears to have a bimodal distribution. The distribution of income, in contrast, is perhaps best characterized as irregular. The pairwise relationships among the variables seem reasonably linear, which means that as we move from left to right across the plot, the average  $y$  values of the points more or less trace out a straight line. The scatter around the regression lines appears to have reasonably constant vertical variability and to be approximately symmetric.

In addition, two or three cases stand out from the others. In the scatterplot of income versus education, ministers are unusual in combining relatively low income with a relatively high level of education, and railroad conductors and engineers are unusual in combining relatively high levels of income with relatively low education. Because education and income are predictors in Duncan's regression, these three occupations will have relatively high *leverage* on the regression coefficients. None of these cases, however, are outliers in the *univariate* distributions of the three variables.

## 1.5.2 Regression Analysis

Duncan was interested in how prestige is related to income and education in combination. We have thus far addressed the univariate distributions of the three variables and the pairwise or marginal relationships among them. Our plots don't directly address the *joint* dependence of prestige on education and income. Graphs for this purpose will be discussed in [Section 1.5.3](#).

Following Duncan, we next fit a linear least-squares regression to the data to model the joint dependence of prestige on the two predictors, under the assumption that the relationship of prestige to education and income is additive and linear:

```
(duncan.model <- lm(prestige ~ education + income, data=Duncan))
```

Call:

```
lm(formula = prestige ~ education + income, data = Duncan)
```

Coefficients:

| (Intercept) | education | income |
|-------------|-----------|--------|
| -6.065      | 0.546     | 0.599  |

Like the `scatterplotMatrix()` function, the `lm()` (linear model) function uses a *formula* to specify the variables in the regression model, and the `data` argument to tell the function where to find these variables. The `formula` argument to `lm()`, however, has two sides, with the response variable `prestige` on the left of the tilde (`~`). The right-hand side of the model formula specifies the predictor variables in the regression, `education` and `income`. We read the formula as “`prestige` is regressed on `education` and `income`.”<sup>46</sup>

[46](#) You'll find much more information about linear-model formulas in R in [Chapter 4](#), particularly [Section 4.9.1](#).

The lm () function returns a *linear-model object*, which we assign to duncan.model. The name of this object is arbitrary—any valid R name would do. Enclosing the command in parentheses causes the assigned object to be printed, in this instance displaying a brief report of the results of the regression. The summary () function produces a more complete report:

```
summary(duncan.model)

Call:
lm(formula = prestige ~ education + income, data = Duncan)

Residuals:
    Min      1Q  Median      3Q     Max 
-29.54   -6.42    0.65    6.61   34.64 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -6.0647    4.2719   -1.42    0.16    
education    0.5458    0.0983    5.56  1.7e-06 ***  
income        0.5987    0.1197    5.00  1.1e-05 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13.4 on 42 degrees of freedom
Multiple R-squared:  0.828,    Adjusted R-squared:  0.82 
F-statistic: 101 on 2 and 42 DF,  p-value: <2e-16
```

Both education and income have large regression coefficients in the Estimate column of the coefficient table, with small two-sided *p*-values in the column labeled Pr (>|t|). For example, holding education constant, a 1% increase in higher income earners is associated on average with an increase of about 0.6% in high prestige ratings.

R writes very small and very large numbers in scientific notation. For example, 1.1e-05 is to be read as  $1.1 \times 10^{-5}$ , or 0.000011, and 2e-16 =  $2 \times 10^{-16}$ , which is

effectively zero.

If you find the “statistical-significance” asterisks that R prints to the right of the  $p$ -values annoying, as we do, you can suppress them, as we will in the remainder of the *R Companion*, by entering the command:<sup>47</sup>

***options (show.signif.stars=FALSE)***

[47](#) It’s convenient to put this command in your .Rprofile file so that it’s executed at the start of each R session, as we explained in the Preface (page xxii). More generally, the options () function can be used to set a variety of global options in R; see ?options for details.

Linear models are described in much more detail in [Chapter 4](#).

### 1.5.3 Regression Diagnostics

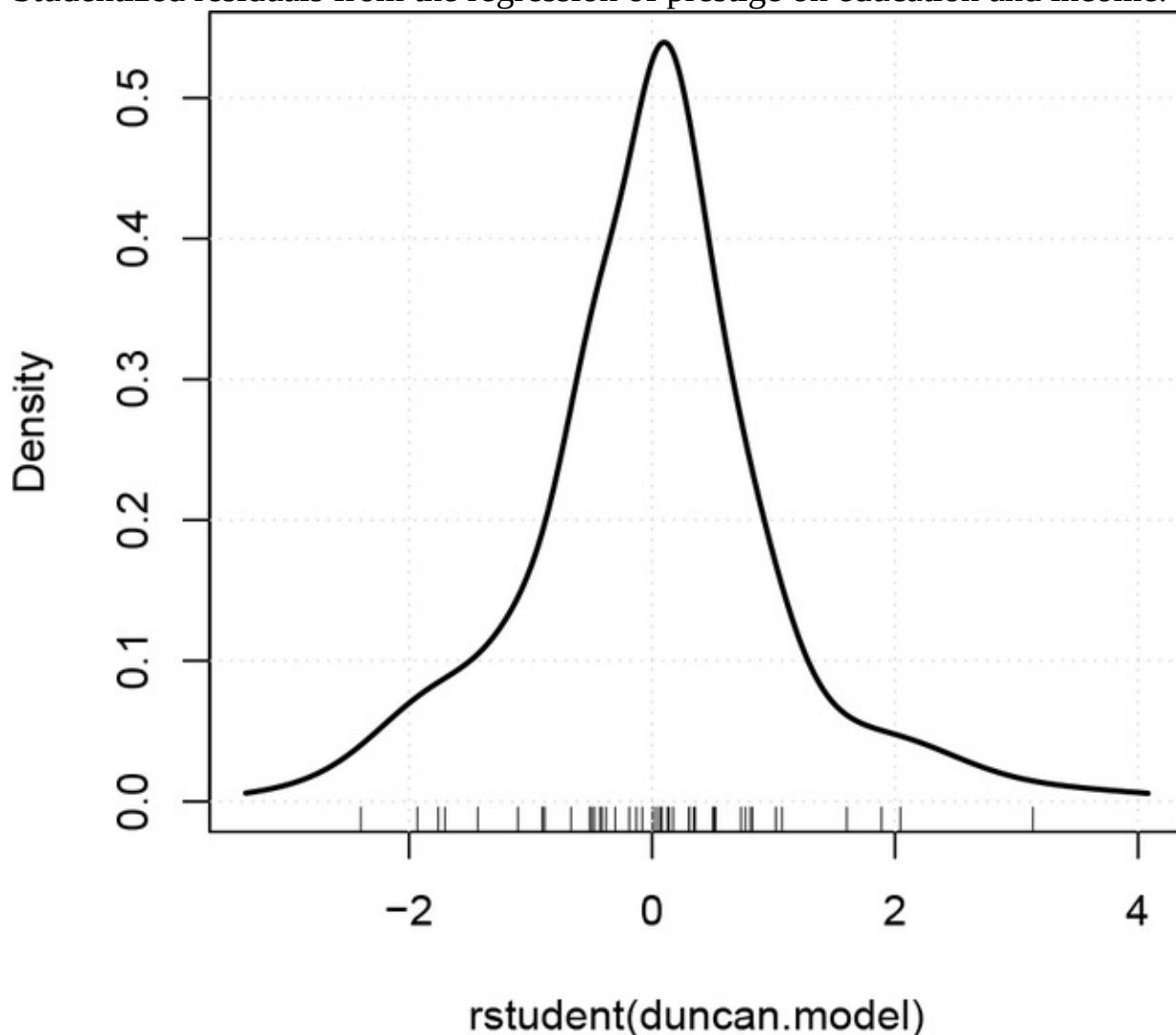
To assume that Duncan’s regression in the previous section adequately summarizes the data does not make it so. It is therefore wise after fitting a regression model to check the fit using a variety of graphs and numeric procedures. The standard R distribution includes some facilities for *regression diagnostics*, and the **car** package substantially augments these capabilities.

The "lm" object duncan.model contains information about the fitted regression, and so we can employ the object in further computations beyond producing a printed summary of the model. More generally, the ability to manipulate statistical models as objects is a strength of R. The rstudent () function, for example, uses some of the information in duncan.model to calculate *Studentized residuals* for the model. A nonparametric density estimate of the distribution of Studentized residuals, produced by the densityPlot () function in the **car** package and shown in [Figure 1.11](#), is unremarkable:

***densityPlot (rstudent (duncan.model))***

**Figure 1.11** Nonparametric density estimate for the distribution of the

Studentized residuals from the regression of prestige on education and income.



Observe the sequence of operations here: `rstudent()` takes the linear-model object `duncan.model`, previously computed by `lm()`, as an argument, and returns the Studentized residuals as a result, then passes the residuals as an argument to `densityPlot()`, which draws the graph. This style of command, where the result of one function becomes an argument to another function, is common in R.

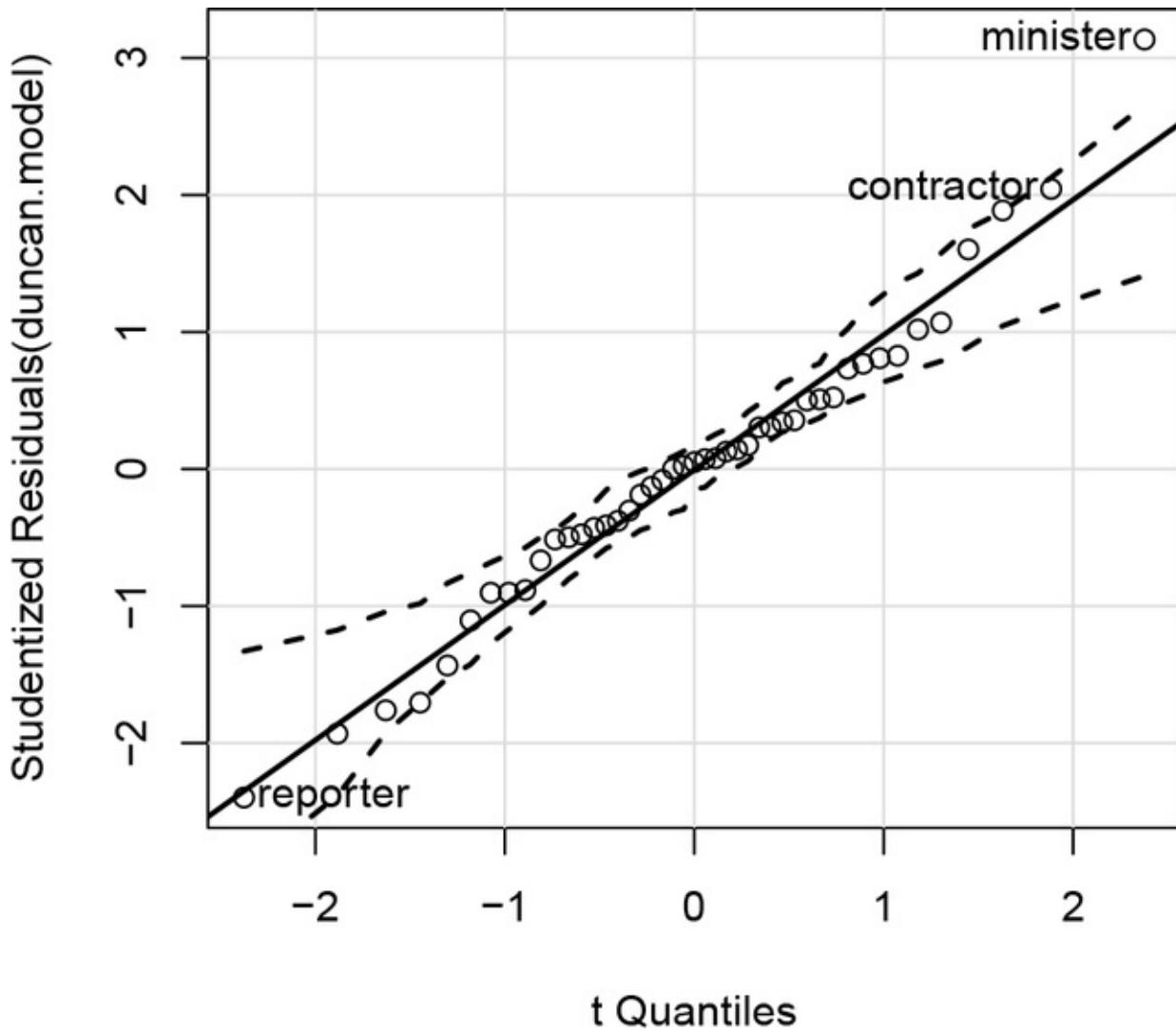
If the errors in the regression are normally distributed with zero means and constant variance, then the Studentized residuals are each  $t$ -distributed with  $n - k - 2$  degrees of freedom, where  $k$  is the number of coefficients in the model, excluding the regression constant, and  $n$  is the number of cases. The generic `qqPlot()` function from the `car` package, which makes *quantile-comparison plots*, has a method for linear models:

```
qqPlot(duncan.model, id=list(n=3))
```

|          |          |            |
|----------|----------|------------|
| minister | reporter | contractor |
| 6        | 9        | 17         |

The resulting quantile-comparison plot is shown in [Figure 1.12](#). The qqPlot () function extracts the Studentized residuals and plots them against the quantiles of the appropriate  $t$ -distribution. If the Studentized residuals are  $t$ -distributed, then the plotted points should lie close to a straight line. The solid comparison line on the plot is drawn by default by robust regression. The argument id=list (n=3) identifies the three most extreme Studentized residuals, and qqPlot () returns the names and row numbers of these cases.

**Figure 1.12** Quantile-comparison plot for the Studentized residuals from the regression of prestige on education and income. The broken lines show a bootstrapped pointwise 95% confidence envelope for the points.



In this case, the residuals pull away slightly from the comparison line at both ends, suggesting that the residual distribution is a bit heavy-tailed. This effect is more pronounced at the upper end of the distribution, indicating a slight positive skew.

By default, `qqPlot()` also produces a bootstrapped pointwise 95% confidence envelope for the Studentized residuals that takes account of the correlations among them (but, because the envelope is computed pointwise, does not adjust for simultaneous inference). The residuals stay nearly within the boundaries of the envelope at both ends of the distribution, with the exception of the occupation minister.<sup>48</sup> A test based on the largest (absolute) Studentized residual, using the `outlierTest()` function in the `car` package, however, suggests that the residual for ministers is not terribly unusual, with a Bonferroni-corrected *p*-value of 0.14:

```
outlierTest (duncan.model)
```

No Studentized residuals with Bonferroni p < 0.05

Largest |rstudent|:

|          | rstudent | unadjusted p-value | Bonferroni p |
|----------|----------|--------------------|--------------|
| minister | 3.1345   | 0.0031772          | 0.14297      |

[48](#) The bootstrap procedure used by qqPlot () generates random samples, and so the plot that you see when you duplicate this command will not be identical to the graph shown in the text.

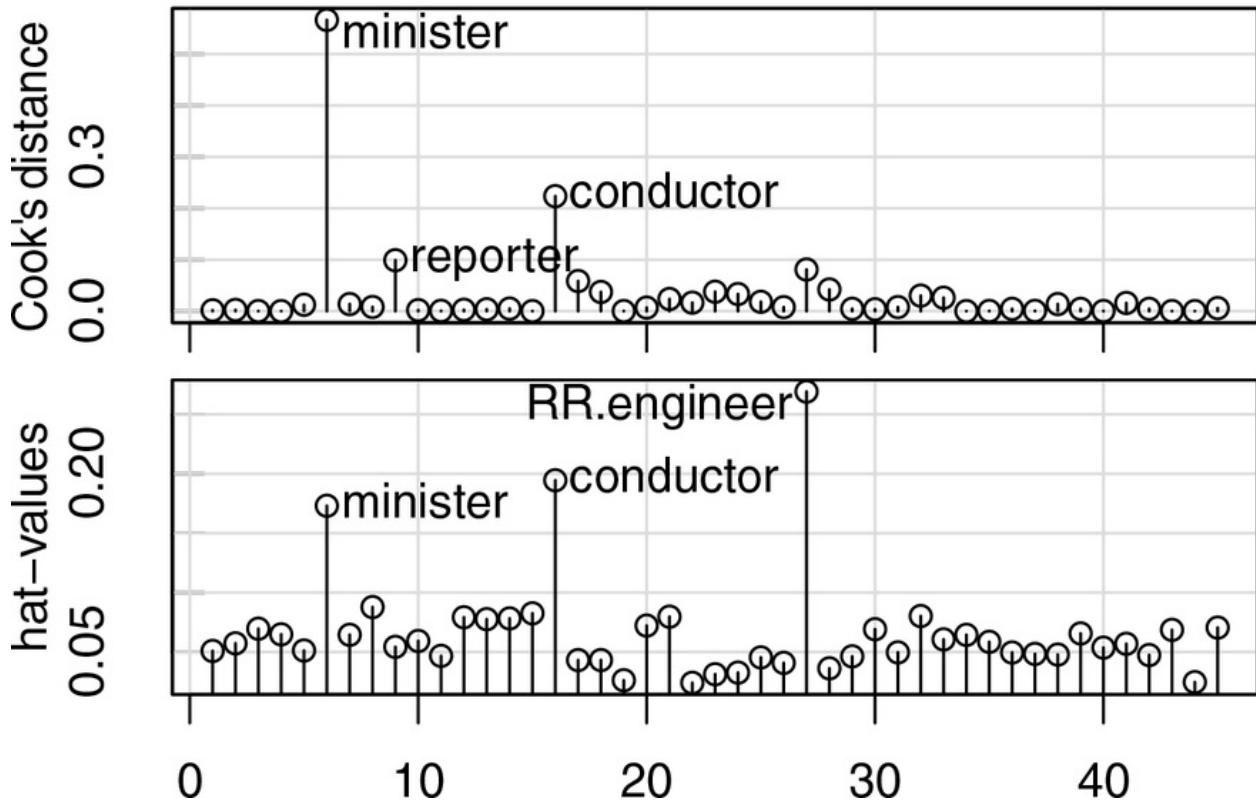
We proceed to check for high-leverage and influential cases by using the influenceIndexPlot () function from the **car** package to plot *hat-values* ([Section 8.3.2](#)) and *Cook's distances* ([Section 8.3.3](#)) against the case indices:

```
influenceIndexPlot (duncan.model, vars=c ("Cook", "hat"), id=list  
(n=3))
```

The two index plots are shown in [Figure 1.13](#). We ask to identify the three biggest values in each plot.

**Figure 1.13** Index plots of Cook's distances and hat-values, from the regression of prestige on income and education.

## Diagnostic Plots



Because the cases in a regression can be *jointly* as well as individually influential, we also examine *added-variable plots* for the predictors, using the `avPlots()` function in the **car** package ([Section 8.2.3](#)):

```
avPlots(duncan.model, id=list(cex=0.75, n=3, method="mahal"))
```

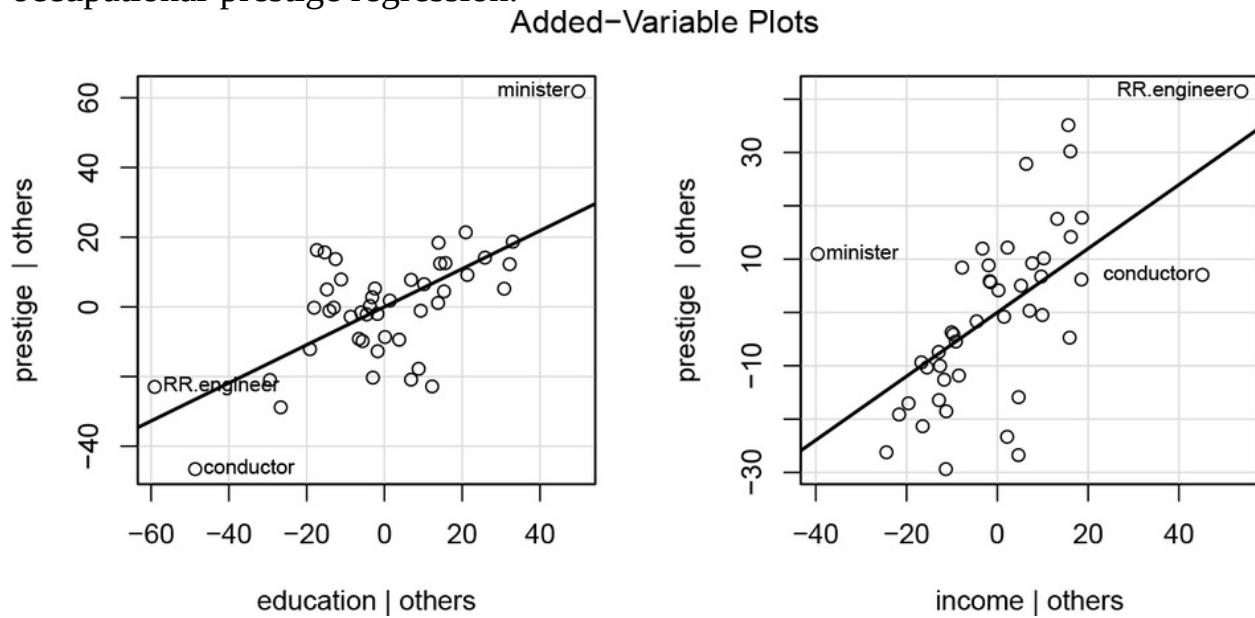
The `id` argument, which has several components here, customizes identification of points in the graph:<sup>49</sup> `cex=0.75` (where `cex` is a standard R argument for “character expansion”) makes the labels smaller, so that they fit better into the plots; `n=3` identifies the three most unusual points in each plot; and `method="mahal"` indicates that unusualness is quantified by Mahalanobis distance from the center of the point-cloud.<sup>50</sup>

<sup>49</sup> See [Section 3.5](#) for a general discussion of point identification in **car**-package plotting functions.

[50](#) Mahalanobis distances from the center of the data take account of the standard deviations of the variables and the correlation between them.

Each added-variable plot displays the *conditional*, rather than the marginal, relationship between the response and one of the predictors. Points at the extreme left or right of the plot correspond to cases that have high leverage on the corresponding coefficient and consequently are potentially influential. [Figure 1.14](#) confirms and strengthens our previous observations: We should be concerned about the occupations minister and conductor, which work jointly to increase the education coefficient and decrease the income coefficient. Occupation RR.engineer has relatively high leverage on these coefficients but is more in line with the rest of the data.

**Figure 1.14** Added-variable plots for education and income in Duncan's occupational-prestige regression.



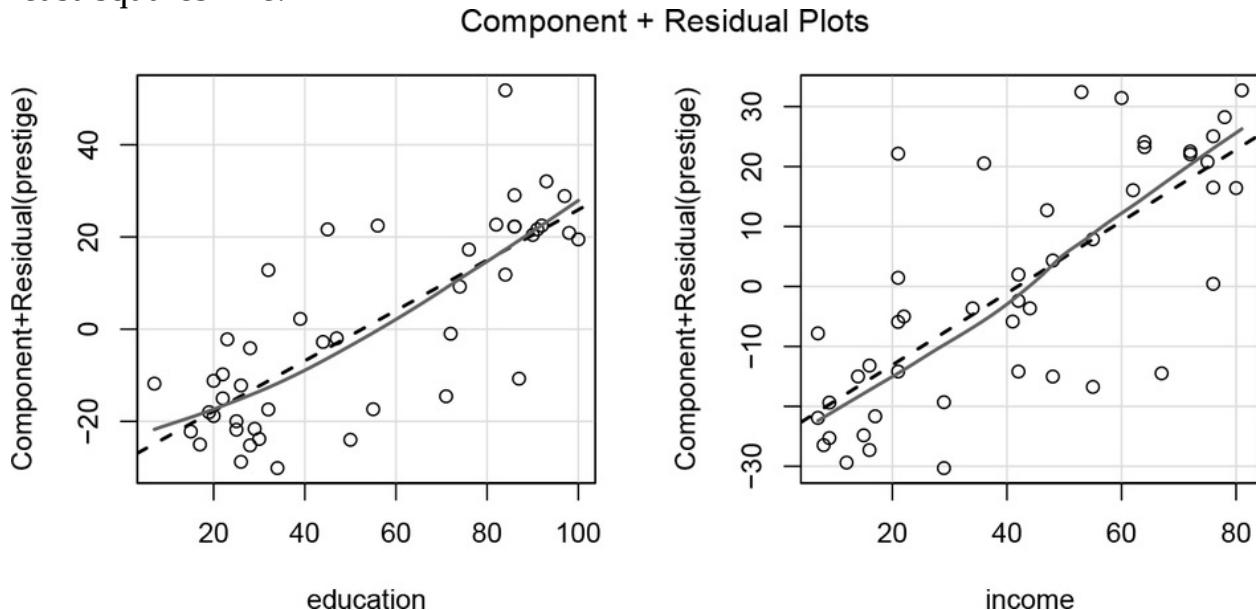
We next use the `crPlots()` function, also in the `car` package, to generate *component-plus-residual plots* for education and income (as discussed in [Section 8.4.2](#)):

### ***crPlots (duncan.model)***

The component-plus-residual plots appear in [Figure 1.15](#). Each plot includes a least-squares line, representing the regression plane viewed edge-on in the

direction of the corresponding predictor, and a *loess* nonparametric-regression smooth.<sup>51</sup> The purpose of these plots is to detect nonlinearity, evidence of which is slight here.

**Figure 1.15** Component-plus-residual plots for education and income in Duncan's occupational-prestige regression. The solid line in each panel shows a loess nonparametric-regression smooth; the broken line in each panel is the least-squares line.



[51](#) See [Section 3.2](#) for an explanation of scatterplot smoothing in the **car** package.

Using the `ncvTest()` function in the **car** package ([Section 8.5.1](#)), we compute score tests for nonconstant variance, checking for an association of residual variability with the fitted values and with *any* linear combination of the predictors:

```
ncvTest(duncan.model)
```

Non-constant Variance Score Test  
Variance formula: ~ fitted.values  
Chisquare = 0.3811, Df = 1, p = 0.537

```
ncvTest(duncan.model, var.formula= ~ income + education)
```

Non-constant Variance Score Test  
Variance formula: ~ income + education  
Chisquare = 0.6976, Df = 2, p = 0.706

Both tests yield large *p*-values, indicating that the assumption of constant variance is tenable.

Finally, on the basis of the influential-data diagnostics, we try removing the cases minister and conductor from the regression:

```
whichNames(c("minister", "conductor"), Duncan)
```

|          |           |
|----------|-----------|
| minister | conductor |
| 6        | 16        |

```
duncan.model.2 <- update(duncan.model, subset=-c(6, 16))  
summary(duncan.model.2)
```

Call:

```
lm(formula = prestige ~ education + income, data = Duncan,  
subset = -c(6, 16))
```

Residuals:

| Min    | 1Q    | Median | 3Q   | Max   |
|--------|-------|--------|------|-------|
| -28.61 | -5.90 | 1.94   | 5.62 | 21.55 |

Coefficients:

|                | Estimate | Std. Error | t value  | Pr(> t )    |         |   |
|----------------|----------|------------|----------|-------------|---------|---|
| (Intercept)    | -6.4090  | 3.6526     | -1.75    | 0.0870 .    |         |   |
| education      | 0.3322   | 0.0987     | 3.36     | 0.0017 **   |         |   |
| income         | 0.8674   | 0.1220     | 7.11     | 1.3e-08 *** |         |   |
| ---            |          |            |          |             |         |   |
| Signif. codes: | 0 '***'  | 0.001 '**' | 0.01 '*' | 0.05 '.'    | 0.1 ' ' | 1 |

Residual standard error: 11.4 on 40 degrees of freedom

Multiple R-squared: 0.876, Adjusted R-squared: 0.87

F-statistic: 141 on 2 and 40 DF, p-value: <2e-16

Rather than respecifying the regression model from scratch with `lm()`, we refit it using the `update()` function, removing the two potentially problematic cases via the `subset` argument to `update()`. We use the `whichNames()` function from the **car** package to remind us of the indices of the two cases to be removed, minister (Case 6) and conductor (Case 16).

The `compareCoefs()` function, also from the **car** package, is convenient for comparing the estimated coefficients and their standard errors across the two regressions fit to the data:

```
compareCoefs(duncan.model, duncan.model.2)
```

Calls:

```
1: lm(formula = prestige ~ education + income, data =  
Duncan)  
2: lm(formula = prestige ~ education + income, data =  
Duncan, subset = -c(6, 16))
```

|             | Model 1 | Model 2 |
|-------------|---------|---------|
| (Intercept) | -6.06   | -6.41   |
| SE          | 4.27    | 3.65    |
| education   | 0.5458  | 0.3322  |
| SE          | 0.0983  | 0.0987  |
| income      | 0.599   | 0.867   |
| SE          | 0.120   | 0.122   |

The coefficients of education and income changed substantially with the deletion of the occupations minister and conductor. The education coefficient is considerably smaller and the income coefficient considerably larger than before. Further work (not shown, but which we invite the reader to duplicate) suggests that removing occupations RR.engineer (Case 27) and reporter (Case 9) does not make much of a difference to the results.

[Chapter 8](#) has much more extensive information on regression diagnostics in R, including the use of various functions in the **car** package.

## 1.6 R Functions for Basic Statistics

The focus of the *R Companion* is on using R for regression analysis, broadly construed. In the course of developing this subject, we will encounter, and indeed already have encountered, a number of R functions for basic statistical methods (`mean()`, `hist()`, etc.), but the topic is not addressed systematically.

[Table 1.1](#) shows the names of some standard R functions for basic data analysis.

The R help system, through ? or help (), provides information on the usage of these functions. Where there is a substantial discussion of a function in a later chapter in the *R Companion*, the location of the discussion is indicated in the column of the table marked *Reference*. The table is not meant to be complete.

**Table 1.1***R Companion*

| <i>Method</i>   | <i>R Function(s)</i>                                 | <i>Reference</i> |
|---|--|------------------|
| <i>Basic Graphs</i>   |  |                  |
| histogram   | hist()   | Chapter 3        |
| stem-and-leaf display   | stem()   | Chapter 3        |
| boxplot   | boxplot()  | Chapter 3        |
| scatterplot   | plot()   | Chapter 3        |
| time-series plot  | ts.plot()  |                  |
| <i>Numerical Summaries</i>  |  |                  |
| mean  | mean()   |                  |
| median  | median()   |                  |
| quantiles   | quantile()   |                  |
| extremes  | range()  |                  |
| variance  | var()  |                  |
| standard deviation  | sd()   |                  |
| covariance matrix   | var(), cov()   |                  |
| correlations  | cor()  |                  |
| <i>Probability</i>  |  |                  |
| normal density, distribution, quantiles, and random numbers         | dnorm(), pnorm(), qnorm(), rnorm()                   | Chapter 3        |
| <i>t</i> density, distribution, quantiles, and random numbers       | dt(), pt(), qt(), rt()                               | Chapter 3        |
| chi-square density, distribution, quantiles, and random numbers     | dchisq(), pchisq(), qchisq(), rchisq()               | Chapter 3        |
| <i>F</i> density, distribution, quantiles, and random numbers       | df(), pf(), qf(), rf()                               | Chapter 3        |
| binomial probabilities, distribution, quantiles, and random numbers | dbinom(), pbinom(), qbinom(), rbinom()               | Chapter 3        |
| generating random samples   | sample(), rnorm(), etc.                              |                  |
| <i>Basic Linear Models</i>  |  |                  |
| simple regression   | lm()   | Chapter 4        |
| multiple regression   | lm()   | Chapter 4        |
| analysis of variance  | aov(), lm(), anova()                                 | Chapter 4        |
| <i>Contingency Tables</i>   |  |                  |
| contingency tables  | xtabs(), table()                                     | Chapter 6        |
| printing tables   | ftable()   | Chapter 6        |
| percentage tables   | prop.table()   | Chapter 6        |
| <i>Simple Hypothesis Tests</i>                                      |  |                  |
| <i>t</i> -tests for means   | t.test()   |                  |
| tests for proportions   | prop.test(), binom.test()                            |                  |
| chi-square test for independence                                    | chisq.test()   | Chapter 6        |
| various nonparametric tests   | friedman.test(), kruskal.test(), wilcox.test(), etc. |                  |

## 1.7 Generic Functions and Their Methods\*

Many of the most commonly used functions in R, such as `summary()`, `print()`, and `plot()`, produce different results depending on the arguments passed to the function.<sup>52</sup> For example, the `summary()` function applied to different columns of the Duncan data frame produces different output. The summary for the variable type is the count in each level of this factor,<sup>53</sup>

```
summary(Duncan$type)
```

|    |      |    |
|----|------|----|
| bc | prof | wc |
| 21 | 18   | 6  |

<sup>52</sup> The generic `print()` function is invoked implicitly and automatically when an object is printed by typing the name of the object at the R command prompt or in the event that the object returned by a function isn't assigned to a variable. The `print()` function can also be called explicitly, however.

<sup>53</sup> `Duncan$type` selects the variable type from the Duncan data frame. Indexing data frames and other kinds of objects is discussed in detail in [Section 2.4.4](#).

while for a numeric variable, such as `prestige`, the summary includes the mean, minimum, maximum, and several quantiles:

```
summary(Duncan$prestige)
```

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max. |
|------|---------|--------|------|---------|------|
| 3.0  | 16.0    | 41.0   | 47.7 | 81.0    | 97.0 |

Similarly, the commands

**summary(Duncan)**

| type    | income       | education     | prestige     |
|---------|--------------|---------------|--------------|
| bc :21  | Min. : 7.0   | Min. : 7.0    | Min. : 3.0   |
| prof:18 | 1st Qu.:21.0 | 1st Qu.: 26.0 | 1st Qu.:16.0 |
| wc : 6  | Median :42.0 | Median : 45.0 | Median :41.0 |
|         | Mean :41.9   | Mean : 52.6   | Mean :47.7   |
|         | 3rd Qu.:64.0 | 3rd Qu.: 84.0 | 3rd Qu.:81.0 |
|         | Max. :81.0   | Max. :100.0   | Max. :97.0   |

and

**summary(lm(prestige ~ education + income, data=Duncan))**

Call:

lm(formula = prestige ~ education + income, data = Duncan)

Residuals:

| Min    | 1Q    | Median | 3Q   | Max   |
|--------|-------|--------|------|-------|
| -29.54 | -6.42 | 0.65   | 6.61 | 34.64 |

Coefficients:

|                | Estimate | Std. Error | t value | Pr(> t )    |
|----------------|----------|------------|---------|-------------|
| (Intercept)    | -6.0647  | 4.2719     | -1.42   | 0.16        |
| education      | 0.5458   | 0.0983     | 5.56    | 1.7e-06 *** |
| income         | 0.5987   | 0.1197     | 5.00    | 1.1e-05 *** |
| ---            |          |            |         |             |
| Signif. codes: | 0 ****   | 0.001 **   | 0.01 *  | 0.05 .      |
|                | 0.1      | '          | '       | 1           |

Residual standard error: 13.4 on 42 degrees of freedom

Multiple R-squared: 0.828, Adjusted R-squared: 0.82

F-statistic: 101 on 2 and 42 DF, p-value: <2e-16

produce output appropriate to these objects—in the first case by summarizing each column of the Duncan data frame and in the second by returning a standard

summary for a linear regression model.

Enabling the same *generic function*, such as `summary()`, to be used for many purposes is accomplished in R through an *object-oriented programming* technique called *object dispatch*. The details of object dispatch are implemented differently in the S3 and S4 object systems, so named because they originated in Versions 3 and 4, respectively, of the original S language on which R is based. There is yet another implementation of object dispatch in R for the more recently introduced system of *reference classes* (sometimes colloquially termed “R5”).

Almost everything created in R is an *object*, such as a numeric vector, a matrix, a data frame, a linear regression model, and so on.<sup>54</sup> In the S3 object system, described in this section and used for most R object-oriented programs, each object is assigned a *class*, and it is the class of the object that determines how generic functions process the object. We won’t take up the S4 and reference-class object systems in this book, but they too are class based and implement (albeit more complex) versions of object dispatch.

<sup>54</sup> Indeed, everything in R that is *returned* by a function is an object, but some functions have *side effects* that create nonobjects, such as files and graphs.

The `class()` function returns the class of an object:

```
class (Duncan$type)
```

```
[1] "factor"
```

```
class (Duncan$prestige)
```

```
[1] "integer"
```

```
class (Duncan)
```

```
[1] "data.frame"
```

The `lm()` function, to take another example, creates an object of class "lm":

```
duncan.model <- lm (prestige ~ education + income, data=Duncan) class
```

## **(duncan.model)**

```
[1] "lm"
```

Generic functions operate on their arguments indirectly by calling specialized functions, referred to as *method functions* or, more compactly, as *methods*. Which method is invoked typically depends on the class of the first argument to the generic function.<sup>55</sup>

[55](#) In contrast, in the S4 object system, method dispatch can depend on the classes of more than one argument to a generic function.

For example, the generic summary () function has the following definition:

### **summary**

```
function (object, ...)  
  UseMethod ("summary")  
  
<bytecode: 0x000000001d03ba78>  
  
<environment: namespace:base>
```

As for any object, we print the definition of the summary () function by typing its name (without the parentheses that would *invoke* rather than *print* the function). The generic function summary () has one required argument, object, and the special argument ... (the ellipses) for additional arguments that could vary from one summary () method to another.<sup>56</sup>

[56](#) You can disregard the last two lines of the output, which indicate that the function has been compiled into *byte code* to improve its efficiency, something that R does automatically, and that it resides in the namespace of the **base** package, one of the standard R packages that are loaded at the start of each session.

When UseMethod ("summary") is called by the summary () generic, and the first

(object) argument to summary () is of class "lm", for example, R searches for a method function named summary.lm (), and, if it is found, executes the command summary.lm (object, ...). It is, incidentally, perfectly possible to call summary.lm () directly; thus, the following two commands are equivalent (as the reader can verify):

**summary (duncan.model)**

**summary.lm (duncan.model)**

Although the generic summary () function has only one explicit argument, the method function summary.lm () has additional arguments:

**args ("summary.lm")**

```
function (object, correlation = FALSE, symbolic.cor = FALSE,
```

```
...)
```

```
NULL
```

Because the arguments correlation and symbolic.cor have default values (FALSE, in both cases), they need not be specified. Thus, for example, if we enter the command summary (duncan.model, correlation=TRUE), the argument correlation=TRUE is absorbed by ... in the call to the generic summary () function and then passed to the summary.lm () method, causing it to print a correlation matrix for the coefficient estimates.

In this instance, we can call summary.lm () directly, but most method functions are hidden in (not “exported from”) the *namespaces* of the packages in which the methods are defined and thus cannot normally be used directly.<sup>57</sup> In any event, it is good R form to use method functions only indirectly through their generics.

<sup>57</sup> For example, the summary () method summary.boot (), for summarizing the results of bootstrap resampling (see [Section 5.1.3](#)), is hidden in the namespace of the **car** package. To call this function directly to summarize an object of class

"boot", we could reference the function with the unintuitive package-qualified name car:::summary.boot (), but calling the unqualified method summary.boot () directly won't work.

Suppose that we invoke the hypothetical generic function fun (), defined as

```
fun <- function (x, ...){  
  UseMethod ("fun")  
}
```

with real argument obj of class "cls": fun (obj). If there is no method function named fun.cls (), then R looks for a method named fun.default (). For example, objects belonging to classes without summary () methods are summarized by summary.default (). If, under these circumstances, there is *no* method named fun.default (), then R reports an error.

We can get a listing of all currently accessible methods for the generic summary () function using the methods () function, with hidden methods flagged by asterisks:<sup>58</sup>

#### **methods (summary)**

```
[1] summary,ANY-method           summary,diagonalMatrix-method  
[3] summary,sparseMatrix-method  summary.Anova.mlm*  
[5] summary.aov                 summary.aovlist*  
[7] summary.aspell*             summary.boot*  
.  
.  
.  
[97] summary.varFunc*           summary.varIdent*  
[99] summary.varPower*  
see '?methods' for accessing help and source code
```

<sup>58</sup> The first three method functions shown, with commas in their names, are S4 methods.

These methods may have different arguments beyond the first, and some method functions, for example, summary.lm (), have their own help pages: ?

`summary.lm`.

You can also determine what generic functions have currently available methods for objects of a particular class. For example,

```
methods(class="lm")  
[1] add1                  alias      anova  
[4] Anova                 avPlot     Boot  
[7] bootCase              boxCox    case.names  
[10] boxcox                drop1     drop1  
[13] drop1                 drop1     drop1  
[16] drop1                 drop1     drop1  
[19] drop1                 drop1     drop1  
[22] drop1                 drop1     drop1  
[25] drop1                 drop1     drop1  
[28] drop1                 drop1     drop1  
[31] drop1                 drop1     drop1  
[34] drop1                 drop1     drop1  
[37] drop1                 drop1     drop1  
[40] drop1                 drop1     drop1  
[43] drop1                 drop1     drop1  
[46] drop1                 drop1     drop1  
[49] drop1                 drop1     drop1  
[52] drop1                 drop1     drop1  
[55] drop1                 drop1     drop1  
[58] drop1                 drop1     drop1  
[61] drop1                 drop1     drop1  
[64] drop1                 drop1     drop1  
[67] drop1                 drop1     drop1  
[70] drop1                 drop1     drop1  
see '?methods' for accessing help and source code
```

Method selection is slightly more complicated for objects whose class is a vector of more than one element. Consider, for example, an object returned by the `glm()` function for fitting generalized linear models (anticipating a logistic-regression example developed in [Section 6.3.1](#)):<sup>59</sup>

```
mod.mroz <- glm (lfp ~ ., family=binomial, data=Mroz) class (mod.mroz)  
[1] "glm" "lm"
```

[59](#) The `.` on the right-hand side of the model formula indicates that the response variable `lfp` is to be regressed on *all* of the other variables in the `Mroz` data set (which is accessible because it resides in the `carData` package).

If we invoke a generic function with `mod.mroz` as its argument, say `fun(mod.mroz)`, then the R interpreter will look first for a method named `fun.glm()`. If a function by this name does not exist, then R will search next for `fun.lm()` and finally for `fun.default()`. We say that the object `mod.mroz` is of *primary class* "glm" and *inherits* from class "lm".<sup>60</sup> Inheritance supports economical programming through generalization.<sup>61</sup>

[60](#) If the class vector of an object has more than two elements, then the classes are searched sequentially from left to right.

[61](#) S3 inheritance can also get us into trouble if, for example, there is no function `fun.glm ()` but `fun.lm ()` exists and is inappropriate for `mod.mroz`. In a case such as this, the programmer of `fun.lm ()` should be careful also to create a function `fun.glm ()`, which calls the default method or reports an error, as appropriate.

## 2 Reading and Manipulating Data

John Fox & Sanford Weisberg

The statistical methods covered in the *R Companion* use data that can be represented as a rectangular array, with rows corresponding to *cases* or *observations* and columns corresponding to *variables*. The basic data structure for a rectangular data set in R is a *data frame*. Many R packages include data frames that illustrate various statistical methods, as we demonstrated in [Section 1.5](#) with the Duncan data set from the **carData** package. When you work with your own data, you generally first need to read the data into R from an external source such as a file, creating a data frame and, much more often than not, modifying the data to put them in a form suitable for analysis. These are the topics taken up in this chapter.

In the *R Companion* and the **carData** package, we adopt naming conventions for data frames and variables that are neither required nor standard in R but that, we believe, generally make our R code clearer: We name data frames, like *Duncan* or *Mroz*, two data sets residing in the **carData** package, beginning with an uppercase letter. Variables, such as the variables *prestige* and *income* in the *Duncan* data frame, have names beginning with a lowercase letter. More generally, you may use any valid R name for a data frame or for a variable within a data frame.

- [Section 2.1](#) shows you how to read data into an R data frame from a plain-text file, from a spreadsheet file, and from other sources.
- [Section 2.2](#) briefly describes alternative approaches to reading and organizing rectangular data sets, focusing on the R packages in the “tidyverse” (Wickham & Grolemund, 2016).
- [Section 2.3](#) explains how to work with data stored in data frames.
- [Section 2.4](#) introduces matrices, higher-dimensional arrays, and lists, three data structures that are commonly used in R.
- [Section 2.5](#) shows you how to work with date and time data in R.
- [Section 2.6](#) explains how to manipulate character data in R, temporarily taking the focus off data in rectangular data sets.
- Finally, [Section 2.7](#) discusses how to handle large data sets in R.

We'd be less than honest if we failed to admit that data management is a dry and potentially tedious subject but one that is necessary to master to perform practical statistical data analysis. It's common to spend much more time managing data in a research project than analyzing them. We suggest that you read as much of this chapter as necessary to get going with your work and that you scan the rest of the chapter so that you can later treat it as a resource when

you encounter a data management problem.

## 2.1 Data Input

In this section, we provide information about several functions for reading data into R. Data to be read into R come in many forms, including data sets in R packages, data in plain-text files, data in specially formatted R files, and data imported from spreadsheets or from other statistical or database software.

### 2.1.1 Accessing Data From a Package

Many R packages include data frames, consequently making data sets easily accessible. For example, when you load the **car** package via the `library()` function, the **carData** package is also loaded.

```
library ("car") # loads car and carData packages
```

Loading required package: carData

The **carData** package includes more than 50 data frames. You can see their names by entering the command `help(package="carData")`.<sup>1</sup> One of the data frames in the **carData** package is named Davis:

```
class(Davis)
```

```
[1] "data.frame"
```

```
brief(Davis)
```

```
200 x 5 data.frame (195 rows omitted)
  sex weight height repwt rephgt
  [f]   [i]     [i]    [i]    [i]
  1   M      77     182     77    180
  2   F      58     161     51    159
  3   F      53     161     54    158
  ...
  199 M      90     181     91    178
  200 M      79     177     81    178
```

<sup>1</sup> A principal motivation for separating the data sets in the **carData** package from the functions in the **car** package was to make the help pages for data sets and functions easier to navigate.

Like all of the data sets in **carData**, Davis is a data frame and hence of class "data.frame". The brief () function in the **car** package when applied to a data frame by default prints the first few and last few rows of the data frame and reports the number of rows and columns. If there are many columns in the data frame, some columns may also be elided.

Davis contains five variables, each represented as a column in the data frame, with names sex, weight, height, repwt, and rept. Also shown in the output from brief () are one-character abbreviations of the classes of the variables: The variable sex is a factor, indicated by the notation [f] at the head of the column, and the remaining four columns are integer (whole number) variables, indicated by [i]. The unnamed left-most column in the printout, with the numbers 1, 2, 3, ..., 199, 200, contains *row names*, in this case simply row numbers. The row names are not a variable in the data frame and consequently are not counted as a column.<sup>2</sup> The help () function or operator, help ("Davis") or ?Davis, provides a "code-book," with definitions of the variables in the data frame.

<sup>2</sup> If you enter the command rownames (Davis), you'll see that these row numbers are represented as character strings: "1", "2", etc.

You can use the data () function to access a data frame from a package in your library without first loading the package. For example, if you have installed the **alr4** package on your computer but not loaded it with a library () command, you can read the Challeng data frame into the global environment with the command<sup>3</sup>

```
data("Challeng", package="alr4")
brief(Challeng)
```

```
23 x 7 data.frame (18 rows omitted)
  temp pres fail n erosion blowby damage
  [i] [i] [i] [i] [i] [i] [i]
  4/12/81 66 50 0 6 0 0 0
  11/12/81 70 50 1 6 1 0 4
  3/22/82 69 50 0 6 0 0 0
  . . .
  11/26/85 76 200 0 6 0 0 0
  1/12/86 58 200 1 6 1 0 4
```

<sup>3</sup> The Challeng data set contains information about space shuttle launches prior

to the *Challenger* disaster; see `help ("Challeng", package="alr4")` for details. Some packages *require* that you call the `data()` function to use data frames in the package, even when the package is loaded. Other packages, like our **carData** package, use R’s *lazy data* mechanism to provide direct access to data frames when the package is loaded. On a package’s help page, accessed, for example, by `help (package="carData")`, click on the link for the package *DESCRIPTION file*. Data frames are automatically available if the line `LazyData: yes` appears in the package description.

## 2.1.2 Entering a Data Frame Directly

We have already seen in [Sections 1.2.4–1.2.5](#) how small vectors can be entered directly at the R command prompt.<sup>4</sup> We can similarly enter a data frame at the command prompt, but this process is tedious and error prone and consequently is useful only for very small data sets.

[4](#) Although we refer here to entering data “at the R command prompt,” we assume both in this chapter and generally in the *R Companion* that your commands originate in R scripts or R Markdown documents, as explained in [Section 1.4](#). Moreover, as we mentioned in the Preface, we don’t show the command prompt in R input displayed in the text.

Consider the data in [Table 2.1](#), from an experiment by Fox and Guyer (1978), in which each of 20 four-person groups of subjects played 30 trials of a prisoners’ dilemma game in which subjects could make either cooperative or competitive choices. Half the groups were composed of women and half of men. Half the groups of each sex were randomly assigned to a public-choice condition, in which the choices of all the individuals were made known to the group after each trial, and the other groups were assigned to an anonymous-choice condition, in which only the aggregated choices were revealed. The data in the table give the number of cooperative choices made in each group, out of  $30 \times 4 = 120$  choices in all.

**Table 2.1**

| Sex    | Condition     |    |    |           |    |    |
|--------|---------------|----|----|-----------|----|----|
|        | Public Choice |    |    | Anonymous |    |    |
| Male   | 49            | 64 | 37 | 52        | 68 | 27 |
| Female | 54            | 61 | 79 | 64        | 29 | 40 |

[Table 2.1](#) displays the data from the experiment compactly, but it does not

correspond directly to the structure of an R data frame, where rows represent cases, four-person groups in the example, and columns represent variables. There are three variables in this data set, two categorical predictors, condition and sex, and one numeric response, cooperation. A data frame for the Fox and Guyer data should therefore consist of three columns, one for each predictor and one for the response. There are 20 values of cooperation, one for each of the 20 groups, and so the data frame should have 20 rows.

We begin by using the `c()` function to enter the data for cooperation, starting with the five cases in the (Public Choice, Male) experimental condition, then the five cases in the (Public Choice, Female) condition, then the five cases in the (Anonymous, Male) condition, and finally the five cases in the (Anonymous, Female) condition:

```
cooperation <- c(49, 64, 37, 52, 68, 54, 61, 79, 64, 29, 27, 58, 52, 41, 30,  
40, 39, 44, 34, 44)
```

The process of typing in data manually like this is error prone, because of the need to separate data values by commas, to get all the numbers right and in the correct order, and to match the opening and closing parentheses.<sup>5</sup>

<sup>5</sup> A somewhat more forgiving, but still problematic, procedure is to use the `scan()` function to read the data. Enter the command `cooperation <- scan()`, and then type the data values, separated by blanks (spaces), on one or more subsequent lines. When you're done, enter an empty line to terminate the operation. See `help("scan")` for more information.

Because of the order in which the Fox and Guyer data are entered, we can much more easily and confidently create the variables condition and sex, using the `rep()` (repeat) function:

```
(condition <- rep(c("public", "anonymous"), c(10, 10)))  
  
[1] "public"     "public"      "public"      "public"      "public"  
[6] "public"     "public"      "public"      "public"      "public"  
[11] "anonymous"  "anonymous"   "anonymous"   "anonymous"  "anonymous"  
[16] "anonymous"  "anonymous"   "anonymous"   "anonymous"  "anonymous"  
  
(sex <- rep(rep(c("male", "female"), each=5), 2))  
  
[1] "male"       "male"        "male"        "male"        "male"       "female"  
[7] "female"     "female"     "female"     "female"     "male"       "male"  
[13] "male"       "male"        "male"        "female"     "female"     "female"  
[19] "female"     "female"
```

The `rep()` function generates patterned vectors. Its first argument specifies the data to be repeated, and its second argument specifies the number of repetitions:

```
rep (5, 3)
```

```
[1] 5 5 5
```

```
rep (c (1, 2, 3), 2)
```

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

When the first argument to rep () is a vector, the second argument can also be a vector of the same length, specifying the number of times each entry of the first argument is to be repeated:

```
rep (1:3, 3:1)
```

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

The command rep (c ("public", "anonymous"), c (10, 10)) therefore returns a vector of length 20, with elements corresponding to the order in which we entered the data values for cooperation, with the first 10 values in the "public" condition and the last 10 in the "anonymous" condition. An equivalent and more compact command uses the each argument: rep (c ("public", "anonymous"), each=10). We employ each to create the variable sex, using rep () twice, first to generate five "male" character strings followed by five "female" character strings and then to repeat this pattern twice to get all 20 values.

Finally, we call the data.frame () function to combine the three vectors into a data frame:

```
brief(Guyer <- data.frame(cooperation, condition, sex))

20 x 3 data.frame (15 rows omitted)
  cooperation condition     sex
      [n]          [f]     [f]
  1       49 public    male
  2       64 public    male
  3       37 public    male
  . . .
  19      34 anonymous female
  20      44 anonymous female
```

Because we didn't explicitly specify variable names in the call to data.frame (), the function infers variable names from its arguments.

We could alternatively bypass defining the three variables individually and instead specify the data directly, including the variable names as arguments to the data.frame () function:

```

Guyer <- data.frame(
  cooperation = c(49, 64, 37, 52, 68, 54, 61, 79, 64, 29,
                  27, 58, 52, 41, 30, 40, 39, 44, 34, 44),
  condition = rep(c("public", "anonymous"), c(10, 10)),
  sex = rep(rep(c("male", "female"), each=5), 2)
)

```

In either case, the variables condition and sex are entered as character vectors. When character vectors are put into a data frame, they are converted by default into factors, which is almost always appropriate for subsequent statistical analysis of the data.<sup>6</sup> In the brief () output, cooperation is flagged as a numeric variable ([n]), and condition and sex are flagged as factors ([f]).

[6](#) Sometimes, however, we wish to retain a column of a data frame, such as a column containing individuals' names, in character form. We can do this by specifying the argument stringsAsFactors=FALSE in the call to data.frame (); see help ("data.frame").

After the Guyer data frame is created, its name appears in the RStudio *Environment* tab. Clicking on the blue arrow-icon to the left of Guyer produces a description of the data frame similar to typing str (Guyer) at the R command prompt:

```
str(Guyer)
```

```

'data.frame':      20 obs. of  3 variables:
 $ cooperation: num  49 64 37 52 68 54 61 79 64 29 ...
 $ condition   : Factor w/ 2 levels "anonymous","public": 2 2 2 ...
 $ sex         : Factor w/ 2 levels "female","male": 2 2 2 2 2 ...

```

The str () function applied to any R object prints information about the internal structure of the object. For a data frame, str () prints the number of rows (obs., observations) and columns (variables), along with the name, class, and first few values of each column. In the example, Guyer has 20 rows and three columns: cooperation, which is a numeric variable, and condition and sex, which are factors with two levels each. The values printed for a factor are the level numbers; thus the first few cases are in level 2 of condition, which is "public". The same information is conveniently available in the RStudio *Environment* tab, by clicking on the arrow to the left of Guyer.

### 2.1.3 Reading Data From Plain-Text Files

Except for some very small data sets, you'll read data into R from files. *Plain-text* data files can be read both by people and by computer programs such as R or Excel and can be opened and edited in a plain-text editor, such as the editor in

RStudio. We'll consider plain-text data files in two common formats: *white-space-delimited values*, typically saved in files with extension .txt, and *comma-separated values*, typically saved in files with extension .csv. Think of plain-text data files as a lowest common denominator for storing rectangular data sets, as most statistical software, spreadsheet software, and database software can both read and write text data files.

## White-Space-Delimited Data

Viewed in a text editor, a white-space-delimited data file looks more or less like a data frame, except that columns may not line up neatly. Just like a data frame, a white-space-delimited data file has *lines* (rows) corresponding to cases and *fields* (columns) corresponding to variables. The file may include *row names* as the left-most field in each data line and may also include *variable names* in the first line of the file. If row names are present, they may or may not have a corresponding variable name on the first line. Each data line in the file has the same number of fields in the same order: the row name, if present, followed by the values of the variables for a particular case. For example, the first seven lines of the file Duncan.txt, which you downloaded to your *R Companion* project directory in [Section 1.1](#), are

```
type income education prestige
accountant prof 62 86 82
pilot prof 72 76 83
architect prof 75 92 90
author prof 55 90 76
chemist prof 64 86 90 minister prof 21 84 87
```

The first line contains variable names for the four variables in the data file. The succeeding lines each contain a row name followed by the values of the four variables, five fields in all. The fields in each line of the file may be separated by any number of blank spaces or tabs, called *white space*. Because a single space separates the fields in Duncan.txt, the fields do not line up neatly in columns. The `read.table()` function creates R data frames from plain-text files, which, by default, are in white-space-delimited form; for example:

```

Duncan <- read.table(file="Duncan.txt", header=TRUE)
brief(Duncan) # first 3 and last 2 rows

45 x 4 data.frame (40 rows omitted)

  type income education prestige
  [f]   [i]     [i]     [i]
accountant prof     62      86      82
pilot        prof    72      76      83
architect    prof    75      92      90
...
policeman   bc      34      47      41
waiter       bc      8       32      10

```

- The file argument to `read.table()` is the quoted name of the file to read, assuming that the file resides in the current RStudio project directory. If the file is somewhere else, for example, in a subdirectory of the project directory, on a flash drive or cloud drive, somewhere on the internet, or copied to the clipboard, you will need to modify the file argument accordingly (see [Section 2.1.4](#)).
- The argument `header=TRUE` indicates that the first line of the file contains variable names.<sup>7</sup> If you don't have variable names in the first line, set `header=FALSE`, and `read.table()` will supply variable names, or you can enter variable names directly via the optional `col.names` argument; see `help("read.table")`.
- The `read.table()` function automatically determines that row names are present if the number of variable names is one fewer than the number of fields in each data line. Had `Duncan.txt` included a variable name, such as `occupation`, for the first field in each data line, then the appropriate command for inputting the data with proper row names would be

```

Duncan <- read.table ("Duncan.txt", header=TRUE,
  row.names="occupation")

```

- The `read.table()` function automatically and silently converts character data to factors unless you add the argument `stringsAsFactors=FALSE` (or set options (`stringsAsFactors=FALSE`); see `help("options")`). The optional `colClasses` argument to `read.table()` may also be used for finer-grain control over how the fields in a data file are converted into variables (see [Section 2.7](#)).
- In many data sets, the values of some variables aren't known for some

cases, as for example when a respondent to a survey fails to answer a question, or are undefined, as is, for example, the age of a childless respondent's oldest child. In a white-space-delimited data file, these *missing values* must be filled in with a missing-data indicator. The default missing-data indicator in R is NA, which is distinct from lowercase na, quoted "NA", or variations such as N/A and N A. You can change the missing-value indicator to something else, for example, to a period, as is used in SAS and some other programs, via the argument na.strings=". ". The na.strings argument to `read.table()` may also be a character vector of missing-data codes, one for each variable in the data set.

- The `read.table()` function and the other functions for data input that we describe in this chapter have several additional arguments that are used infrequently. The help pages for these functions, such as `help("read.table")`, provide information on the various arguments.

[7](#) The `read.table()` function *may* be able to figure out whether the first row contains variable names, but it is risky to rely on its ability to do so.

It is very common, and potentially very frustrating, for plain-text data files to contain formatting errors that prevent them from being read, or read correctly, by `read.table()`:

- For example, character values in a white-space-delimited data file cannot contain embedded blanks. To illustrate, one of the occupations in the Duncan data set is *coal miner*. Had we entered the row label for this occupation as coal miner, then R would have treated this intended label as two fields, coal and miner, producing an error because the corresponding data line would have too many fields. We solved this problem by entering the occupation label as coal.miner; another solution is to enclose the label in quotes, "coal miner". The `count.fields()` function is useful for localizing this kind of error (see `help("count.fields")`).
- If you create a white-space-delimited .txt file from a spreadsheet, an empty cell in the spreadsheet will produce a missing field in the file. Be careful to put NA (or another missing-data indicator) in missing cells. An alternative solution is to export the data from a spreadsheet or other software as a comma-delimited data file, as discussed below, or to read the spreadsheet file directly (see [Section 2.1.6](#)). You may also experience a similar problem when exporting data from other programs. The `count.fields()` function can help here as well.
- Column names in the first line of the file must be valid variable names in R (see [Section 1.2.4](#)). Otherwise, R will change the names to valid names, or it may get confused if you use special characters like blanks, #, %, or \$ in

an intended variable name, or if the variable names extend over more than one line.

- Numbers may include decimal points and negative signs and may be specified in scientific notation (e.g., -6.6e-5), but they cannot include separators like commas (as in 1,000,000) or modifiers like dollar signs or percent signs (as in \$200 or 41%). Should your data file contain commas, dollar signs, or percent signs, use a text editor to remove them before reading the data into R.
- Typing errors, such as confusing a lowercase letter l (“el”) with the numeral 1 (one), or a uppercase letter O (“oh”) with the numeral 0 (zero), are common and troublesome. As a general matter, any column (i.e., field) in the input data file that includes *even one* nonnumeric value will be converted to a factor by `read.table()`. Consequently, if a numeric variable unexpectedly becomes a factor, suspect a stray nonnumeric character in the corresponding column of the data file.

## Comma-Separated-Values Files

Comma-separated-values or .csv data files are similar to white-space-delimited .txt files, except that commas, rather than white space, are used to separate fields within a line. Variable names, if present, must also be separated by commas. Character fields in a .csv file should not include embedded commas but may include embedded blanks.<sup>8</sup>

<sup>8</sup> In .csv files, as in .txt files, you may enclose a character value in quotes (e.g., "divorced, separated, or widowed"), and then blanks or commas that would otherwise cause an error are permitted.

For example, the first seven lines of the file `Duncan.csv` are

```
type,income,education,prestige  
accountant,prof,62,86,82  
pilot,prof,72,76,83  
architect,prof,75,92,90  
author,prof,55,90,76  
chemist,prof,64,86,90  
minister,prof,21,84,87
```

The first line of the file contains names for the four variables in the data set but no variable name for the row names, which are in the initial field of each subsequent data line.

The function `read.csv()` reads .csv files:

```
Duncan <- read.csv (file = "Duncan.csv")
```

The default value of the `header` argument for `read.csv()` is TRUE, so you do not

need the argument unless the first line of the data file doesn't contain variable names.<sup>9</sup>

<sup>9</sup> The function `read.csv2()` uses semicolons as the default field separator and assumes that commas indicate decimal points in numbers, as is conventional in some European countries and the Canadian province of Québec. The functions `read.csv()` and `read.csv2()` call `read.table()` but have different default arguments.

## More Complex Text Files\*

Data in plain-text files may come in more complex forms:

- Each case in the data set may correspond to more than one line in the data file.
- The values in each data line may not be separated by delimiters such as white space or commas and may instead run together.
- There may even be different numbers of values for different cases.

The `read.fwf()` (read fixed-width format) function can handle data sets with more than one line per case and fields that are not separated by delimiters. The even more flexible `readLines()` function reads each line of the file as a character string, which then can be processed as text (using methods discussed in [Section 2.6](#)). See `help("read.fwf")` and `help("readLines")` for details.

### 2.1.4 Files and Paths

Functions for reading and writing files in R generally have a `file` argument,<sup>10</sup> which specifies the name of a file to be read or written. The `file` argument can simply be the quoted name of a file, but it may specify the path to a file in your computer's file system. The `file` argument also allows you to read data copied to the clipboard or from a *URL* (internet address).

<sup>10</sup> Some functions have a `con` (for connection) argument instead of a `file` argument; both arguments can take *connections*, which include, but are more general than, files and URLs; see `help("connections")`.

R defines a particular directory (or folder) in your file system as its *working directory*. To read a data file from the working directory, or to write data to a new file in the working directory, all you need to do is specify the file name in quotes, as in the examples in [Section 2.1.3](#). When you start RStudio in a project, the working directory is set to the project directory.

The `getwd()` function returns the working directory:

**`getwd()`**

[1] "C:/Users/John Fox/Documents/R-Companion"

This path to the R-Companion-project working directory is for a Windows computer. A peculiarity of R on Windows is that directories in paths are

separated by forward slashes (/), because the backslash (\) character is reserved for other uses in R.<sup>11</sup> You can change your working directory via the RStudio menus, *Session > Set Working Directory*, or the setwd () command.

<sup>11</sup> The backslash is an *escape character*, indicating that the next character has special meaning; for example, "\t" is the tab character; see [Section 2.6](#) on working with character data in R.

To read a data file from the working directory, just specify its name in a read.table () or read.csv () command; for example,

```
Duncan <- read.table ("Duncan.txt", header=TRUE)
```

If the file resides in a directory other than the working directory, you must specify a proper path to the file. Imagine, for example, that Duncan.txt resides in the data subdirectory of our working directory; then either of the following commands, the first specifying a *relative path* to the file (starting at the working directory) and the second specifying the *absolute path*, would be appropriate:

```
Duncan <- read.table ("data/Duncan.txt", header=TRUE)
```

```
Duncan <- read.table (
```

```
  "C:/Users/John Fox/Documents/R-Companion/data/Duncan.txt",
```

```
  header=TRUE
```

```
)
```

If you are working directly in the *Console*, you can use either the function file.choose () or the system's clipboard to provide input to read.table (), but you should not use these functions in an R Markdown document because they require direct interaction with the user. The file.choose () function opens a standard file selection dialog, allowing the user to navigate to the location of a file on the computer's file system. The function returns a character string specifying the path to the selected file, suitable for the file argument of read.table () or other functions, as in

```
Duncan <- read.table (file.choose (), header=TRUE)
```

You can also call read.table () or read.csv () to input data that have been copied to the clipboard. The contents of the clipboard must be in the format expected by the function. For small data sets, this can be a fast and effective way of reading spreadsheet data into a data frame. Use the mouse to select the cells to be read from a spreadsheet, including possibly a row of variable names at the top; copy the selected cells to the clipboard (e.g., by the key-combination Ctrl-c or command-c); and then on Windows enter

```
Data <- read.table ("clipboard", header=TRUE)
```

or, on macOS,

```
Data <- read.table (pipe ("pbpaste"), header=TRUE)
```

This command assumes that an initial row of variable names is included in the

clipboard selection and that you want to create a data frame named Data, but you can omit the header row with header=FALSE or specify a different name for the data set.

Finally, you can also read a data file from an internet URL; for example:<sup>12</sup>

```
Duncan <- read.table (  
  "https://socialsciences.mcmaster.ca/jfox/Books/Companion/data/Duncan.txt"  
  header=TRUE)
```

<sup>12</sup> Because of the long URL, we have broken the character string in this example into two lines to print the command in the book; the string would not normally be broken in this manner.

For some functions, though not read.table (), you may need to use the url () function, as in

```
Duncan <- read.table (  
  url  
  ("https://socialsciences.mcmaster.ca/jfox/Books/Companion/data/Duncan.txt")  
  header=TRUE)
```

## 2.1.5 Exporting or Saving a Data Frame to a File

R data frames written to plain-text files are human-readable and are also easily read back into R or into another program, such as a spreadsheet, a database, or a different statistical program. This is a reasonable strategy for exporting “smaller” data sets of up to a few tens of thousands of cases and perhaps a few hundred variables. For larger data sets to be reread into R—data sets perhaps with a million or more cases and hundreds or even thousands of variables—plain-text files are too large to be of much use. It is preferable to save larger data sets in a compressed internal format that takes much less space to store and much less time to read back into R. For consistency, we prefer to save all data frames, regardless of size, in internal form.

The following two commands use write.table () and write.csv () to export the Duncan data frame as plain-text files, with values separated respectively by blanks and commas:

```
write.table (Duncan, file="Duncan.txt")  
write.csv (Duncan, file="Duncan.csv")
```

Variable names are included by default on the first line of the file and row names in the first field of each subsequent line. A feature of write.csv () is that it includes an empty variable name ("", followed by a comma) in the first field (the row-name field) of the first line (the variable-names line) of the file: This is typically the format that a spreadsheet program expects in a .csv file. To read the resulting file back into R with read.csv (), include the argument row.names=1,

indicating that the first field in each data line contains the row name. By default, character strings, including row names, in the output text file are surrounded by double quotes. You can suppress the double quotes, which may cause problems for some programs other than R, with the argument quote=FALSE, but do so only if you're sure that there are no embedded blanks in character strings when using write.table () or embedded commas when using write.csv (). See [Section 2.1.4](#) for help with the file argument when you want to save a data set in a directory other than the current directory.

To save a data frame to a file in a compressed internal form, use the save () command:

```
save (Duncan, file="Duncan.RData")
```

The save () function is more general than this and can be used to save any object in R, not just a data frame, or to save multiple objects in a single compressed file. The resulting .RData file can be reread into R in a subsequent session or even opened in an R session on a different computer; see help ("save") for details. Files saved in this manner conventionally have the extension .RData, but this is not required.

A .RData file is read back into R via the load () command; for example:

```
load (file="Duncan.RData")
```

The load () function restores all objects saved in the .RData file to your current session.<sup>13</sup> In addition to efficiency, which is a consideration for large data sets (see [Section 2.7](#)), using save () and load () allows you to conveniently save your data after you've invested some time in data management tasks, such as cleaning the data and transforming variables in the data set, without having to re-execute the data management commands in subsequent R sessions.

<sup>13</sup> This characteristic of save ()/load ()—to restore objects to the global environment by side effects of the load () command—is a potential source of problems as well as a convenience. For example, if an object named Duncan exists in the global environment, it is silently replaced when you execute the command load (file="Duncan.RData"). An alternative is to use the command saveRDS () to save an R object like a data frame in a compressed format and readRDS (), which requires an explicit assignment (as in Duncan <-readRDS ("Duncan.rds")), to restore the object; see help ("readRDS").

## **2.1.6 Reading and Writing Other File Formats**

Data can be imported from other software to create data frames in R, including data in Microsoft Excel spreadsheets, SAS .sas7bdat files, SPSS .sav or .por files, Stata .dta files, and so on. Functions for reading data in a variety of file formats are available both in the standard R distribution, chiefly in the **foreign**

package, and in various contributed R packages. Using several other packages, the **rio** package (Chan, Chan, Leeper, & Becker, 2017) provides unified functions, import () and export (), which can respectively read and write data sets in many different formats. The **car** package includes a function named Import () that calls the import () function in **rio** but uses different defaults and arguments, more closely matching the specifications that we expect users of the **car** package to prefer. In particular:

1. The import () function does not support row names, while Import () allows you to convert a column in the input data file into row names.
2. The import () function does not support automatic conversion of character data to factors, while Import () by default converts character variables to factors.

For example, the Excel spreadsheet Duncan.xlsx, which you downloaded in [Section 1.1](#) to your R-Companion project, can be read as

**Duncan <- Import (file="Duncan.xlsx")**

If you view the Duncan.xlsx file in your spreadsheet program, you will see that it is in the same row and column format as a plain-text file, but the row-names column has been given the name id.<sup>14</sup> By default, Import () assumes that the first character column with unique values in the input data set represents row names. To read your own Excel file, you must ensure that column names are legal R names and appear in the first row of the spreadsheet; otherwise, Import () will at best make the names legal and at worst produce an error.

[14](#) It would also work to leave the column name of the first column blank. See help ("import") for a list of types of files that import (), and hence Import (), support. If you do not want Import () to use row names, set the argument row.names=FALSE. To suppress converting character variables to factors, add the argument stringsAsFactors=FALSE. For Excel files specifically, several additional arguments might be useful: The which argument may be used to specify the sheet number to read from an Excel file with several worksheets; which=1 is the default. The skip argument allows skipping the first few lines of a spreadsheet, with default skip=0.

The Export () function in the **car** package similarly writes data files in various formats. Export () is identical in use to the export () function in the **rio** package, but Export () has an additional argument, keep.row.names: Setting keep.row.names=TRUE adds the row names in a data frame as an additional initial column of the output data set, with the column name id. For example, to write the Duncan data frame, including row names, to a SAS data file:

**Export (Duncan, file="Duncan.sas7bdat", keep.row.names=TRUE)**

Export () (via export ()) infers the type of the output file from the file extension,

here .sas7bdat.

## 2.2 Other Approaches to Reading and Managing Data Sets in R

Hadley Wickham and his RStudio colleagues have produced a widely used set of R packages referred to collectively at the “tidyverse” (see Wickham & Grolemund, 2016). Installing the **tidyverse** package (Wickham, 2017) from CRAN via the command

```
install.packages ("tidyverse")
```

installs several other packages for reading, modifying, and visualizing data. In particular, the tidyverse **readr** package includes the functions `data_frame()`, `read_table()`, `read_csv()`, and `read_excel()` for reading data into a data-frame-like object called a *tibble*—an object of class "tbl\_df" that inherits from class "data.frame".<sup>15</sup> In [Section 2.3.5](#), we use the **dplyr** tidyverse package for an application that involves aggregating data, producing a tibble.

[15](#) See [Sections 1.7](#) and [10.9](#) for an explanation of how classes and inheritance work in R.

A tibble differs in several respects from an ordinary data frame:

1. The code for reading and writing tibbles is newer and faster than the analogous code for data frames, an advantage that can be important for large data sets (see [Section 2.7](#)).
2. Tibbles have a nicer `print()` method than do data frames, producing output that is similar to the output from the `brief()` function in the **car** package.
3. Some of the file-reading functions in the **readr** package are integrated with RStudio and can be accessed through the *Import Dataset* menu in the RStudio *Environment* tab.
4. The functions that create tibbles are relatively tolerant of nonstandard variable names—a mixed blessing, in that nonstandard names can cause problems when the tibble is used with other R functions.
5. The **readr** package and the tidyverse more generally are actively antagonistic to the use of row names. Avoiding row names may be a reasonable strategy for huge data sets where the cases don’t have individual identities beyond their row numbers, but automatic labeling of individual cases by their row names can be very useful in regression diagnostics and for other methods that identify unusual or influential cases (see [Section 8.3](#)).
6. Reading data from a file into a tibble does not automatically convert character variables to factors. As a general matter, the packages in the tidyverse are also hostile to factors and prefer to represent categorical data

as character variables. We think that factors offer some advantages over character variables, such as the ability to order levels naturally rather than alphabetically.

In our opinion, the advantages of the first three differences between tibbles and ordinary data frame are usually outweighed by the disadvantages of the last two differences, and so we continue to use data frames rather than tibbles in the *R Companion*. To be clear, tibbles support factors and row names, but they make it relatively difficult to use them. For example, the `read_table()` function in the `readr` package, which creates a tibble from a white-space-delimited text file, makes no provision for row names or for creating factors from character data. That said, all of the functions in the `car` packages that work with data frames also work with tibbles; when labeling cases in graphs and elsewhere, however, you must use the `label` argument to get labels other than row numbers.

Consequently, should you decide to use the tidyverse packages for data input and data management, you can still employ the `car` package.

Once you have installed the tidyverse packages, you can easily convert from a data frame to a tibble, and vice versa; for example:

```
library("tidyverse") # loads all of the tidyverse packages
Duncan.tibble <- as_tibble(Duncan)
print(Duncan.tibble, n=5) # note print() method

# A tibble: 45 x 4
  type   income education prestige
* <fct>  <int>     <int>    <int>
  1 prof      62        86       82
  2 prof      72        76       83
```

```

3 prof      75      92      90
4 prof      55      90      76
5 prof      64      86      90
# ... with 40 more rows

brief(as.data.frame(Duncan.tibble))

45 x 4 data.frame (40 rows omitted)

  type income education prestige
  [f]   [i]       [i]       [i]
accountant prof     62      86      82
pilot        prof    72      76      83
architect    prof    75      92      90
.
.
.
policeman   bc      34      47      41
waiter       bc      8       32      10

```

The variable type remains a factor in the tibble, and the conversion back to a data frame doesn't lose the row names, but, as we have explained, tibbles created directly from data files have neither factors nor row names.

There are other contributed R packages, most notably the **data.table** package (Dowle & Srinivasan, 2017), that support alternatives to data frames to read and represent data more efficiently.

## 2.3 Working With Data Frames

### 2.3.1 How the R Interpreter Finds Objects

This section explains how the R interpreter locates objects, a technical subject that is nevertheless worth addressing, because failure to understand it can result in confusion. To illustrate, we work with the Duncan occupational prestige data set, read in [Section 2.1.3](#):

### **str (Duncan)**

```
'data.frame':      45 obs. of  4 variables:  
 $ type      : Factor w/ 3 levels "bc","prof","wc": 2 2 2 2 2 ..  
 $ income    : int  62 72 75 55 64 21 64 80 67 72 ...  
 $ education: int  86 76 92 90 86 84 93 100 87 86 ...  
 $ prestige   : int  82 83 90 76 90 87 93 90 52 88 ...
```

When you type the name of an object in a command, such as the function `str ()` or the data set `Duncan`, the R interpreter looks for an object of that name in the locations specified by the *search path*, which may be printed by the `search ()` command:

### **search ()**

```
[1] ".GlobalEnv"           "package:forcats"     "package:stringr"  
[4] "package:dplyr"        "package:purrr"       "package:readr"  
[7] "package:tidyverse"     "package:tibble"      "package:ggplot2"  
[10] "package:tidyverse"     "package:car"         "package:carData"  
[13] "package:knitr"         "package:stats"       "package:graphics"  
[16] "package:grDevices"     "package:utils"       "package:datasets"  
[19] "package:methods"       "Autoloads"          "package:base"
```

The global environment, with contents listed in the RStudio *Environment* tab, is always the first location searched. Any object that you create by name at the R command prompt, such as the object `x` created by the assignment `x <- 3`, resides in the global environment. In parsing the command `str (Duncan)`, the R interpreter locates `Duncan` in the global environment. The `str ()` function is not found in the global environment, and so the interpreter searches through the other locations in the order given by the search path, until the `str ()` function is located in the *namespace* of the `utils` package. The packages on the search path displayed above, beginning with the `stats` package and ending with the `base` package, are part of the basic R system and are loaded by default when R starts up. The other packages in the path were put there either directly or indirectly by calls to the `library ()` function during the course of the current session.

The `carData` package also includes an object called `Duncan`, but because `carData` is further down the search path than the global environment, we say that `Duncan` in the global environment *shadows* or *masks* `Duncan` in the `carData` package. Had we deleted `Duncan` from the global environment, via the command `remove ("Duncan")`, the interpreter would have found `Duncan` in the `carData` package. As it happens, `Duncan` in the global environment is identical to `Duncan` in the `carData` package, and so it's immaterial which object is used. Because an object more commonly shadows an object of the *same name* but with *different*

*content*, this behavior can produce errors and confusion. You may have noticed that when a package is loaded, the library () command sometimes prints a message indicating *name clashes* of this kind.<sup>[16](#)</sup>

[16](#) R can tell syntactically, for example, that in the command str (Duncan), str () refers to a function and Duncan to a data object—formally, an R *variable*—and so the interpreter will ignore any objects named str that are located earlier on the path but are not functions. Thus, you can have both a variable and a function of the same name and R will use the right one depending on context. As mentioned, after [Chapter 1](#), we suppress package-startup messages, including warnings about name clashes.

Variables in the global environment or in a package on the search path may be referenced directly by name. There are several ways to reference variables in a data frame. For example, if you want to use the variable prestige in the Duncan data frame in a command, you can qualify the name of the variable with the name of the data frame in which it resides using a dollar sign (\$), or use square brackets ([, ]) to specify the column in the data set in which prestige is located:<sup>[17](#)</sup>

```
Duncan$prestige  
[1] 82 83 90 76 90 87 93 90 52 88 57 89 97 59 73 38 76 81 45 92  
[21] 39 34 41 16 33 53 67 57 26 29 10 15 19 10 13 24 20 7 3 16  
[41] 6 11 8 41 10  
mean(Duncan$prestige)  
[1] 47.689  
  
mean(Duncan[ , "prestige"]) # equivalent, column by name  
[1] 47.689  
  
mean(Duncan[ , 4]) # equivalent, column by number  
[1] 47.689
```

As we have seen, we can also use the with() function to access variables in a data frame:

```
with(Duncan, mean(prestige, trim=0.1))
```

```
[1] 47.297
```

[17](#) Indexing data frames (and other kinds of objects) is discussed in [Section 2.4.4](#).

As we have seen, we can also use the with () function to access variables in a

data frame:

```
with (Duncan, mean (prestige, trim=0.1))  
[1] 47.297
```

The first argument to with () is the name of a data frame, and the second argument is an expression, which can access variables in the data frame by name, in this case a call to the mean () function to compute the 10% trimmed mean of the variable prestige in the Duncan data frame.

Most statistical-modeling functions in R include a data argument, which allows us to reference variables in a data frame to specify a model. For example, to regress prestige in the Duncan data frame on income and education:

```
mod.duncan <- lm (prestige ~ income + education, data=Duncan)
```

To execute this command, R first looks for variables named income and education in the Duncan data set. Should either of these variables not reside in Duncan (which is not the case here), R would then look for the variables in the global environment and along the rest of the search path. It is a recipe for errors and confusion to combine variables from different locations in the same model, and so when we fit a statistical model in the *R Companion*, we will always access data from a single data frame. We recommend that you do the same. R provides yet another general mechanism for accessing variables in a data frame, using the attach () function—for example, attach (Duncan)—to add the data frame to the search path. By default, attach () adds a data frame at position two of the path, behind the global environment, pushing existing members of the path down one position. Similarly, by default, library () attaches an R package at position two of the path.

Despite its apparent convenience, we *strongly recommend that you do not use* attach (), because doing so can cause unanticipated difficulties:

- Attaching a data frame makes a *copy* of the attached data frame; the attached data set is a snapshot of the data frame at the moment when it is attached. If changes are made subsequently to the data frame, these are *not* reflected in the attached version of the data. Consequently, after making such a change, it is necessary to detach () and re-attach () the data frame. We find this procedure awkward and a frequent source of confusion and frustration.
- If two data frames containing (some) variables with the same names are simultaneously attached to the search path, variables in the data frame earlier on the path will shadow variables in the data frame later on the path.<sup>18</sup> It may, therefore, be unclear where the data are coming from when we execute a command. Potentially even worse, if you inadvertently attach two different versions of the *same* data frame to the search path, the

variables in the second version will shadow those in the first version.<sup>19</sup>

[18](#) If you want to experiment with this potentially confusing situation, try attaching both the Duncan and Prestige data frames from the **carData** package.

[19](#) Lest you think that this scenario is improbable, we observed it frequently in our classes before we discouraged the use of `attach()`.

Indeed, we mention `attach()` here to caution you against using it!

Name clashes, in which distinct objects of the same name are in different locations on the path, can occur for functions as well as for data objects. For example, there are different functions named `recode()` in the **car** and **dplyr** packages, both of which are currently on the search path; moreover, because the **dplyr** package, one of the tidyverse packages, was loaded *after* the **car** package in the current R session, the **dplyr** package appears *before* the **car** package on the path: Recall that the `library()` command by default attaches a package in position two of the search path, pushing other packages down the path. As a consequence, if we issue a `recode()` command, we'd invoke the function in the **dplyr** package. We can still use the `recode()` function in the **car** package by qualifying its name with the package namespace in which it resides followed by two colons: `car::recode()`.

## 2.3.2 Missing Data

Missing data are a regrettably common feature of real data sets. Two kinds of issues arise in handling missing data:

- There are relatively profound statistical issues concerning how best to use available information when missing data are encountered (see, e.g., Little & Rubin, 2002; Schafer, 1997). We will ignore these issues here, except to remark that R is well designed to make use of sophisticated approaches to missing data.<sup>20</sup>
- There are intellectually trivial but often practically vexing mechanical issues concerning computing with missing data in R. These issues, which are the subject of the present section, arise partly because of the diverse data structures and kinds of functions available simultaneously to the R user. Similar issues arise in *all* statistical software, however, although they may sometimes be disguised.

[20](#) Notable R packages for handling missing data include **Amelia** (Honaker, King, & Blackwell, 2011), **mi** (Gelman & Hill, 2015), **mice** (van Buuren & Groothuis-Oudshoorn, 2011), and **norm** (Schafer, 1997), which perform various versions of multiple imputation of missing data. We discuss multiple imputation of missing data in an online appendix to the *R Companion*.

Missing values are conventionally encoded in R by the symbol NA, and the

same symbol is used to print missing values. Most functions in R are equipped to handle missing data, although sometimes they have to be explicitly told what to do.

To illustrate, we introduce the Freedman data set in the **carData** package:

### ***brief(Freedman)***

| 110 x 4 data.frame (105 rows omitted) |            |          |         |       |
|---------------------------------------|------------|----------|---------|-------|
|                                       | population | nonwhite | density | crime |
|                                       | [i]        | [n]      | [i]     | [i]   |
| Akron                                 | 675        | 7.3      | 746     | 2602  |
| Albany                                | 713        | 2.6      | 322     | 1388  |
| Albuquerque                           | NA         | 3.3      | NA      | 5018  |
| ...                                   |            |          |         |       |
| York                                  | 316        | 2.0      | 220     | 1062  |
| Youngstown                            | 528        | 9.2      | 513     | 1698  |

These data, on 110 U.S. metropolitan areas, were originally from the *1970 Statistical Abstract of the United States* and were used by J. L. Freedman (1975) as part of a wide-ranging study of the social and psychological effects of crowding. Freedman argues, by the way, that high density tends to intensify social interaction, and thus the effects of crowding are not simply negative. The variables in the data set include 1968 total population, in thousands, 1960 percent nonwhite population, 1968 population density in persons per square mile, and the 1968 crime rate, in crimes per 100,000 residents.

The population and density for Albuquerque are missing. Here are the first few values for density, revealing more missing data:

```
head(Freedman$density, 20) # first 20 values  
[1] 746 322 NA 491 1612 770 41 877 240 147 272 1831  
[13] 1252 832 630 NA NA 328 308 1832
```

A more complete summary of the missing data pattern can be useful in some problems. The function **md.pattern()** in the **mice** package provides a compact, if less than self-explanatory, summary:

```
library("mice")
md.pattern(Freedman, plot=FALSE)
```

|     | nonwhite | crime | population | density |    |    |
|-----|----------|-------|------------|---------|----|----|
| 100 | 1        | 1     |            | 1       | 1  | 0  |
| 10  | 1        | 1     |            | 0       | 0  | 2  |
|     | 0        | 0     |            | 10      | 10 | 20 |

We use the argument `plot=FALSE` to suppress a graph of the missing-data patterns. The first row of the output indicates that 100 cases in the data set have complete data for all four variables, indicated by the 1 below each of the variable names in this row. The last entry in the first row, 0, means that zero values are missing for each of these 100 cases. The second row indicates that 10 cases have nonmissing values for nonwhite and crime but missing values for population and density, and so each of these 10 cases is missing two values. The third row is a general summary: Zero rows have missing values for each of nonwhite and crime, while 10 rows have missing values for each of population and density, and there is a total of 20 missing values in the data set.

Suppose now that we try to calculate the median density; as we will see shortly, the density values are highly positively skewed, and so using the *mean* as a measure of the center of the distribution would be a bad idea:

```
median (Freedman$density)
```

```
[1] NA
```

The median () function tells us that the median density is missing. This is the pedantically correct answer because 10 of the density values are missing, and consequently we cannot in the absence of those values know the median. By setting the `na.rm` (`na-remove`) argument of `median ()` to `TRUE`, we instruct R to calculate the median of the remaining 100 nonmissing values:

```
median (Freedman$density, na.rm=TRUE)
```

```
[1] 412
```

Many other R functions that calculate simple statistical summaries, such as `mean ()`, `var ()` (`variance`), `sd ()` (`standard deviation`), and `quantile ()`, also have `na.rm` arguments that behave in this manner.

The median of the 100 observed values of density is almost surely not the median density for the 110 metropolitan areas in the data set. Moreover, if the 110 cities were a random sample from a population of cities, then the median of all 110 densities would be an estimate of the population median, but, with missing values, the median of the observed values may not be a useful estimate

of the population median. (Reader: Can you think of why this might be the case?) The default value of the argument, na.rm=FALSE, reminds us to think about whether or not simply removing missing values makes sense in the context of our research. As we'll see presently, however, R isn't consistent in this respect.

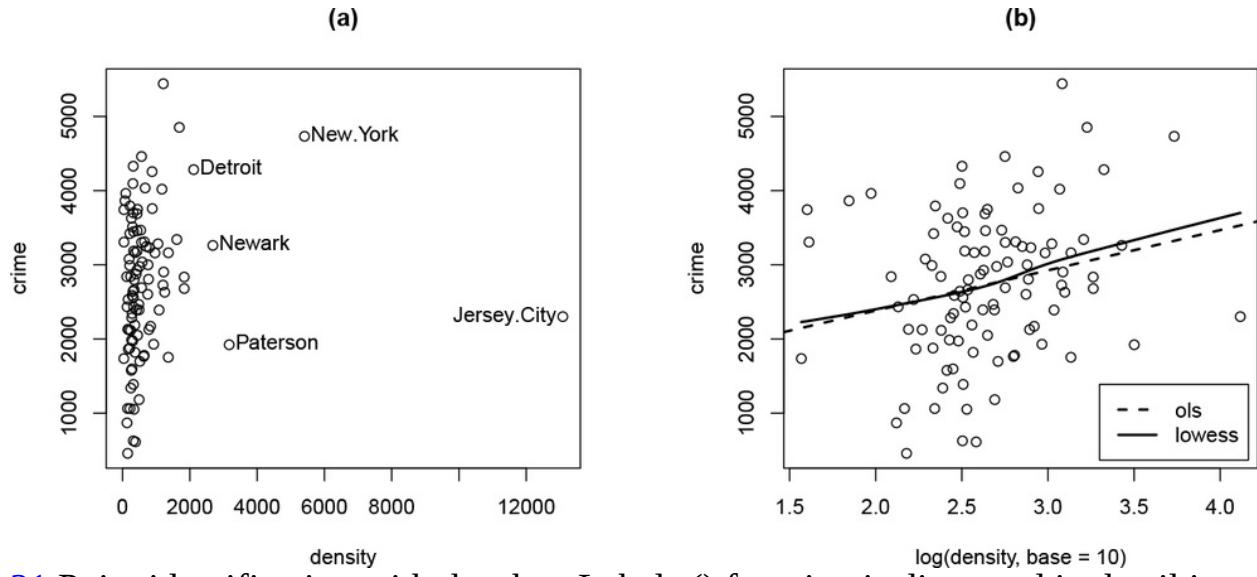
Not all R functions handle missing data via an na.rm argument. Most plotting functions, for example, simply and silently ignore missing data. A scatterplot of crime versus density, including only the cases with valid data for both variables, is produced by the command

```
with(Freedman, {  
  plot(density, crime, main="(a)")  
  showLabels(density, crime, labels=row.names(Freedman),  
             n=5, method="x")  
})
```

The second argument to with () is a *compound expression*, consisting of all the commands between the opening left-brace { and the closing right-brace }. In this case the compound expression consists of calls both to the plot () and to the showLabels () functions. Both functions reference variables in the Freedman data frame, supplied as the first argument to with ().

The resulting graph, including several cases identified using the showLabels () function in the **car** package,<sup>21</sup> appears in panel (a) of [Figure 2.1](#). The argument main="(a)" to plot () supplies the title for the graph. The arguments n=5 and method="x" to showLabels () tell the function to identify the five points with the most extreme values for density; four of the five identified cities with the largest densities are in the New York metropolitan area. It is apparent that density is highly positively skewed, making the plot very difficult to read. We would like to try plotting crime against the log of density to correct the skew but wonder whether the missing values in density will spoil the computation.<sup>22</sup>

**Figure 2.1** Scatterplots of crime by population density for Freedman's data: (a) Using the original density scale, with a few high-density cities identified by the showLabels () function, and (b) using the  $\log_{10}$  density scale, showing linear least-squares (broken) and lowess nonparametric-regression (solid) lines. Cases with one or both values missing are silently omitted from both graphs.



[21](#) Point identification with the `showLabels()` function is discussed in detail in [Section 3.5](#).

[22](#) Transformations, including the log transformation, are the subject of [Section 3.4](#).

Vectorized mathematical functions and operators in R, like `log()` and `*` (multiplication), return missing values only where math is performed elementwise on missing values:

```
log(c(1, 10, NA, 100), base=10)
```

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

```
c(1, NA, 3, 4) * c(2, 3, 4, NA)
```

```
[1] 2 NA 12 NA
```

We may therefore proceed as follows, producing the graph in panel (b) of [Figure 2.1](#):<sup>23</sup>

```
with(Freedman, plot(log(density, base=10), crime, main="(b)"))
```

[23](#) An alternative here is to plot crime against density using a log axis for density: `plot(density, crime, log="x")`. See [Chapters 3](#) and [9](#) for more general discussions of plotting data in R.

This graph is much easier to read, and it now appears that there is a weak, positive, roughly linear relationship between crime and  $\log(\text{density})$ . We will address momentarily how to produce the lines on the plot.

Most statistical-modeling functions in R have an `na.action` argument, which controls how missing data are handled; `na.action` is set to a function that takes a

data frame as an argument and returns a similar data frame composed entirely of valid data. The default na.action is na.omit (), which removes all cases with missing data on *any* variable in the computation. All the examples in the *R Companion* use na.omit (). An alternative, for example, would be to supply an na.action that substitutes imputed values for the missing values.<sup>24</sup>

[24](#) There is another missing-data function, na.exclude (), that is similar to na.omit () but that saves information about missing cases that are removed. That can be useful, for example, in labeling quantities, such as residuals, that are derived from a statistical model. We encourage the use of na.exclude () for modeling and other functions that work with it. The functions in the **car** package work properly with na.exclude (). Yet another standard missing-data filter is na.fail (), which causes a statisticalmodeling function to produce an error if any missing data are encountered—behavior consistent with the default behavior of simple statistical summaries like the median () function. You can change the default na.action with the options () command—for example, options (na.action="na.exclude").

The prototypical statistical-modeling function in R is lm (), which is described extensively in [Chapter 4](#). For example, to fit a linear regression of crime on  $\log_{10}$  (density), removing cases with missing data on either crime or density, enter the command

```
lm(crime ~ log(density, base=10), data=Freedman)
```

Call:

```
lm(formula = crime ~ log(density, base = 10), data = Freedman)
```

Coefficients:

|             |                         |
|-------------|-------------------------|
| (Intercept) | log(density, base = 10) |
| 1297        | 543                     |

The lm () function returns a linear-model object, but because we didn't assign the object to an R variable, the model was simply printed and not saved in the global environment. The fitted least-squares regression line is added to the scatterplot in [Figure 2.1 \(b\)](#) by the command

```
abline(lm(crime ~ log(density, base=10), data=Freedman),
lty="dashed", lwd=2)
```

The linear-model object returned by lm () is passed as an argument to abline (), which draws the regression line; specifying the line type lty="dashed" and line width lwd=2 produces a thick broken line.

Some functions in R, especially older ones inherited from early versions of S, make no provision for missing data and simply fail if an argument has a missing entry. In these cases, we need somewhat tediously to handle the missing data ourselves. A relatively straightforward way to do so is to use the `complete.cases()` function to find the location of valid data and then to exclude the missing data from the calculation. To locate all cases with valid data for *both* crime and density, we compute

```
good <- with(Freedman, complete.cases(crime, density))
head(good, 20) # first 20 values
```

|      |      |      |       |      |      |       |       |      |      |      |
|------|------|------|-------|------|------|-------|-------|------|------|------|
| [1]  | TRUE | TRUE | FALSE | TRUE | TRUE | TRUE  | TRUE  | TRUE | TRUE | TRUE |
| [11] | TRUE | TRUE | TRUE  | TRUE | TRUE | FALSE | FALSE | TRUE | TRUE | TRUE |

The logical vector `good` is TRUE for cases where values of both crime and density are present and FALSE if at least one of the two values is missing. We can then use `good` to select the valid cases by indexing (a topic described in detail in [Section 2.4.4](#)).

For example, it is convenient to use the `lowess()` function to add a nonparametric-regression smooth to the scatterplot in [Figure 2.1 \(b\)](#),<sup>25</sup> but `lowess()` makes no provision for missing data:

```
with(Freedman,
  lines(lowess(log(density[good]), base=10), crime[good], f=1.0), lwd=2))
  legend("bottomright", legend=c("OLS", "lowess"), lty=c("dashed",
  "solid"), lwd=2, inset=0.02)
```

<sup>25</sup> Lowess (Chambers, Cleveland, Kleiner, & Tukey, 1983) is an acronym for *locally weighted scatterplot smoother*, and the `lowess()` function is an implementation of *local polynomial regression*. R includes a more modern implementation of this method of nonparametric regression in the `loess()` function (Cleveland, 1994), which we often use in the `car` package for scatterplot smoothing. See [Section 9.2](#) for an explanation of *local linear regression*, which is a simplified version of `lowess`.

By indexing the predictor density and response crime with the logical vector `good`, we extract only the cases that have valid data for *both* variables. The argument `f` to the `lowess()` function specifies the *span* of the lowess smoother—that is, the fraction of the data included in each local-regression fit; large spans (such as the span 1.0 employed here) produce smooth regression curves. Finally, we used the `legend()` function to add a key at the lower right of the graph.<sup>26</sup>

<sup>26</sup> See [Chapter 9](#) on drawing graphs in R.

Suppose, as is frequently the case, that we analyze a data set with a complex pattern of missing data, fitting several statistical models to the data. If the

models do not all use the same variables, then it is likely that they will be fit to different subsets of complete cases. If we subsequently compare the models with a likelihood-ratio test, for example, the comparison will be invalid.<sup>27</sup>

<sup>27</sup> How statistical-modeling functions in R handle missing data is described in [Section 4.9.5](#). Likelihood-ratio tests for linear models are discussed in [Section 5.3.2](#) and for generalized linear models in [Section 6.3.4](#).

To avoid this problem, we can first call `na.omit()` to *filter* the data frame for missing data, including all the variables that we intend to use in our data analysis. For example, for Freedman's data, we may proceed as follows, assuming that we want subsequently to use all four variables in the data frame, eliminating the 10 cases with missing values for population and density:

```
Freedman.good <- na.omit(Freedman)
brief(Freedman.good)
```

|            | population | nonwhite | density | crime |
|------------|------------|----------|---------|-------|
|            | [i]        | [n]      | [i]     | [i]   |
| Akron      | 675        | 7.3      | 746     | 2602  |
| Albany     | 713        | 2.6      | 322     | 1388  |
| Allentown  | 534        | 0.8      | 491     | 1182  |
| ...        |            |          |         |       |
| York       | 316        | 2.0      | 220     | 1062  |
| Youngstown | 528        | 9.2      | 513     | 1698  |

*A cautionary note:* Filtering for missing data on variables that we *do not* intend to use can result in discarding data unnecessarily. We have seen cases where students and researchers inadvertently and needlessly threw away most of their data by filtering an entire data set for missing values, even when they intended to use only a few variables in the data set.

Finally, a few words about *testing* for missing data in R: A common error is to assume that one can check for missing values using the `==` (equals) operator, as in

```
NA == c(1, 2, NA, 4, NA)
[1] NA NA NA NA NA
```

As this small example illustrates, testing equality of NA against *any* value in R

returns NA as a result. After all, if the value is missing, how can we know whether it's equal to something else?

The proper way to test for missing data is with the `is.na()` function:

```
is.na (c (1, 2, NA, 4, NA))
```

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

For example, to count the number of missing values in the `Freedman` and `Freedman.good` data frames:

```
sum (is.na (Freedman))
```

```
[1] 20
```

```
sum (is.na (Freedman.good))
```

```
[1] 0
```

These commands rely on the automatic *coercion* of the logical matrices of TRUE and FALSE values produced by the commands `is.na` (`Freedman`) and `is.na` (`Freedman.good`) to matrices of zeros and ones, which can then be added by the `sum()` function.

### 2.3.3 Modifying and Transforming Data

Data modification in R often occurs naturally and unremarkably. When we wanted to plot crime against the log of density in Freedman's data, for example, we simply specified `log(density, base=10)` as an argument to the `plot()` function. We did not have to create a new variable, say `log.density <- log(density, 10)`, as one may be required to do in a typical statistical package, such as SAS or SPSS. Similarly, in regressing crime on the log of density, we just used `log(density, base=10)` on the right-hand side of the linear-model formula in a call to the `lm()` function.

Creating new variables that are functions of existing variables is straightforward. For example, the variable `cooperation` in the `Guyer` data set counts the number of cooperative choices out of a total of 120 choices in each of 20 four-person groups. To define a new variable with the percentage of cooperative choices in each group:

```
perc.coop <- 100*Guyer$cooperation/120
```

This command creates the variable `perc.coop` in the global environment and it is therefore displayed as a numeric vector in the RStudio *Environment* tab. We suggest that you instead add new variables like this to the data frame from which they originate. Keeping related variables together in a data frame decreases confusion and the possibility of error:

```
remove("perc.coop") # from the global environment
Guyer$perc.coop <- 100*Guyer$cooperation/120
brief(Guyer) # first 3 rows and last 2 rows
```

20 x 4 data.frame (15 rows omitted)

|       | cooperation | condition | sex    | perc.coop |
|-------|-------------|-----------|--------|-----------|
|       | [n]         | [f]       | [f]    | [n]       |
| 1     | 49          | public    | male   | 40.833    |
| 2     | 64          | public    | male   | 53.333    |
| 3     | 37          | public    | male   | 30.833    |
| . . . |             |           |        |           |
| 19    | 34          | anonymous | female | 28.333    |
| 20    | 44          | anonymous | female | 36.667    |

A similar procedure may be used to *modify* an existing variable in a data frame. The following command, for example, replaces the original cooperation variable in Guyer with the logit (log-odds) of cooperation:

```
Guyer$cooperation <- with(Guyer,
    log(perc.coop/(100 - perc.coop)))
brief(Guyer)
```

20 x 4 data.frame (15 rows omitted)

|       | cooperation | condition | sex    | perc.coop |
|-------|-------------|-----------|--------|-----------|
|       | [n]         | [f]       | [f]    | [n]       |
| 1     | -0.37086    | public    | male   | 40.833    |
| 2     | 0.13353     | public    | male   | 53.333    |
| 3     | -0.80792    | public    | male   | 30.833    |
| . . . |             |           |        |           |
| 19    | -0.92799    | anonymous | female | 28.333    |
| 20    | -0.54654    | anonymous | female | 36.667    |

We recommend that you work in an R script file or an R Markdown document when you modify a data frame, so that you create a reproducible record of what you have done. As well, saving the modified data set with the save () command, as in [Section 2.1.5](#), makes repeating data management operations in subsequent sessions unnecessary.

The transform () function can be used to create and modify several variables in a data frame at once. For example, if we have a data frame called Data with numeric variables named a, b, and c, then the command

```
Data <- transform (Data, c=-c, asq=a^2, a.over.b=a/b)
```

replaces Data by a new data frame in which the variables a and b are included unchanged, c is replaced by -c, and two new variables are added: asq, with the squares of a, and a.over.b, with the ratios of a to b.

The within () function is similar to transform (), except that new variables are created and existing ones altered by assignments. Like with (), the first argument of within () is a data frame and the second argument a compound expression; like transform (), within () returns a modified version of the original data frame, which then must be assigned to an object. For example:

```
Data <- within (Data, {  
  c <- -c  
  asq <- a^2 a.over.b <- a/b  
)
```

In some social science applications, continuous variables are binned to form categorial variables. Categorizing numeric data in this manner and recoding categorical variables are often more complicated than performing arithmetic calculations. Several functions in R may be employed to create factors from numeric data and to manipulate categorical data, but we will limit our discussion to three that we find particularly useful: (1) the standard R function cut (), (2) the Recode () function in the **car** package, and (3) the standard ifelse () function.

The cut () function dissects the range of a numeric variable into class intervals, or *bins*. The first argument to the function is the variable to be binned; the second argument gives either the number of equal-width bins or a vector of cut-points at which the division is to take place. For example, to divide the range of perc.coop in the Guyer data set into four equal-width bins, we specify

```

Guyer$coop.4 <- cut(Guyer$perc.coop, breaks=4)
summary(Guyer$coop.4)

(22.5,33.3] (33.3,44.2] (44.2,55] (55,65.9]
              6          7          5          2

```

The `cut()` function responds by creating a factor, the levels of which are named for the end points of the bins, unless you specify the level names directly via the optional `labels` argument. The argument `breaks=4` specifies dividing the range of the original variable into four bins of equal width. The first bin includes all values with `Guyer$perc.coop` greater than 22.5, which is slightly smaller than the minimum value of `Guyer$perc.coop`, and less than or equal to 33.3, the cut-point between the first two bins. Each bin contains a different number of cases, as shown in the summary; the `summary()` function applied, as here, to a factor prints a one-dimensional table of the number of cases in each level of the factor. Alternatively, to define three bins, named "low", "med", and "high", that contain approximately equal numbers of cases, we may proceed as follows:

```

Guyer$coop.thirds <- with(Guyer, cut(perc.coop,
  quantile(perc.coop, c(0, 1/3, 2/3, 1)),
  labels=c("low", "med", "high"),
  include.lowest=TRUE))
summary(Guyer$coop.thirds)

```

|     |     |      |
|-----|-----|------|
| low | med | high |
| 7   | 6   | 7    |

The `quantile()` function finds the specified quantiles of `perc.coop`. Had we wished to divide `perc.coop` into four groups, for example, we would simply have specified different quantiles, `c(0, 0.25, 0.5, 0.75, 1)`. The `cut()` function is called here with three arguments in addition to the variable to be binned: (1) the result returned by `quantile()`, specifying cut-points; (2) a character vector of labels for the three levels of the factor; and (3) `include.lowest=TRUE` to include the smallest value in the first interval.

The more flexible `Recode()` function in the `car` package can also be used to dissect a quantitative variable into class intervals.<sup>28</sup> For example,

```
(Guyer$coop.2 <- Recode(Guyer$perc.coop,
  'lo:50="low"; 50:hi="high" '))
[1] "low"  "high" "low"  "low"  "high" "low"  "high" "high"
[9] "high" "low"  "low"  "low"  "low"  "low"  "low"  "low"
[17] "low"  "low"  "low"  "low"
```

[28](#) Recode () is a synonym for the recode () function in the **car** package. As we explained in [Section 2.3.1](#), there is also a function named recode () in the **dplyr** package, which, in our current R session, appears earlier on the search path than the **car** package. Thus, recode () in **dplyr** shadows recode () in **car**. We could access the latter via the namespace-qualified name car::recode (), but we opt here for the simpler unqualified Recode ().

The Recode () function works as follows:

- The first argument is the variable to be recoded, here perc.coop.
- The second argument is a character string, enclosed in single quotes, as in the example, or double quotes, containing the recode specifications.
- Recode specifications are of the form *old.values=new.value*. There may be several recode specifications, separated by semicolons.
- The *old.values* may be a single value, including NA; a numeric range, of the form *minimum:maximum*, as in the example, where the special values lo and hi are used to stand in for the smallest and largest values of the variable, and where *minimum* and *maximum* need not be integers; a vector of values, typically specified with the c () function; or the special symbol else, which, if present, should appear last.
- A case that fits into more than one recode specification is assigned the value of the first one encountered. For example, a group with perc.coop exactly equal to 50 would get the new value "low".
- Character data may appear both as *old.values* and as *new.value*. You must be careful with quotation marks, however: If single quotes are employed to enclose the recode specifications, then double quotes must be used for the values, as in this example. Similarly, if you use double quotes around the recode specifications, then you must use single quotes around character values.
- When a factor is recoded, the levels in *old.values* should be specified as character strings; the result is a factor, even if the *new.values* are numbers, unless the argument as.factor is set to FALSE.
- Character data may be recoded to numeric and vice versa. To recode a character or numeric variable to a factor, set as.factor=TRUE.
- If a case does not satisfy any of the recode specifications, then the existing

value for that case is carried over into the result.

To provide a richer context for additional illustrations of the use of Recode (), we turn our attention to the Womenlf data frame from the **carData** package:

```

set.seed(12345) # for reproducibility
(sample.20 <- sort(sample(nrow(Womenlf), 20))) # 20 random cases

[1] 1 9 39 43 44 84 96 98 100 115 119 131 185 186 190
[16] 199 230 231 233 252

Womenlf[sample.20, ] # 20 randomly selected rows

      partic hincome children   region
1    not.work       15  present Ontario
9    not.work       15  present Ontario
39   not.work        9  present Atlantic
43   parttime       28  absent  Ontario
44   not.work       23  present Ontario
84   fulltime       17  present Ontario
96   not.work       17  present Ontario
98   fulltime       15  absent  Ontario
100  not.work       15  present Ontario
115  parttime       13  present Prairie
119  fulltime       15  absent   BC
131  parttime       19  present Ontario
185  not.work       13  absent  Ontario
186  parttime       15  present   BC
190  not.work       23  present   BC
199  fulltime       10  absent  Quebec
230  parttime       23  present  Quebec
231  not.work        7  present  Quebec
233  fulltime       15  absent  Quebec
252  not.work       23  absent  Quebec

```

The `sample()` function is used to pick a random sample of 20 rows in the data frame, selecting 20 random numbers without replacement from one to the number of rows in `Womenlf`; the numbers are placed in ascending order by the `sort()` function.<sup>29</sup>

<sup>29</sup> If the objective here were just to sample 20 rows from `Womenlf`, then we could more simply use the `some` function in the `car` package, `some(Womenlf, 20)`, but we will reuse this sample to check on the results of our recodes.

We call the `set.seed()` function to specify the seed for R's pseudo-random-number generator, ensuring that if we repeat the `sample()` command, we will obtain the same sequence of pseudo-random numbers. Otherwise, the seed of the random-number generator is selected unpredictably based on the system clock

when the first random number is generated in an R session. Setting the random seed to a known value before a random simulation makes the result of the simulation reproducible. In serious work, we generally prefer to start with a known but randomly selected seed, as follows:<sup>30</sup>

```
(seed <- sample (2^31 - 1, 1))
```

```
[1] 974373618
```

```
set.seed (seed)
```

[30 Section 10.7](#) discusses using R in random simulations.

The number  $2^{31} - 1$  is the largest integer representable as a 32-bit binary number.

The data in `Womenlf` originate from a social survey of the Canadian population conducted in 1977 and pertain to married women between the ages of 21 and 30, with the variables defined as follows:

- `partic`: Labor force participation, "parttime", "fulltime", or "not. work" (not working outside the home)
- `hincome`: Husband's income, in \$1,000s (actually, family income minus wife's income)
- `children`: Presence of children in the household: "present" or "absent"
- `region`: "Atlantic", "Quebec", "Ontario", "Prairie" (the Prairie provinces), or "BC" (British Columbia)

Now consider several recodes. In all these examples, factors, either `partic` or `region`, are recoded, and consequently `Recode()` returns factors as results:

```
# recode in two ways:
Womenlf$working <- Recode(Womenlf$partic,
  ' c("parttime", "fulltime")="yes"; "not.work"="no" ')
Womenlf$working.alt <- Recode(Womenlf$partic,
  ' c("parttime", "fulltime")="yes"; else="no" ')
Womenlf$working[sample.20] # the 20 sampled cases

[1] no  no  no  yes no  yes no  yes yes yes no  yes no
[16] yes yes no  yes no
Levels: no yes

with(Womenlf, all(working == working.alt)) # check

[1] TRUE
```

The two examples above yield identical results, with the second example illustrating the use of `else` in the `Recode()` specification. To verify that all the values in `working` and `working.alt` are the same, we use the `all()` function along with the elementwise comparison operator `==` (equals).

```

Womenlf$fulltime <- Recode(Womenlf$partic,
  ' "fulltime"="yes"; "parttime"="no"; "not.work"=NA ')
Womenlf$fulltime[sample.20] # the 20 sampled cases

[1] <NA> <NA> <NA> no    <NA> yes   <NA> yes   <NA> no    yes   no
[13] <NA> no    <NA> yes   no    <NA> yes   <NA>

Levels: no yes

```

Recode () creates the factor fulltime, indicating whether a woman who works outside the home works full-time or part-time; fulltime is NA (missing) for women who do not work outside the home.

A final example illustrates how values that are *not* recoded (here "Atlantic", "Quebec", and "Ontario" in the factor region) are simply carried over to the result:

```

Womenlf$region.4 <- Recode(Womenlf$region,
  ' c("Prairie", "BC")="West" ')
Womenlf$region.4[sample.20] # the 20 sampled cases

[1] Ontario  Ontario  Atlantic Ontario  Ontario  Ontario
[7] Ontario  Ontario  Ontario  West     West     Ontario
[13] Ontario  West     West     Quebec  Quebec  Quebec
[19] Quebec  Quebec

Levels: Atlantic Ontario Quebec West

```

The standard R ifelse () command, discussed further in [Section 10.4.1](#), can also be used to recode data. For example:

```

Womenlf$working.alt.2 <- factor (with (Womenlf, ifelse (partic %in% c
  ("parttime", "fulltime"), "yes", "no")))
  with (Womenlf, all.equal (working, working.alt.2)))
[1] TRUE

```

The first argument to ifelse () is a logical vector, containing the values TRUE and FALSE, generated in the example by the expression partic %in% c ("parttime", "fulltime"); the second argument supplies the value to be assigned where the first argument is TRUE; and the third argument supplies the value to be assigned where the first argument is FALSE.

The logical expression in this example uses the *matching operator*, %in%, which returns a logical vector containing TRUE wherever an element in the first vector operand (partic) is a member of the second vector operand (c ("parttime", "fulltime")) and FALSE otherwise. See help ("%in%") for more information.

The example also uses the `all.equal()` function to test the equality of the alternative recodings of partic. When applied to numeric variables, `all.equal()` tests for *approximate* equality, within the precision of floating-point computations; more generally, `all.equal()` not only reports whether two objects are approximately equal but, if they are not equal, provides information on how they differ.

The second and third arguments to `ifelse()` may also be vectors of the same length as the first argument; we contrive a small example to illustrate:

```
husbands.income <- c(10, 30, 50, 20, 120) # imagine in $1000s
wifes.income     <- c(15, 20, 45, 22,  90)
ifelse (husbands.income > wifes.income,
       husbands.income, wifes.income) # larger of the two

[1] 15 30 50 22 120
```

For this computation, we could also use the `pmax()` (parallel maxima) function:

```
pmax (husbands.income, wifes.income)
```

```
[1] 15 30 50 22 120
```

See `help("pmax")` for this and other maxima and minima functions.

We next use “cascading” `ifelse()` commands to create the variable `fulltime.alt`, assigning the value "yes" to those working "fulltime", "no" to those working "parttime", and NA otherwise (i.e., where `partic` takes on the value "not.work"):

```
Womenlf$fulltime.alt <- factor (with (Womenlf,
  ifelse (partic == "fulltime", "yes",
  ifelse (partic == "parttime", "no", NA)))
  with (Womenlf, all.equal (fulltime, fulltime.alt)))
[1] TRUE
```

An alternative to the test in the first `ifelse()` example is `all(working == working.alt.2)`, but this approach won’t work properly in the last example because of the missing data:<sup>31</sup>

```
with (Womenlf, all (fulltime == fulltime.alt))
[1] NA
```

<sup>31</sup> See [Section 2.3.2](#) for a discussion of missing data.

We clean up before proceeding by removing the Guyer data set along with the copy of `Womenlf` that we made in the global environment:

```
remove (Guyer, Womenlf)
```

When you modify a data frame originally obtained from an R package, such as the `Womenlf` data frame from the `carData` package, a *copy* of the data set is made in the global environment, and the original data set in the package is

unaltered. The modified copy of the data in the global environment then shadows the original data set in the package (see [Section 2.3.1](#)). The remove () command deletes the copy of the data set from the global environment, once more exposing the original data set in the package.

### 2.3.4 Binding Rows and Columns

R includes useful functions for concatenating data frames, matrices, and vectors, either by rows or by columns. The rbind () function attaches two or more objects with the same number of columns, one below the next. For example, in the Guyer data set, introduced in [Section 2.1.2](#) and a copy of which is in the **carData** package, some rows correspond to groups composed of female subjects and the others to groups composed of male subjects. For purposes of illustration, we divide the Guyer data frame into two parts by sex:

```
brief(Guyer.male <- Guyer[Guyer$sex == "male", ],  
      rows=c(2, 1))  
  
10 x 3 data.frame (7 rows omitted)  
  cooperation condition sex  
    [n]          [f]  [f]  
 1           49 public male  
 2           64 public male  
 . . .  
 15          30 anonymous male  
  
brief(Guyer.female <- Guyer[Guyer$sex == "female", ],  
      rows=c(2, 1))  
  
10 x 3 data.frame (7 rows omitted)  
  cooperation condition sex  
    [n]          [f]  [f]  
 6           54 public female  
 7           61 public female  
 . . .  
 20          44 anonymous female
```

The two resulting data frames can then be recombined by the command

```
brief(Guyer.reordered <- rbind(Guyer.male, Guyer.female))

20 x 3 data.frame (15 rows omitted)
  cooperation condition sex
      [n]       [f]   [f]
1         49 public male
2         64 public male
3         37 public male
. . .
19        34 anonymous female
20        44 anonymous female
```

Guyer.reordered has all the rows for male groups preceding all the rows for female groups.

The cbind () function similarly concatenates data frames, matrices, and vectors by columns. Applied to data frames, cbind () requires that the rows of its arguments correspond to the same cases in the same order and similarly for matrices and vectors. For example, we can use cbind () to add a new variable, percent cooperation, to the Guyer.reordered data frame:

```
brief(Guyer.reordered <- cbind(Guyer.reordered,
pctcoop=round(100*Guyer.reordered$cooperation/120, 2)))

20 x 4 data.frame (15 rows omitted)
  cooperation condition sex pctcoop
      [n]       [f]   [f]     [n]
1         49 public male    40.83
2         64 public male    53.33
3         37 public male    30.83
. . .
19        34 anonymous female  28.33
20        44 anonymous female  36.67
```

## 2.3.5 Aggregating Data Frames

Interesting data often come from more than one source, and data analysis can therefore require the potentially complicated preliminary step of combining data from different sources into a single rectangular data set. We consider by way of example two data frames in the **carData** package:

1. MplsStops contains data on stops of individuals by Minneapolis police officers during the calendar year 2017. Included are nearly all stops for

which the stated reason was either a traffic issue or a suspicious person or vehicle. The Minneapolis Police Department publishes these data on their public-access website (Minneapolis Police Department, 2018). The variables in this data set include geographic information, such as longitude, latitude, neighborhood name, and police precinct number; temporal data, giving the date and time of the stop; and other descriptors, such as the type of stop, the race and gender of the person stopped, and outcomes of the stop assessed in a variety of ways. A complete description of the data is given in help ("MplsStops"). In all there are 51,920 stops recorded in the data set, with measurements on 14 variables.

#### ***brief (MplsStops)***

| 51920 x 14 data.frame (51915 rows and 11 columns omitted) |           |                |                 |
|---|-----------|----------------|-----------------|
|   | idNum     | policePrecinct | neighborhood    |
|   | [f]       | [i]            | [f]             |
| 6823  | 17-000003 | 1              | Cedar Riverside |
| 6824  | 17-000007 | 1              | Downtown West   |
| 6825  | 17-000073 | 5              | Whittier        |
| .   | .         | .              | .               |
| 60837   | 17-491480 | 2              | Marcy Holmes    |
| 60838   | 17-491482 | 5              | Lowry Hill East |

2. MplsDemo provides demographic information for Minneapolis neighborhoods, from the 2011–2015 American Community Survey (Minnesota Compass, 2018). This data frame has one row for each of 84 Minneapolis neighborhoods. See help ("MplsDemo") for definitions of the variables available.

#### ***brief (MplsDemo)***

| 84 x 8 data.frame (79 rows and 4 columns omitted) |                  |            |         |             |
|---|------------------|------------|---------|-------------|
|   | neighborhood     | population | poverty | collegeGrad |
|   | [c]              | [n]        | [n]     | [n]         |
| 1   | Cedar Riverside  | 8247       | 0.060   | 0.258       |
| 3   | Phillips West    | 5184       | 0.042   | 0.211       |
| 4   | Downtown West    | 7141       | 0.057   | 0.551       |
| .   | .                | .          | .       | .           |
| 98  | Columbia Park    | 1699       | 0.058   | 0.418       |
| 100   | Marshall Terrace | 1587       | 0.063   | 0.409       |

Suppose that we want to use the demographic variables to model one of the outcomes that can be computed from the stops data, such as the total number of stops or the fraction of traffic stops that resulted in a citation. One approach is to aggregate the stops data in MplsStops by neighborhood, merging the aggregated data with the neighborhood data in MplsDemo. It's possible to merge the two data sets because their rows pertain to the same neighborhoods.

We use several functions in the **dplyr** package, one of the tidyverse packages loaded earlier in the chapter, to aggregate the data in MplsStops by neighborhood:

```
MplsStops %>% group_by(neighborhood) %>%
  summarize(nstops = n(),
            ntraffic = sum(problem == "traffic"),
            nNoCitation = sum(problem == "traffic" &
                                citationIssued == "NO", na.rm=TRUE),
            nYesCitation = sum(problem == "traffic" &
                                citationIssued == "YES", na.rm=TRUE),
            lat = mean(lat),
            long = mean(long)) -> Neighborhood
```

This command uses syntax that we have not yet encountered and which requires some explanation:

- We use the *pipe operator* `%>%`, supplied by several of the tidyverse packages, to compose successive function calls. The effect is to supply the left operand of `%>%` as the first argument in the function call given as the right operand. For example, the MplsStops data frame is the first argument to the `group_by()` function and the variable neighborhood is the second argument to `group_by()`. The `group_by()` function divides the MplsStops data frame into groups according to the value of neighborhood. The resulting object is the first argument to the `summarize()` function that computes summaries for each neighborhood. This result is then saved in the new object called Neighborhood. Functions like `group_by()` and `summarize()` in the **dplyr** package are written to work naturally with pipes, but the resulting syntax is distinctly un-R-like, with data flowing from left to right and top to bottom in the command.
- The various arguments to `summarize()` create columns in the aggregated data set, where the rows are neighborhoods; the argument names, such as

ntraffic and lat, become column names in the aggregated data. The variable nstops uses the n() function in the **dplyr** package to count the number of stops in each neighborhood. We use the standard-R sum() function to count the number of traffic stops, the number of traffic stops resulting in no citation, and the number resulting in a citation. The variable citationIssued in the disaggregated MplsStops data set consists mostly of NAs, and so we have to be careful to filter out missing data. Finally, we compute the mean latitude and longitude for the stops in each neighborhood as a representative location in the neighborhood.

- Because the flow of data using pipes is from left to right, it is natural to use the *right-assign operator* -> to assign the resulting data set to the R variable Neighborhood. This data set is a tibble, not a data frame (see [Section 2.2](#)):

### **Neighborhood**

```
# A tibble: 87 x 7
  neighborhood   nstops ntraffic nNoCitation nYesCitation     lat
  <fct>        <int>    <int>       <int>       <int> <dbl>
1 Armatage         77      12          5          0  44.9
2 Audubon Park     554     348         76         15  45.0
3 Bancroft        134      21          7          4  44.9
4 Beltrami         211     158         33         30  45.0
5 Bottineau        377     281         55         19  45.0
6 Bryant            96      19          7          1  44.9
7 Bryn - Mawr      125      47         13         10  45.0
8 Camden Indus...     34      22          9          2  45.0
9 CARAG            559     325         95         18  44.9
10 Cedar - Isle...    153     99         26          3  45.0
# ... with 77 more rows, and 1 more variable: long <dbl>
```

## 2.3.6 Merging Data Frames

Merging data sets (also called *match-merging*) is potentially a more complex operation than simply concatenating rows and columns. We illustrate by merging the Neighborhood data set, created in the preceding section, with the MplsDemo data. The variable neighborhood is the same in both data sets, except that it is a factor in the Neighborhood tibble and a character variable in the MplsDemo data frame.

We use the merge() function, which is part of the standard R distribution, to combine the two data sets:<sup>32</sup>

```

Neigh.combined <- merge(Neighborhood, MplsDemo,
  by="neighborhood")
brief(Neigh.combined)

84 x 14 data.frame (79 rows and 9 columns omitted)
  neighborhood nstops ntraffic . . . poverty collegeGrad
    [f]     [i]     [i]     [n]     [n]
1 Armatage      77      12      0.050    0.622
2 Audubon Park  554     348      0.053    0.440
3 Bancroft     134      21      0.053    0.443
. . .
83 Windom       404     221      0.050    0.543
84 Windom Park  461     303      0.058    0.499

```

[32](#) This operation works even though Neighborhood is a tibble and MplsDemo is a data frame.

The argument by supplies the *key* for the match, in this case, the variable neighborhood. There can be more than one key, and names of the keys do not have to be the same in the two data sets, as long as their contents match. We could now proceed to analyze the Neigh.combined data set, which combines neighborhood data on traffic stops with demographic neighborhood data. You may have noticed that the MplsDemo data frame has 84 rows while Neighborhood had 87:

```

Neighborhood$neighborhood[!(Neighborhood$neighborhood
  %in% MplsDemo$neighborhood)]

```

```

[1] Camden Industrial          Humboldt Industrial Area
[3] Near - North

```

87 Levels: Armatage Audubon Park Bancroft ... Windom Park  
 These three neighborhoods apparently have no housing, and therefore no demographic information. The merge () function silently omitted these neighborhoods from the merged data set, but they can be retained, with NAs entered for the missing demographics:

```

Neigh.combined.2 <- merge(Neighborhood, MplsDemo,
  by="neighborhood", all.x=TRUE)
dim (Neigh.combined.2)
[1] 87 14

```

See help ("merge") for details.

It is also possible to merge the individual-stops data with the demographics data

set or with the combined neighborhood-level data:

```
MplsStops.2 <- merge(MplsStops, Neigh.combined.2,
  by="neighborhood")
brief(MplsStops.2)

51920 x 27 data.frame (51915 rows and 23 columns omitted)
  neighborhood      idNum . . . poverty collegeGrad
              [f]          [f]     [n]        [n]
1       Armatage 17-354249    0.050    0.622
2       Armatage 17-156263    0.050    0.622
3       Armatage 17-263998    0.050    0.622
. . .
51919 Windom Park 17-111453    0.058    0.499
51920 Windom Park 17-127035    0.058    0.499
```

The resulting data frame has one row for each of the rows in the MplsStops data set. The neighborhood-level data in Neigh.combined.2 is repeated for each stop within the corresponding neighborhood, including the three neighborhoods with no demographic data with NA filled in appropriately. This kind of data management operation is common in assembling data sets for hierarchical models, discussed in [Chapter 7](#).

### 2.3.7 Reshaping Data

Analyzing longitudinal data, or other data with repeated observations on the same individuals, may require moving between *wide* and *long* representations of the data. In the wide version of a data set, repeated observations on each individual are represented as separate variables in a single row of the data set. In the long version of the data set, each repeated observation is recorded in a separate row of the data set, which includes an *identifier variable* (or *ID variable*) that indicates that the rows belong to a single individual. See [Figure 2.2](#) for a schematic representation of wide and long data.

**Figure 2.2** Schematic representation of (a) wide and (b) long data, for two variables x and y collected on three occasions, showing the first three of presumably many cases. The values of x and y are simply numbered serially.

(a) Wide form

| Case | x1 | x2 | x3 | y1 | y2 | y3 |
|------|----|----|----|----|----|----|
| 1    | 1  | 2  | 3  | 1  | 2  | 3  |
| 2    | 4  | 5  | 6  | 4  | 5  | 6  |
| 3    | 7  | 8  | 9  | 7  | 8  | 9  |

...

(b) Long form

| Case | x | y | occasion |
|------|---|---|----------|
| 1    | 1 | 1 | 1        |
| 1    | 2 | 2 | 2        |
| 1    | 3 | 3 | 3        |
| 2    | 4 | 4 | 1        |
| 2    | 5 | 5 | 2        |
| 2    | 6 | 6 | 3        |
| 3    | 7 | 7 | 1        |
| 3    | 8 | 8 | 2        |
| 3    | 9 | 9 | 3        |

...

Whether wide or long data are required depends on the software that we use to analyze the data. For example, traditional repeated-measures analysis of variance, as implemented by the Anova () function in the **car** package, requires a

multivariate linear model fit to a wide data set, in which the repeated measures are treated as multiple response variables.<sup>33</sup> In contrast, modern mixed-effects regression models, as implemented in the **nlme** and **lme4** packages discussed in [Chapter 7](#), assume that the data are in long form.

[33](#) Multivariate and repeated-measures ANOVA are discussed in an online appendix to the *R Companion*.

An example of repeated-measures data in wide form is provided by the OBrienKaiser data set, from O'Brien and Kaiser (1985), in the **carData** package:

```
head(OBrienKaiser, 2) # first two subjects
```

|   | treatment | gender | pre.1 | pre.2 | pre.3 | pre.4 | pre.5 | post.1 | post.2 |
|---|-----------|--------|-------|-------|-------|-------|-------|--------|--------|
| 1 | control   | M      | 1     | 2     | 4     | 2     | 1     | 3      | 2      |
| 2 | control   | M      | 4     | 4     | 5     | 3     | 4     | 2      | 5      |
| 1 | 5         | 3      | 2     | 2     | 3     | 2     | 4     | 4      |        |
| 2 | 3         | 5      | 3     | 4     | 5     | 6     | 4     | 1      |        |

```
nrow(OBrienKaiser)
```

```
[1] 16
```

The variables treatment and gender in this data set are so-called *between-subjects factors*, with levels "control", "A", and "B", and "F" (female) and "M" (male), respectively. Although it doesn't say so explicitly in the original source, these are probably contrived data, and so we will imagine that the 16 subjects in the study are students with attention deficit disorder, that treatment represents three treatment groups to which the subjects are randomly assigned, and that the repeated measures—the remaining variables in the data set—represent scores on a task requiring attention. This task is administered in three phases: prior to treatment (pre), immediately after treatment (post), and several weeks after treatment (fup, “follow-up”). During each phase, subjects are tested five times, at hourly intervals, producing 15 repeated measures in all.

The regular structure of the OBrienKaiser data makes it simple to transform the data to long form using the base-R `reshape()` function:

```

OBK.L <- reshape(OBrienKaiser, direction="long",
                  varying=list(response=3:17),
                  v.names="response", idvar="subject")
nrow(OBK.L)

[1] 240

head(OBK.L, 16) # first measurement for each subject

```

|      | treatment | gender | time | response | subject |
|------|-----------|--------|------|----------|---------|
| 1.1  | control   | M      | 1    | 1        | 1       |
| 2.1  | control   | M      | 1    | 4        | 2       |
| 3.1  | control   | M      | 1    | 5        | 3       |
| 4.1  | control   | F      | 1    | 5        | 4       |
| 5.1  | control   | F      | 1    | 3        | 5       |
| 6.1  | A         | M      | 1    | 7        | 6       |
| 7.1  | A         | M      | 1    | 5        | 7       |
| 8.1  | A         | F      | 1    | 2        | 8       |
| 9.1  | A         | F      | 1    | 3        | 9       |
| 10.1 | B         | M      | 1    | 4        | 10      |
| 11.1 | B         | M      | 1    | 3        | 11      |
| 12.1 | B         | M      | 1    | 6        | 12      |
| 13.1 | B         | F      | 1    | 5        | 13      |
| 14.1 | B         | F      | 1    | 2        | 14      |
| 15.1 | B         | F      | 1    | 2        | 15      |
| 16.1 | B         | F      | 1    | 4        | 16      |

- The first argument to reshape () is the data set to be transformed, OBrienKaiser.
- The direction argument, in this case "long", indicates that a wide data set is transformed to long form.
- The varying argument is a list of sets of variables in the wide form of the data that are to constitute single variables in the long form of the data; in this case, there is only one such set of variables. The list elements need not be named, but doing so helps keep things straight.

- The `v.names` argument supplies names for the variables in the long form of the data, in this case the single variable `response`.

See `help("reshape")` for details.

The reshaped data set, `OBK.L`, consists of  $16 \times 15 = 240$  rows. The first 16 rows, shown above, represent the first measurement (`pre.1` in the wide data) for the 16 subjects (as recorded in the `ID` variable `subject`), marked as `time = 1` in the long data. The next 16 rows, not shown, represent the second measurement (from `pre.2`, marked `time = 2`), and so on. The between-subjects factors in the original wide form of the data, `treatment` and `gender`, are repeated for each subject in the 15 rows pertaining to that subject, for which `time = 1, 2, \dots, 15`. It is convenient but not essential for analyzing the long form of the data to reorder the rows so that all rows for a particular subject are adjacent in the data:

```

ord <- with(OBK.L, order(subject, time))
OBK.L <- OBK.L[ord, ]
head(OBK.L, 15) # rows for subject 1

```

|      | treatment | gender | time | response | subject |
|------|-----------|--------|------|----------|---------|
| 1.1  | control   | M      | 1    | 1        | 1       |
| 1.2  | control   | M      | 2    | 2        | 1       |
| 1.3  | control   | M      | 3    | 4        | 1       |
| 1.4  | control   | M      | 4    | 2        | 1       |
| 1.5  | control   | M      | 5    | 1        | 1       |
| 1.6  | control   | M      | 6    | 3        | 1       |
| 1.7  | control   | M      | 7    | 2        | 1       |
| 1.8  | control   | M      | 8    | 5        | 1       |
| 1.9  | control   | M      | 9    | 3        | 1       |
| 1.10 | control   | M      | 10   | 2        | 1       |
| 1.11 | control   | M      | 11   | 2        | 1       |
| 1.12 | control   | M      | 12   | 3        | 1       |
| 1.13 | control   | M      | 13   | 2        | 1       |
| 1.14 | control   | M      | 14   | 4        | 1       |
| 1.15 | control   | M      | 15   | 4        | 1       |

The 15 occasions on which each subject is measured are further structured by phase and hour. Because of the regular structure of the data, we can easily add these *within-subjects factors* to the long form of the data:

```

OBK.L$phase <- factor(rep(rep(c("pre", "post", "fup"),
each=5), 16))
OBK.L$hour <- factor(rep(rep(1:5, 3), 16))
head(OBK.L, 15)

```

|      | treatment | gender | time | response | subject | phase | hour |
|------|-----------|--------|------|----------|---------|-------|------|
| 1.1  | control   | M      | 1    | 1        | 1       | pre   | 1    |
| 1.2  | control   | M      | 2    | 2        | 1       | pre   | 2    |
| 1.3  | control   | M      | 3    | 4        | 1       | pre   | 3    |
| 1.4  | control   | M      | 4    | 2        | 1       | pre   | 4    |
| 1.5  | control   | M      | 5    | 1        | 1       | pre   | 5    |
| 1.6  | control   | M      | 6    | 3        | 1       | post  | 1    |
| 1.7  | control   | M      | 7    | 2        | 1       | post  | 2    |
| 1.8  | control   | M      | 8    | 5        | 1       | post  | 3    |
| 1.9  | control   | M      | 9    | 3        | 1       | post  | 4    |
| 1.10 | control   | M      | 10   | 2        | 1       | post  | 5    |
| 1.11 | control   | M      | 11   | 2        | 1       | fup   | 1    |
| 1.12 | control   | M      | 12   | 3        | 1       | fup   | 2    |
| 1.13 | control   | M      | 13   | 2        | 1       | fup   | 3    |
| 1.14 | control   | M      | 14   | 4        | 1       | fup   | 4    |
| 1.15 | control   | M      | 15   | 4        | 1       | fup   | 5    |

Now imagine that we *start* with the long form of the O'Brien–Kaiser data, OBK.L, and wish to transform the data into wide form. We can use reshape() in reverse, in this case recovering the original OBrienKaiser data set (but with the additional variable subject):

```

OBK.W <- reshape(OBK.L, direction="wide", idvar="subject",
  v.names="response", drop=c("phase", "hour"),
  varying=list(response=paste0(
    rep(c("pre", "post", "fup"), each=5), ".", rep(1:5, 3)))))
head(OBK.W, 2)

  treatment gender subject pre.1 pre.2 pre.3 pre.4 pre.5
1.1   control      M      1      1      2      4      2      1
2.1   control      M      2      4      4      5      3      4
  post.1 post.2 post.3 post.4 post.5 fup.1 fup.2 fup.3 fup.4
1.1      3      2      5      3      2      2      3      2      4
2.1      2      2      3      5      3      4      5      6      4
  fup.5
1.1      4
2.1      1

nrow(OBK.W)
[1] 16

# ignore subject in column 3 of OBK.W:
all(OBrienKaiser == OBK.W[, c(1:2, 4:18)])

```

[1] TRUE  
In more complex data sets, specialized software, such as the `reshape2` package (Wickham, 2007), may facilitate reshaping the data.

## 2.4 Working With Matrices, Arrays, and Lists

We have thus far encountered and used several data structures in R:

- *Vectors*: One-dimensional objects containing numbers, character strings, or logical values.<sup>34</sup> Single numbers, character strings, and logical values in R are treated as vectors of length one.
- *Factors*: One-dimensional objects representing categorical variables.
- *Data frames*: Two-dimensional data tables, with the rows defining cases and the columns defining variables. Data frames are heterogeneous, in the sense that some columns may be numeric and others factors, and some columns may even contain character data or logical data.

<sup>34</sup> Vectors in R are more general than this and can include data of any *mode*. For example, a vector of mode "list" is a list, a data structure discussed later in this section. Simple vectors of mode "numeric", "character", or "logical" are termed *atomic*. See help ("vector") for details.

In this section, we describe three other common data structures in R: *matrices*,

*arrays*, and *lists*.

### 2.4.1 Matrices

Matrices play a central role in the mathematics of statistics because numeric data arranged in rows and columns are so common. R has convenient and intuitive tools for working with matrices.

A matrix in R is a two-dimensional array of elements, all of which are of the same *mode*, usually numbers, but matrices of character strings or logical values are also supported.<sup>35</sup> Matrices can be constructed using the `matrix()` function, which reshapes its first argument into a matrix with the specified number of rows and columns, supplied as the second and third arguments, `nrow` and `ncol`. For example:

```
(A <- matrix(1:12, nrow=3, ncol=4))
```

```
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
(B <- matrix(c("a", "b", "c"), 4, 3,
byrow=TRUE)) # 4 rows, 3 columns
```

```
      [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"
```

<sup>35</sup> Modes in R are more complicated than this statement implies, but we need not get into the details here; see `help("mode")` and the help pages linked to it for additional information.

A matrix is filled by columns, unless the optional argument `byrow` is set to `TRUE`. The second example illustrates that if there are fewer elements in the first argument than are required to fill the matrix, then the elements are simply *recycled*, that is, extended by repetition to the required length.

A defining characteristic of a matrix is that it has a `dim` (dimension) *attribute* with two elements, the number of rows and the number of columns:

```
dim (A)
```

```
[1] 3 4
```

```
dim (B)
```

```
[1] 4 3
```

As mentioned, a vector is a one-dimensional object containing numbers or other elements. For example, here is a vector with a random permutation of the first 10 integers:

```
set.seed (54321) # for reproducibility
```

```
(v <- sample (10, 10)) # permutation of 1 to 10
```

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

A vector has a *length* but not a `dim` attribute:

```
length (v)
```

```
[1] 10
```

```
dim (v)
```

```
NULL
```

R often treats vectors differently than matrices. You can turn a vector into a one-column matrix using the `as.matrix ()` *coercion* function:

```
as.matrix (v)
```

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

You can also use the `matrix ()` function to create a one-row matrix:

```
matrix(v, nrow=1)
 [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,] 5 10 2 8 7 9 1 4 3 6
```

R includes extensive facilities for matrix computations, some of which are described in [Section 10.3](#).

## 2.4.2 Arrays

Higher-dimensional arrays of elements of the same mode are used much less frequently than matrices in statistical applications. They can be created with the array () function, here a three-dimensional array:

```
(array.3 <- array(1:24,
  dim=c(4, 3, 2))) # 4 rows, 3 columns, 2 layers
, , 1
[,1] [,2] [,3]
[1,] 1 5 9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
, , 2
[,1] [,2] [,3]
[1,] 13 17 21
[2,] 14 18 22
[3,] 15 19 23
[4,] 16 20 24
```

The order of the dimensions is row, column, and layer. The array is filled with the index of the first dimension changing most quickly: here row, then column, then layer.

## 2.4.3 Lists

Lists are data structures composed of potentially heterogeneous elements. The

elements of a list may be complex data structures, including other lists. Because a list can contain other lists as elements, each of which can also contain lists, lists are *recursive* structures. In contrast, the elements of a simple vector, such as an individual number, character string, or logical value, are *atomic* objects. Here is an example of a list, constructed by the `list()` function:

```
(list.1 <- list(mat.1=A, mat.2=B, vec=v)) # a 3-item list

$mat.1
 [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

$mat.2
 [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"

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

This list contains a numeric matrix, a character matrix, and a numeric vector. In this example, we name the arguments in the call to the `list()` function; the names are arbitrary and optional, not standard arguments to `list()`. If they are supplied, as here, the argument names become the names of the list elements.<sup>36</sup>

<sup>36</sup> It's similarly possible to supply element names as optional arbitrary arguments to the `c()` function, as in `c(a=1, b=2)`.

Because lists permit us to collect related information regardless of its form, they provide the foundation for the class-based S3 object system in R.<sup>37</sup> Data frames, for example, are lists with some special properties that make them behave somewhat like matrices, in that they can be indexed by rows and columns (as described in the next section).

<sup>37</sup> Classes and object-oriented programs in R are described in [Sections 1.7](#) and [10.9](#).

Like all R objects, matrices, arrays, and lists can be saved using the `save()`

function (described in [Section 2.1.5](#) for saving a data frame). These objects can then be read back into R using `load()`.

## 2.4.4 Indexing

A common operation in R is to extract one or more of the elements of a vector, matrix, array, list, or data frame by supplying the *indices* of the elements to be extracted. Indices are specified between square brackets, [ and ]. We have already used this syntax on several occasions, and it is now time to consider indexing more systematically.

### Indexing Vectors

A vector can be indexed by a single number or by a vector of numbers. Indices may be specified out of order, and an index may be repeated to extract the corresponding element more than once:

```
v # previously defined, permutation of 1:10
[1] 5 10 2 8 7 9 1 4 3 6

v[2] # second element
[1] 10

v[c(4, 2, 6)] # selected out of order
[1] 8 10 9

v[c(4, 2, 4)] # selecting 4th element twice
[1] 8 10 8
```

Specifying *negative* indices suppresses the corresponding elements of the vector:

```
v[-c(2, 4, 6, 8, 10)] # equivalent to v[c(1, 3, 5, 7, 9)]
[1] 5 2 7 1 3
```

If a vector has a `names` attribute, then we can also index the elements by name:<sup>38</sup>

```
names(v) <- letters[1:10]
```

```
v
```

|   |    |   |   |   |   |   |   |   |   |
|---|----|---|---|---|---|---|---|---|---|
| a | b  | c | d | e | f | g | h | i | j |
| 5 | 10 | 2 | 8 | 7 | 9 | 1 | 4 | 3 | 6 |

```
v[c("f", "i", "g")]
```

|   |   |   |
|---|---|---|
| f | i | g |
|---|---|---|

|   |   |   |
|---|---|---|
| 9 | 3 | 1 |
|---|---|---|

[38](#) The built-in vector letters contains the 26 lowercase letters from "a" to "z"; LETTERS similarly contains the uppercase letters.

Finally, a vector may be indexed by a logical vector of the same length, retaining the elements corresponding to TRUE and omitting those corresponding to FALSE:

```
v < 6
```

|      |       |      |       |       |       |      |      |      |       |
|------|-------|------|-------|-------|-------|------|------|------|-------|
| a    | b     | c    | d     | e     | f     | g    | h    | i    | j     |
| TRUE | FALSE | TRUE | FALSE | FALSE | FALSE | TRUE | TRUE | TRUE | FALSE |

```
v[v < 6] # all elements that are less than 6
```

|   |   |   |   |   |
|---|---|---|---|---|
| a | c | g | h | i |
| 5 | 2 | 1 | 4 | 3 |

Any of these forms of indexing may be used on the left-hand side of the assignment operator to replace the elements of a vector, an unusual and convenient feature of the R language. For example:

```

(vv <- v)  # make a copy of v

a   b   c   d   e   f   g   h   i   j
5  10  2   8   7   9   1   4   3   6

vv[c(1, 3, 5)] <- 1:3  # replace elements 1, 3, 5

vv

a   b   c   d   e   f   g   h   i   j
1  10  2   8   3   9   1   4   3   6

vv[c("b", "d", "f", "h", "j")] <- NA

vv

a   b   c   d   e   f   g   h   i   j
1  NA  2  NA  3  NA  1  NA  3  NA

```

**remove(vv) # clean up**

## Indexing Matrices and Arrays

Indexing extends straightforwardly to matrices and to higher-dimensional arrays. Indices corresponding to the different dimensions of an array are separated by commas; if the index for a dimension is left unspecified, then *all* the elements along that dimension are selected. We demonstrate with the matrix A:

**A # previously defined**

|       | [,1] | [,2] | [,3] | [,4] |
|-------|------|------|------|------|
| [1, ] | 1    | 4    | 7    | 10   |
| [2, ] | 2    | 5    | 8    | 11   |
| [3, ] | 3    | 6    | 9    | 12   |

```

A[2, 3] # element in row 2, column 3
[1] 8

A[c(1, 2), 2] # rows 1 and 2, column 2 (returns a vector)
[1] 4 5

A[c(1, 2), c(2, 3)] # rows 1 and 2, columns 2 and 3 (submatrix)
[,1] [,2]
[1,]    4    7
[2,]    5    8

A[1:2, ] # rows 1 and 2, all columns
[,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11

```

The second example above, A[2, 3], returns a single-element vector rather than a  $1 \times 1$  matrix; likewise, the third example, A[c (1, 2), 2], returns a vector with two elements rather than a  $2 \times 1$  matrix. *More generally, in indexing a matrix or array, dimensions of extent one are automatically dropped.* In particular, if we select elements in a single row or single column of a matrix, then the result is a vector, not a matrix with a single row or column, a convention that occasionally produces problems in R computations. We can override this default behavior with the argument drop=FALSE, however:

```

A[ , 2] # produces a vector
[1] 4 5 6

A[ , 2, drop=FALSE] # produces a one-column matrix
[,1]
[1,]    4
[2,]    5
[3,]    6

```

The row index is missing in both of these examples and is therefore taken to be all rows of the matrix. For clarity, we prefer to leave a space after an indexing comma, for example, A[, 2] rather than A[,2], and we sometimes leave a space for an empty index as well, A[, 2].

Negative indices, row or column names (if they are defined), and logical vectors of the appropriate length may also be used to index a matrix or a higher-dimensional array:

```
A[, -c(1, 3)] # omit columns 1 and 3

[,1] [,2]
[1,]    4   10
[2,]    5   11
[3,]    6   12

A[-1, -2] # omit row 1 and column 2
[,1] [,2] [,3]
[1,]    2   8   11
[2,]    3   9   12

rownames(A) <- c("one", "two", "three") # set row names
colnames(A) <- c("w", "x", "y", "z")      # set column names
A

      w  x  y  z
one   1  4  7 10
two   2  5  8 11
three 3  6  9 12

A[c("one", "two"), c("x", "y")]

      x  y
one 4  7
two 5  8

A[c(TRUE, FALSE, TRUE), ] # select 1st and 3rd rows

      w  x  y  z
one   1  4  7 10
three 3  6  9 12
```

Used on the left of the assignment arrow, we can replace indexed elements in a matrix or array:

```

(AA <- A)      # make a copy of A

      w  x  y  z
one   1  4  7 10
two   2  5  8 11
three 3  6  9 12

AA[1, ] <- 0  # set first row to zeros
AA[2, 3:4] <- NA
AA

```

|       | w | x | y  | z  |
|-------|---|---|----|----|
| one   | 0 | 0 | 0  | 0  |
| two   | 2 | 5 | NA | NA |
| three | 3 | 6 | 9  | 12 |

### ***remove (AA)***

### **Indexing Lists**

Lists may be indexed in much the same manner as vectors, but some special considerations apply. Recall the list that we constructed earlier:

***list.1 # previously defined***

```

$mat.1
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11

```

```
[3,]      3      6      9     12
```

\$mat.2

```
 [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"
```

\$vec

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

Here are some straightforward examples:

```

list.1[c(2, 3)] # elements 2 and 3

$mat.2
      [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"

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

list.1[2]          # returns a one-element list

```

```

$mat.2
      [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"

```

Even when we select just one element from a list, as in the last example, we get a single-element list rather than a matrix. To extract the *matrix* in position two of the list, we can use *double-bracket notation*, [[ ]]:

```
list.1[[2]] # returns a matrix
```

```

      [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "a"  "b"  "c"
[3,] "a"  "b"  "c"
[4,] "a"  "b"  "c"

```

The distinction between a one-element list and the element itself (here, a matrix) can be important if the extracted element of the list is to be used in other computations.

If the list elements are named, then we can use the names in indexing the list:

```
list.1["mat.1"] # produces a one-element list
```

```
$mat.1  
[,1] [,2] [,3] [,4]  
[1,] 1 4 7 10  
[2,] 2 5 8 11  
[3,] 3 6 9 12
```

```
list.1[["mat.1"]] # extracts a single element
```

```
[,1] [,2] [,3] [,4]  
[1,] 1 4 7 10  
[2,] 2 5 8 11  
[3,] 3 6 9 12
```

An element name may also be used either quoted or, if it is a legal R name, unquoted, after the \$ (dollar sign) to extract a list element:

```
list.1$mat.1
```

```
[,1] [,2] [,3] [,4]  
[1,] 1 4 7 10  
[2,] 2 5 8 11  
[3,] 3 6 9 12
```

Used on the left-hand side of the assignment arrow, dollar sign indexing allows us to replace list elements, define new elements, or delete an element by assigning NULL to the element:

```
list.1$mat.1 <- matrix(7:10, 2, 2)      # replace element
list.1$title <- "an arbitrary list"     # new element
list.1$mat.2 <- NULL                    # delete element
list.1
```

```
$mat.1
 [,1] [,2]
[1,]    7    9
[2,]    8   10
```

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

```
$title
[1] "an arbitrary list"
```

Setting a list element to NULL is trickier:

```
list.1["title"] <- list(NULL)
```

```
list.1
```

```
$mat.1
 [,1] [,2]
[1,]    7    9
[2,]    8   10
```

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

```
$title
```

```
NULL
```

Once a list element is extracted, it may itself be indexed; for example:

```
list.1$vec[3]
```

```
[1] 2
```

```
list.1[["mat.1"]][2, 1]
```

```
[1] 8
```

Finally, extracting a *nonexistent* element returns NULL:

```
list.1$foo
```

```
NULL
```

This behavior is potentially confusing because it is not possible to distinguish by the value returned between a NULL element, such as list.1\$title in the example, and a nonexistent element, such as list.1\$foo.

## Indexing Data Frames

Data frames may be indexed either as lists or as matrices. Recall the Guyer data frame:

***brief(Guyer)***

```
20 x 3 data.frame (15 rows omitted)
  cooperation condition sex
            [n]      [f]      [f]
  1           49 public   male
  2           64 public   male
  3           37 public   male
  . . .
  19          34 anonymous female
  20          44 anonymous female
```

The row names of the Guyer data set are the character representation of the row numbers:

***rownames(Guyer)***

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

Indexing Guyer as a matrix:

```
Guyer[, 1] # first column, returned as a vector
[1] 49 64 37 52 68 54 61 79 64 29 27 58 52 41 30 40 39 44 34 44
```

```

Guyer[ , "cooperation"] # equivalent
[1] 49 64 37 52 68 54 61 79 64 29 27 58 52 41 30 40 39 44 34 44

Guyer[1:3, ] # first 3 rows
  cooperation condition sex
1          49    public male
2          64    public male
3          37    public male

Guyer[c("1", "2"), "cooperation"] # by row and column names
[1] 49 64

Guyer[-(6:20), ] # drop rows 6 through 20
  cooperation condition sex
1          49    public male
2          64    public male
3          37    public male
4          52    public male
5          68    public male

with(Guyer, Guyer[sex == "female" & condition == "public", ])
  cooperation condition sex
6          54    public female
7          61    public female
8          79    public female
9          64    public female
10         29   public female

```

The `with()` function is required in the last example to access the variables `sex` and `condition`, because the `Guyer` data frame is not attached to the search path. More conveniently, we can use the `subset()` function to perform this operation:

```
subset(Guyer, sex == "female" & condition == "public")
```

|    | cooperation | condition | sex    |
|----|-------------|-----------|--------|
| 6  | 54          | public    | female |
| 7  | 61          | public    | female |
| 8  | 79          | public    | female |
| 9  | 64          | public    | female |
| 10 | 29          | public    | female |

Alternatively, indexing the Guyer data frame as a list:

```
Guyer$cooperation
```

```
[1] 49 64 37 52 68 54 61 79 64 29 27 58 52 41 30 40 39 44 34 44
```

```
Guyer[["cooperation"]] # equivalent
```

```
[1] 49 64 37 52 68 54 61 79 64 29 27 58 52 41 30 40 39 44 34 44
```

```
Guyer[[1]] # equivalent
```

```
[1] 49 64 37 52 68 54 61 79 64 29 27 58 52 41 30 40 39 44 34 44
```

```
head(Guyer["cooperation"]) # first six rows
```

cooperation

|   |    |
|---|----|
| 1 | 49 |
| 2 | 64 |
| 3 | 37 |
| 4 | 52 |
| 5 | 68 |
| 6 | 54 |

Specifying Guyer["cooperation"] (or Guyer[1], not shown) returns a one-column data frame rather than a vector.

## 2.5 Dates and Times

The standard R system includes functions for manipulating date and time data (see, e.g., `help("DateTimeClasses")`), and there are several packages available from CRAN for working with dates and times.<sup>39</sup> Date and time data are typically initially encoded in character strings of varying formats, and the process of converting these character strings to date and time objects can be confusing. For most users, the straightforward functions in the tidyverse **lubridate** package

(Grolemund & Wickham, 2011) for converting character data into dates and times, along with the date and time data classes in base R, are all that are needed. Date and time variables can be included as columns in data frames, as we illustrate in this section.

[39](#) See in particular the packages discussed in the Time Series Analysis CRAN task view.

Consider first two character vectors representing dates encoded in different formats:

```
c.starts <- c ("02/27/2017", "02/24/2016", "05/14/1984")  
c.ends <- c ("27-03-17", "3-1-17", "1-11-85")
```

The vector c.starts includes three dates in the common U.S. date format of month-day-year, while c.ends has dates in the day-month-year format common in Canada, Europe, and elsewhere. In addition, c.starts uses four-digit years, while c.ends uses two-digit years.

Transparently named function in the **lubridate** package can be used to convert these character vectors to "Date" objects:

```
library("lubridate")  
(Dates <- data.frame(starts=mdy(c.starts), ends=dmy(c.ends)))  
  
      starts        ends  
1 2017-02-27 2017-03-27  
2 2016-02-24 2017-01-03  
3 1984-05-14 1985-11-01
```

The function mdy () expects dates in the month-day-year format and automatically removes any separators, while dmy () uses the day-month-year format. The functions guess at the right century for two-digit years, currently interpreting years 69–99 as 1969–1999, and years 00–68 as 2000–2068. If your data span a longer time period, use four-digit years to avoid the dreaded “Y2K problem.” The **lubridate** package has additional similar functions to read dates in other formats; for example, ymd () expects dates in year-month-day format, such as "2017-12-19".

You can conveniently perform arithmetic on dates; for example,

```
with(Dates, ends - starts)
```

Time differences in days

```
[1] 28 314 536
```

computes the number of days between corresponding pairs of dates. You can also compute a variety of other information from dates:

```
weekdays(Dates$starts, abbreviate=TRUE)
```

```

[1] "Mon" "Wed" "Mon"
months (Dates$starts, abbreviate=FALSE)
[1] "February" "February" "May"
quarters (Dates$starts, abbreviate=TRUE)
[1] "Q1" "Q1" "Q2"
as.numeric (format (Dates$starts, "%Y"))
[1] 2017 2016 1984
as.numeric (format (Dates$starts, "%m"))
[1] 2 2 5

```

See help ("months") for complete descriptions of the first three functions. The format () function for dates is documented in help ("strptime"), which provides comprehensive information about working with dates and times. In the example, we use format () to convert the dates in starts to numeric years and months, where "%Y" signifies conversion to four-digit years, and "%m" signifies months. Time data can also be processed by **lubridate** functions, usually using hms (), hm (), or ms (), with h representing hours, m minutes, and s seconds:

```

(time1 <- hms ("17:5:3"))
[1] "17H 5M 3S"
(time2 <- hm ("7:4"))
[1] "7H 4M 0S"
(time3 <- ms ("7:4"))
[1] "7M 4S"
time1 - time2
[1] "10H 1M 3S"
hour (time1)
[1] 17
minute (time1)
[1] 5
second (time1)
[1] 3

```

Finally, there are functions in **lubridate** to handle character data that combine dates and times; for example:

```

(date.time <- mdy_hms ("7/10/18 23:05:01"))
[1] "2018-07-10 23:05:01 UTC"

```

The date and time variable created is of class "POSIXct", a class supported by base R.<sup>40</sup>

<sup>40</sup> UTC represents “Universal Coordinated Time,” which, although there is a subtle distinction between the two, is for practical purposes equivalent to Greenwich Mean Time (GMT). R can make allowance for time zones in time

data, but the topic is beyond the current discussion; see help ("POSIXct"). The MplsStops data set on police stops in Minneapolis, introduced in [Section 2.3.5](#), includes the variable date, the date and time of each stop. Stops are of two types according to the level of the variable problem, either "traffic", for a traffic stop, or "suspicious", for any other type of stop. We can, for example, count the number of stops of each type by day of the week, comparing the two distributions by converting the counts to percentages:

```
MplsStops$wkday <- factor(weekdays(MplsStops$date,
                                         abbreviate=TRUE),
                            levels=c("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"))
(tabl <- xtabs(~ problem + wkday, data=MplsStops))

      wkday
problem      Sun Mon Tue Wed Thu Fri Sat
suspicious 3021 3605 3975 3844 3783 4027 3567
traffic     2561 3118 3833 4219 3967 4788 3612

100*round(prop.table(tabl, margin=1), 3) # row percents

      wkday
problem      Sun Mon Tue Wed Thu Fri Sat
suspicious 11.7 14.0 15.4 14.9 14.7 15.6 13.8
traffic     9.8 11.9 14.7 16.2 15.2 18.3 13.8

chisq.test(tabl)
```

Pearson's Chi-squared test

```
data: tabl
X-squared = 162, df = 6, p-value <2e-16
```

We specify the levels for the factor wkday explicitly to avoid the default alphabetic ordering; the argument margin=1 to prop.table () calculates proportions along the first coordinate of the table, producing row percentages when we multiply by 100. The small  $p$ -value for the chi-square test of independence suggests that the weekly patterns for "suspicious" and "traffic" stops are different, but the percentage table reveals that the differences are not terribly large.

We can similarly classify stops by when in the day they occur:

```

MplsStops$daynight <- with(MplsStops,
  ifelse(hour(date) < 6 | hour(date) >= 18,
    "night", "day"))
(tab2 <- xtabs(~ problem + daynight, MplsStops))

      daynight
problem      day night
  suspicious 11610 14212
  traffic    10609 15489

100*round(prop.table(tab2, margin=1), 3)

      daynight
problem      day night
  suspicious 45.0 55.0
  traffic    40.7 59.3

chisq.test(tab2)

```

Pearson's Chi-squared test with Yates' continuity correction

```

data: tab2
X-squared = 98.4, df = 1, p-value <2e-16
There are therefore relatively more "suspicious" than "traffic" stops during the day.
These few tools are adequate for most applications of dates and times, but if you have data from different time zones or want to perform fancier operations on dates or times, you may need to know more. The vignette in the lubridate package is a good place to start.

```

## 2.6 Character Data

Handling text is an underappreciated capability of R, and using R for this purpose is a viable alternative to specialized text-processing tools, such as the PERL scripting language and the Unix utilities sed, grep, and awk. Just like these other tools, most of the text-processing functions in R make use of *regular expressions* for matching text in character strings. In this section, we provide a brief introduction to manipulating character data in R, primarily by example. More complete information may be found in the online help for the various text-

processing functions; in `help ("regexp")`, which describes how regular expressions are implemented in R; and in the sources cited at the end of the chapter.

We'll turn to the familiar "To Be or Not to Be" soliloquy from Shakespeare's *Hamlet* in the plain-text file `Hamlet.txt`, which you downloaded in [Chapter 1](#) and which resides in the RStudio R-Companion project directory. We begin by using the `readLines ()` function to input the lines of the file into a character vector, one line per element:

```
Hamlet <- readLines("Hamlet.txt") # from the current directory
head(Hamlet)      # first 6 lines

[1] "To be, or not to be: that is the question:"
[2] "Whether 'tis nobler in the mind to suffer"
[3] "The slings and arrows of outrageous fortune,"
[4] "Or to take arms against a sea of troubles,"
[5] "And by opposing end them? To die: to sleep; "
[6] "No more; and by a sleep to say we end"

length(Hamlet)      # number of lines

[1] 35

nchar(Hamlet)      # number of characters per line

[1] 42 41 44 42 43 37 46 42 40 50 47 43 39 36 48 49 44 38 41 38
[21] 43 38 44 41 37 43 39 44 37 47 40 42 44 39 26

sum(nchar(Hamlet)) # number of characters in all

[1] 1454
```

The `length ()` function counts the number of character strings in the character vector `Hamlet`—that is, the number of lines in the soliloquy—while the `nchar ()` function counts the number of characters, including blanks, in each string—that is, in each line. We then use `sum ()` to compute the total number of characters in the data.

The `paste ()` function is useful for joining character strings into a single string. For example, to join the first six lines:

```
(lines.1_6 <- paste(Hamlet[1:6], collapse = " " ))
```

"To be, or not to be: that is the question ... to say we end"

Here and elsewhere in this section, we've edited the R output where necessary so that it fits properly on the page (with the ellipses ... representing elided text). Alternatively, we can use the `strwrap ()` function to "wrap" the text into lines of

approximately equal length, which once again divides the text into separate character strings (but not strings identical to the original lines in Hamlet.txt):

***strwrap (lines.1\_6)***

- [1] "To be, or not to be: that is the question: Whether 'tis"
- [2] "nobler in the mind to suffer The slings and arrows of"
- [3] "outrageous fortune, Or to take arms against a sea of"
- [4] "troubles, And by opposing end them? To die: to sleep; No"
- [5] "more; and by a sleep to say we end"

By default, the “target” number of characters for each line is 90% of the current character width of R output.<sup>41</sup>

[41](#) Output width in RStudio adjusts to the size of the *Console* and in an R Markdown document to the width of the text. To discover the current output width, enter the command `getOption ("width")`; to reset the output width, enter the command `options (width=nn)`, where *nn* is the desired width in characters—for example, `options (width=64)`, which is the width of R output in the *R Companion*.

The `substring ()` function, as its name implies, selects part of a character string—for example, to select the characters 22 through 41 in the text of the first six lines of the soliloquy:

***substring (lines.1\_6, 22, 41)***

- [1] "that is the question"

Suppose that we want to divide `lines.1_6` into words. As a first approximation, let’s split the character string at the blanks:

***strsplit (lines.1\_6, " ")***

```
[[1]]  
[1] "To"           "be,"          "or"           "not"  
[5] "to"           "be:"          "that"         "is"  
[9] "the"          "question:"   "Whether"      "'tis"  
[13] " "             " "             " "             "  
[17] " "             " "             " "             "  
[21] " "             " "             " "             "  
[25] " "             " "             " "             "  
[29] " "             " "             " "             "  
[33] " "             " "             " "             "  
[37] " "             " "             " "             "  
[41] " "             " "             " "             "  
[45] " "             " "             " "             "  
[49] "sleep"       "to"          "say"          "we"  
[53] "end"
```

The string-splitting function `strsplit ()` takes two required arguments, the first of which is a character vector of strings to be split. In this example, `lines.1_6` is a

vector of length one containing the first six lines of the soliloquy. The second argument is a quoted *regular expression* specifying a pattern that determines where splits will take place. In the example, the regular expression contains the single character " " (space). Most characters in regular expressions, including spaces, numerals, and lowercase and uppercase alphabetic characters, simply match themselves.

The strsplit () function returns a list, with one element corresponding to each element of its first argument. Because the input in the example, lines.1\_6, is a *single* character string, the output is a one-element list: Recall from [Section 2.4.4](#) the distinction between a one-element list and the element itself (here, a vector of character strings). In the other calls to the strsplit () function shown below, we append [[1]] to extract the first (and only) element of the returned list—a character vector containing several character strings.

Our first attempt at dividing lines.1\_6 into words isn't entirely successful: Perhaps we're willing to live with words such as "tis" (which is a contraction of "it is"), but it would be nice to remove the punctuation from "be,", "be:" and "them?" (the last of which isn't shown in the partial printout above), for example.

Characters enclosed in square brackets in a regular expression represent alternatives; thus, for example, the regular expression "[,;.:?!]" will match a space, comma, semicolon, colon, period, question mark, or exclamation point. Because some words are separated by more than one of these characters (e.g., a colon followed by a space), we may add the *quantifier* + (plus sign) immediately to the right of the closing bracket; the resulting regular expression, "[,;.:?!]+", will match *one or more* adjacent spaces and punctuation characters:

```
strsplit(lines.1_6, "[ ,;.:?!]+") [[1]]
```

```
[1] "To"          "be"          "or"          "not"  
[5] "to"          "be"          "that"        "is"  
[9] "the"         "question"    "Whether"     "'tis"  
[13] " "           " "           " "           "  
[17] " "           " "           " "           "  
[21] " "           " "           " "           "  
[25] " "           " "           " "           "  
[29] " "           " "           " "           "  
[33] " "           " "           " "           "  
[37] " "           " "           " "           "  
[41] " "           " "           " "           "  
[45] " "           " "           " "           "  
[49] "sleep"      "to"          "say"         "we"  
[53] "end"
```

Other quantifiers in regular expressions, which are examples of so-called *meta-characters*, include \* (asterisk), which matches the preceding expression *zero or*

more times, and ? (question mark), which matches the preceding expression zero or one time. We can, for example, divide the text into sentences by splitting at any of., ?, or !, followed by zero or more spaces:

```
strsplit(lines.1_6, "[.?!]*")[[1]]
```

- [1] "To be, or not to be: that is the question: ... end them"
- [2] "To die: to sleep; No more; and by a sleep to say we end"

We can divide the text into individual characters by splitting at the *empty string*, "":

```
characters <- strsplit(lines.1_6, "")[[1]]  
length(characters)      # number of characters  
[1] 254  
  
head(characters, 20)  # first 20 characters  
  
[1] "T" "o" " " "b" "e" ", " " " "o" "r" " " "n" "o" "t" " " "t"  
[16] "o" " " "b" "e" ":"
```

Returning to the whole soliloquy, we divide the text into words at spaces, a strategy that, as we have seen, is flawed:

```
all.lines <- paste(Hamlet, collapse=" ")  
words <- strsplit(all.lines, " ")[[1]]  
length(words)      # number of words  
[1] 277  
  
head(words, 20)  # first 20 words  
  
[1] "To"          "be,"         "or"          "not"         "to"  
[6] "be:"         "that"        "is"          "the"         "question:  
[11] "Whether"     "'tis"        "nobler"      "in"          "the"  
[16] "mind"        "to"          "suffer"      "The"         "slings"
```

We can fix the words that have extraneous punctuation by substituting the empty string for the punctuation, using the sub () (substitute) function:

```
words <- sub("[,;.:?!]", "", words)  
head(words, 20)  
  
[1] "To"          "be"          "or"          "not"         "to"  
[6] "be"          "that"        "is"          "the"         "question"  
[11] "Whether"     "'tis"        "nobler"      "in"          "the"  
[16] "mind"        "to"          "suffer"      "The"         "slings"
```

The sub () function takes three required arguments: (1) a regular expression

matching the text to be replaced; (2) the replacement text, here the empty string; and (3) a character vector in which the replacement is to be performed. If the pattern in the regular expression matches more than one substring in an element of the third argument, then only the first occurrence is replaced. The `gsub()` function behaves similarly, except that *all* occurrences of the pattern are replaced:

```
sub ("me", "you", "It's all, 'me, me, me' with you!")
```

```
[1] "It's all, 'you, me, me' with you!"
```

```
gsub ("me", "you", "It's all, 'me, me, me' with you!")
```

```
[1] "It's all, 'you, you, you' with you!"
```

Returning once more to the soliloquy, suppose that we want to determine and count the different words that Shakespeare used. A first step is to use the `tolower()` function to change all the characters to lowercase, so that, for example, "the" and "The" aren't treated as distinct words:

```
head(words <- tolower(words), 20) # first 20 words
```

|      |           |        |          |       |            |
|------|-----------|--------|----------|-------|------------|
| [1]  | "to"      | "be"   | "or"     | "not" | "to"       |
| [6]  | "be"      | "that" | "is"     | "the" | "question" |
| [11] | "whether" | "'tis" | "nobler" | "in"  | "the"      |
| [16] | "mind"    | "to"   | "suffer" | "the" | "slings"   |

```
word.counts <- sort(table(words), decreasing=TRUE)
```

```
word.counts[word.counts > 2] # words used more than twice
```

words

|     |    |    |      |      |   |       |    |    |      |
|-----|----|----|------|------|---|-------|----|----|------|
| the | of | to | and  | that | a | sleep | be | we | bear |
| 22  | 15 | 15 | 12   | 7    | 5 | 5     | 4  | 4  | 3    |
| in  | is | us | with |      |   |       |    |    |      |
| 3   | 3  | 3  | 3    |      |   |       |    |    |      |

```
head(sort(unique(words)), 20) # first 20 unique words
```

|      |           |         |        |          |            |
|------|-----------|---------|--------|----------|------------|
| [1]  | "'tis"    | "_"     | "a"    | "action" | "after"    |
| [6]  | "against" | "all"   | "and"  | "arms"   | "arrows"   |
| [11] | "awry"    | "ay"    | "bare" | "be"     | "bear"     |
| [16] | "bodkin"  | "bourn" | "but"  | "by"     | "calamity" |

```
length(unique(words)) # number of unique words
```

```
[1] 167
```

We use the `table()` command to obtain the word counts, `unique()` to remove duplicate words, and `sort()` to order the words from the most to the least used and to arrange the unique words in alphabetical order. The alphabetized words reveal a problem: We're treating the hyphen (“-”) as if it were a word.

The `grep()` function may be used to search character strings for a regular expression, returning the indices of the strings in a character vector for which there is a match.<sup>42</sup> For our example,

```
grep ("-", words)
[1] 55 262
words[grep ("-", words)]
[1] "heart-ache" "-"
```

[42](#) `grep` is an acronym for “*g lobally search a regular expression and print*” and is the name of a standard Unix command-line utility.

We find matches in two character strings: the valid, hyphenated word “heartache” and the spurious word “-”. We would like to be able to differentiate between the two, because we want to discard the latter from our vector of words but retain the former. We can do so as follows:

```
grep ("^-", words)
[1] 262
words <- words[- grep("^-", words)] # negative index to delete
head(sort(unique(words)), 20)
[1] "'tis"      "a"        "action"    "after"     "against"
[6] "all"       "and"      "arms"      "arrows"    "awry"
[11] "ay"        "bare"     "be"        "bear"      "bodkin"
[16] "bourn"     "but"      "by"        "calamity"  "cast"
```

The meta-character `^` (caret) is an *anchor* representing the beginning of a text string, and thus, the regular expression “`^-`” only matches first-character hyphens. The meta-character `$` is similarly an anchor representing the end of a text string:

```
grep ("!$", c ("!10", "wow!"))
[1] 2
grep ("^and$", c ("android", "sand", "and", "random"))
[1] 3
grep ("and", c ("android", "sand", "and", "random"))
[1] 1 2 3 4
```

A hyphen (-) may be used as a meta-character within square brackets to

represent a range of characters, such as the digits from 0 through 9.<sup>43</sup> In the following command, for example, we pick out the elements of a character vector that are composed of numerals, periods, and minus signs:

```
data <- c ("-123.45", "three hundred", "7550", "three hundred 23",
"Fred")
data[grep ("^[-.0-9]*$", data)]
[1] "-123.45" "7550"
```

<sup>43</sup> Using the hyphen to represent ranges of characters can be risky, because character ranges can vary from one language locale to another—say between English and French. Thus, for example, we cannot rely on the range a-zA-Z to contain all the alphabetic characters that may appear in a word—it will miss the accented letters, for example. As a consequence, there are special character classes defined for regular expressions, including [:alpha:], which matches all the alphabetic characters in the current locale. Other frequently useful character classes include [:lower:] (lowercase characters), [:upper:] (uppercase characters), [:alnum:] (alphanumeric characters, i.e., the letters and numerals), [:space:] (white-space characters), and [:punct:] (punctuation characters).

The hyphen before the closing bracket represents itself and will match a minus sign.

Used after an opening square bracket, the meta-character “^” represents negation, and so, for example, to select elements of the vector data that *do not* contain any numerals, hyphens, or periods:

```
data[grep ("^[-.0-9]*$", data)]
[1] "three hundred" "Fred"
```

Parentheses are used for grouping in regular expressions, and the bar character () means *or*. To find all the articles in the soliloquy, for example:

```
words[grep ("^(the|a|an)$", words)]
[1] "the" "the" "the" "a" "a" "the" "the" "a" "the" "the"
[11] "the" "the" "the" "the" "the" "the" "the" "a" "a"
[21] "the" "the" "the" "the" "the" "the" "the"
```

To see why the parentheses are needed here, try omitting them. Similarly, to count all of the unique words in the soliloquy that *aren't* articles:

```
length(unique(words[- grep ("^(the|a|an)$", words)]))
[1] 164
```

What happens if we want to treat a meta-character as an ordinary character? We have to *escape* the meta-character by using a backslash (\), and because the backslash is the escape character for R as well as for regular expressions, we must, somewhat awkwardly, double it; for example:

```
grep ("\\$", c ("$100.00", "100 dollars"))
```

```
[1] 1
```

We clean up by deleting all of the objects created in the global environment thus far in the chapter:

```
remove (list=objects ())
```

## 2.7 Large Data Sets in R\*

R has a reputation in some quarters for choking on large data sets. This reputation is only partly deserved. We will explain in this section why very large data sets may pose a problem for R and suggest some strategies for dealing with such data sets.

The most straightforward way to write functions in R is to access data that reside in the computer's main memory—corresponding to the global environment (also called the R *workspace*). This is true, for example, of the statistical-modeling functions in the standard R distribution, such as the lm () function for fitting linear models (discussed in [Chapter 4](#)) and the glm () function for fitting generalized linear models ([Chapter 6](#)). The size of computer memory then becomes a limitation on the size of statistical analyses that can be handled successfully. The problem is exacerbated by R's tendency to make copies of objects during a computation, for example, when they are modified.<sup>44</sup>

[44](#) The R Core team has devoted considerable effort to decreasing unnecessary copying of objects, and so R has improved significantly in this area.

A computer with a 32-bit processor can't address more than 4 GB (gigabytes) of memory, and depending on the system, not all this memory may be available to R.<sup>45</sup> Almost all computers now have 64-bit processors and operating systems and can therefore address vastly larger amounts of memory, making analysis of very large data sets directly in memory more practical.

[45](#) One *gigabyte* is a little more than one billion bytes, and one *byte* is 8 *bits* (i.e., binary digits). R represents most numbers as *double-precision floating-point numbers*, which are stored in eight bytes (64 bits) each. In a data set composed of floating-point numbers, then, 4 GB corresponds to  $4 \times 1,024^3 / 8$ , or somewhat more than 500 million data values.

Handling large data sets in R is partly a matter of programming technique. Although it is *convenient* in R to store data in memory, it is not *necessary* to do so. There are many R programs designed to handle very large data sets, such as those used in genomic research, most notably the R packages distributed by the Bioconductor Project (at [www.bioconductor.org](http://www.bioconductor.org)). Similarly, the **biglm** package (Lumley, 2013) on CRAN has functions for fitting linear and generalized linear models that work serially on chunks of the data rather than all the data at once.

Moreover, some packages, such as **biglm** and the **survey** package for the analysis of data from complex sample surveys (Lumley, 2010), are capable of accessing data in database management systems.<sup>46</sup>

[46](#) R has more general tools for accessing databases, which can be an effective way of managing very large data sets. We won't address the use of databases in this book, but some of the references given at the end of the chapter take up the topic.

### 2.7.1 How Large Is “Large”?

Problems that are quite large by social science standards can be handled routinely in R and don't require any special care or treatment. In the following examples, we will fit linear least-squares and logistic regressions to a data set with one million cases and 100 explanatory variables. All this is done on a modest 64-bit Windows 10 laptop with 8 GB of memory.

We begin by constructing a model matrix for the regression by sampling 100 million values from the standard-normal distribution and then reshaping these values into a matrix with one million rows and 100 columns:<sup>47</sup>

```
set.seed(123456789) # for reproducibility
system.time(X <- rnorm(1e6*100))

      user   system elapsed
    7.68     0.11    7.79

X <- matrix(X, 1e6, 100)
str(X)

  num [1:1000000, 1:100] 0.505 0.396 1.416 -0.722 -0.618 ...
```

[47](#) The X matrix doesn't include a column of ones for the regression constant because the lm () and glm () functions employed below to fit linear and generalized linear models to the data automatically include the regression constant in the model.

Recall that  $1\text{e}6 = 1 \times 10^6$  is scientific notation for one million. Here, and elsewhere in this section, we use the system.time () function to time some computations; the results reported are in seconds, and the figure to which to attend is “elapsed time”—about 8 seconds in the computation above.

Under Windows, the memory.size () and memory.limit () functions allow us to check the amount of memory, in MB (*megabytes*—approximately one million bytes, or 0.001 of a GB), that we've used and the amount available to the R process, respectively:

```
memory.size()
```

```
[1] 826.55
```

```
memory.limit()
```

```
[1] 8076
```

We've thus far used about  $800 \text{ MB} = 0.8 \text{ GB}$ , which is approximately 10% of the memory available to R.

Next, we generate values of the response variable  $y$ , according to a linear regression model with normally distributed errors that have a standard deviation of 10:

```
system.time(y <- 10 + as.vector(X %*% rep(1, 100)
+ rnorm(1e6, sd=10)))
```

```
user system elapsed
```

```
0.36 0.00 0.36
```

In this expression, `%*%` is the matrix-multiplication operator (see [Section 10.3](#)), and the coercion function `as.vector()` is used to coerce the result to a vector, because matrix multiplication of a matrix by a vector in R returns a one-column matrix. The vector of population regression coefficients consists of ones—`rep(1, 100)`—and the regression intercept is 10.

To fit the regression model to the data,

```
system.time(m <- lm(y ~ X))
```

```
user system elapsed
```

```
18.41 1.20 19.63
```

```
head(coef(m), 5) # first 5 coefficients
```

|             | X1       | X2       | X3       | X4       |
|-------------|----------|----------|----------|----------|
| (Intercept) | 9.987159 | 1.017738 | 1.005440 | 1.005723 |
|             |          |          |          | 1.005806 |

```
memory.size()
```

```
[1] 2505.13
```

We print the first five regression coefficients: The intercept and the first four slopes are close to the population values of the coefficients, as we would expect in a sample of  $n = 1,000,000$  cases. The computation took about 20 seconds, and because the object returned by `lm()` contains a lot of information (including a copy of the original data to which the regression model was fit), we're now using 2.5 GB of memory.<sup>48</sup>

[48](#) It's possible to tell lm () to save less information, but we won't have to do that; see help ("lm").

For a logistic regression (see [Section 6.3](#)), we first generate the response vector yy of zeros and ones according to a model with an intercept of zero and slope coefficients of 1/4:

```
p <- as.vector(1/(1 + exp(-X %*% rep(0.25, 100))))  
summary(p)  
  
Min. 1st Qu. Median Mean 3rd Qu. Max.  
0.0000057 0.1556270 0.4993556 0.4998564 0.8438453 0.9999873  
  
system.time(yy <- rbinom(1e6, 1, prob=p))  
  
user system elapsed  
0.08 0.00 0.08  
  
table(yy)  
  
yy  
0 1  
501078 498922
```

The vector p gives probabilities, derived from the logistic-regression model, that the response  $y = 1$  for each of the one million cases, and rbinom () samples one value, either zero or 1, for each case with corresponding probability.[49](#) As the table () command reveals, we calibrated the values for this problem to produce approximately equal numbers of zeros and ones.

[49](#) Because the number of binomial trials is one for each case, the samples are each drawn from a Bernoulli distribution.

Then, to fit the model,

```
system.time(m <- glm(yy ~ X, family=binomial))  
  
user system elapsed  
96.87 5.23 104.74  
  
head(coef(m), 5) # first 5 coefficients  
  
(Intercept) X1 X2 X3 X4  
-0.007101478 0.250070838 0.251718299 0.247488511 0.249186890
```

Again, the computation goes through in a reasonable amount of time for the magnitude of the problem (105 seconds) and in the available memory. The results are as expected, with an estimated intercept near zero and slopes near

0.25.

The `memory.size()` function suggests, however, that we've used up most of the 8 GB of memory available to R:

```
memory.size()
```

```
[1] 6776.47
```

Using the `gc()` (garbage collection) function to free up memory shows that we still have plenty of memory available:

```
gc()
```

```
...
```

```
memory.size()
```

```
[1] 2530.51
```

Moreover, it isn't really necessary to perform garbage collection manually: It would have been done automatically by R in due course. We suppress the output produced by `gc()` because it's just of technical interest.

## 2.7.2 Reading and Saving Large Data Sets

In [Section 2.1.3](#), we suggested using `read.table()` to input data from white-space-delimited plain-text files into R, but `read.table()` can be very slow for reading large data sets. To provide a simple example, we create a data frame from our simulated data and export the data frame to a plain-text file with the `write.table()` function:

```
system.time(D <- data.frame(X, y, yy))
```

| user | system | elapsed |
|------|--------|---------|
| 0.73 | 0.81   | 1.98    |

```
dim(D)
```

```
[1] 1000000      102
```

```
object.size(D)
```

```

812011176 bytes

system.time(write.table(D,
  "largeData.txt")) # to the project directory

  user  system elapsed
 202.07   3.13 206.44

```

The data frame D consists of one million rows and 102 columns; the object.size() function reveals that it occupies about 800 MB of memory.<sup>50</sup> Using write.table() to save the data as a text file is time-consuming, taking about 3.5 minutes. To read the data back into memory from the text file using read.table() takes even longer (more than 4 minutes):

```

system.time(DD <- read.table("largeData.txt", header=TRUE))

  user  system elapsed
 258.41   2.40 265.02

```

<sup>50</sup> The object size of about 800 MB makes sense because, as we mentioned, each of the 102 million numbers in the data set occupies 8 bytes.

The read.table() function is slow partly because it has to figure out whether data should be read as numeric variables or as factors. To determine the class of each variable, read.table() reads *all* the data in character form and then examines each column, converting the column to a numeric variable or a factor, as required. We can make this process considerably more efficient by explicitly telling read.table() the class of each variable, via the colClasses argument:

```

system.time(DD <- read.table("largeData.txt", header=TRUE,
  colClasses=c("character", rep("numeric", 102)))))

  user  system elapsed
 71.12   1.33  72.70

```

Reading the data now takes a little more than a minute. The "character" class specified for the first column is for the row name in each line of the text file created by write.table(); for example, the name of the first row is "1" (with the quotation marks included). For more details about specifying the colClasses argument, see help("read.table").

The save() function, discussed in [Section 2.1.5](#), stores data frames and other R objects in an internal format, which can be reread using the load() function much more quickly than a plain-text data file. For our example, the time to read the data is reduced to only about 5 seconds (although writing the .RData file

takes a non-negligible amount of time):

```
system.time(save(D, file="D.RData"))  
  
       user   system elapsed  
 35.75    1.13   37.01  
remove("D")  
  
system.time(load("D.RData"))  
  
       user   system elapsed  
 4.90    0.33   5.24
```

**`dim(D)`**

```
[1] 1000000      102
```

Before continuing, we clean up the global environment:

```
remove(list=objects())
```

## 2.8 Complementary Reading and References

- The *R Data Import/Export* manual, which is part of the standard R distribution and is accessible in the RStudio *Help* tab (press the *Home* icon at the top of the tab), provides a great deal of information about getting data into and out of R, including on database interfaces. For a slightly dated introduction to the latter topic, see Ripley (2001), and for more up-to-date information, see Chambers (2016).
- The various contributed packages in the “tidyverse” provide alternative integrated methods for reading, storing, and manipulating data, addressing most of the topics discussed in this chapter. See Wickham (2017) and Wickham and Grolemund (2016).
- Spector (2008) provides a reasonably broad treatment of data management in R, a dry topic but one that is vital for the effective day-to-day use of R (or of any statistical data analysis system).
- Munzert, Rubba, Meißner, and Nyhuis (2014) discuss *data scraping* in R, methods for gathering data from a wide variety of sources on the internet and then combining them into a usable form.
- Both Gentleman (2009, chap. 5) and Chambers (2008, chap. 8) include extensive discussions of manipulating character data in R. Further information about regular expressions is available in many sources,

including the *Wikipedia* (at  
[http://en.wikipedia.org/wiki/Regular\\_expression](http://en.wikipedia.org/wiki/Regular_expression)).

- *Text mining* turns words into data for statistical analysis; see Feinerer, Hornik, and Meyer (2008) and Feinerer and Hornik (2017) for discussion of several packages for text mining with R.

# 3 Exploring and Transforming Data

John Fox & Sanford Weisberg

Statistical graphs play three important roles in data analysis. Graphs provide an initial look at the data, a step that is skipped at the peril of the data analyst. At this stage, we learn about the data, their oddities, outliers, and other interesting features. John Tukey (1977) coined the term *exploratory data analysis* for this phase of an analysis. A skilled analyst also draws graphs during the model-building phase of a data analysis, particularly in diagnostic methods used to assess the adequacy of a model fit to the data. Finally, *presentation graphics* can summarize data or a fitted model for the benefit of others.

Functions in the **car** and **effects** packages are relevant to all three stages. In exploratory data analysis, we want to be able to draw many graphs quickly and easily. The **car** package includes functions that are intended to simplify the process of data exploration, as described in the current chapter. For the modeling phase, quick and convenient graphs are also important. The **car** package includes a wide array of graphical functions for assessing regression models, discussed in [Chapter 8](#). Presentation graphics should effectively summarize an analysis, and their exact form depends both on the nature of the analysis performed and on the intended audience. These graphs may simply duplicate exploratory or model-building plots, or they may be more specialized and elaborate. In this edition of the *R Companion*, we greatly expand discussion of *effect plots* (introduced in [Section 4.3](#)), provided by our **effects** package, which are useful and innovative graphs for summarizing and understanding regression models. Effect plots can also play a role in assessing the adequacy of regression models (see [Section 8.4.2](#)).

- In [Section 3.1](#), we explain how to construct graphs for examining the univariate distributions of numeric variables: histograms, density plots, quantile-comparison plots, and boxplots.
- [Section 3.2](#) introduces graphs for examining relationships between two variables: bivariate scatterplots, with various enhancements, for two numeric variables, and parallel boxplots for a discrete predictor and a numeric response.
- [Section 3.3](#) describes methods for visualizing relationships in multivariate data: three-dimensional dynamic scatterplots, and scatterplot matrices for several numeric variables.
- In [Section 3.4](#), we show how to use power and modified-power transformations, including the invaluable log transformation, to make variables better behaved prior to statistical modeling by transforming them

toward normality, equal conditional variance, and linearity. We describe both analytic and exploratory methods for selecting transformations.

- [Section 3.5](#) presents a uniform framework, implemented in the **car** package, for identifying interesting points on statistical graphs, both automatically and interactively.
- [Section 3.6](#) explains how scatterplot smoothers, which trace the regression function in a scatterplot without a prior assumption about its form, are implemented in the **car** package.

The aim of the methods and tools for visualizing and transforming data described in this chapter is to help make subsequent statistical modeling simpler, more effective, and more faithful to the data.

## 3.1 Examining Distributions

### 3.1.1 Histograms

A *histogram* plots the distribution of a numeric variable by dividing the observed range of values of the variable into class intervals called *bins* and counting the number of cases falling into the bins. Either the counts, or percentages, proportions, or densities calculated from the counts, are plotted in a bar graph. To provide an example, we use the *Prestige* data set in the **carData** package:

```
library("car") # also loads carData
some(Prestige) # 10 randomly sampled rows
```

|                           | education | income | women | prestige |
|---------------------------|-----------|--------|-------|----------|
| gov.administrators        | 13.11     | 12351  | 11.16 | 68.8     |
| draughtsmen               | 12.30     | 7059   | 7.83  | 60.0     |
| secondary.school.teachers | 15.08     | 8034   | 46.80 | 66.1     |
| travel.clerks             | 11.43     | 6259   | 39.17 | 35.7     |
| bartenders                | 8.50      | 3930   | 15.51 | 20.2     |
| funeral.directors         | 10.57     | 7869   | 6.01  | 54.9     |
| babysitters               | 9.46      | 611    | 96.53 | 25.9     |
| auto.repairmen            | 8.10      | 5795   | 0.81  | 38.1     |
| train.engineers           | 8.49      | 8845   | 0.00  | 48.9     |
| typesetters               | 10.00     | 6462   | 13.58 | 42.2     |
|                           | census    | type   |       |          |
| gov.administrators        | 1113      | prof   |       |          |

|                           |      |      |
|---------------------------|------|------|
| draughtsmen               | 2163 | prof |
| secondary.school.teachers | 2733 | prof |
| travel.clerks             | 4193 | wc   |
| bartenders                | 6123 | bc   |
| funeral.directors         | 6141 | bc   |
| babysitters               | 6147 | <NA> |
| auto.repairmen            | 8581 | bc   |
| train.engineers           | 9131 | bc   |
| typesetters               | 9511 | bc   |

Like the Duncan data, introduced in [Chapter 1](#), the Prestige data set focuses on occupational prestige scores but for 102 Canadian occupations in the mid-1960s. The data set also includes potential predictors of occupational prestige scores: average years of education of incumbents in each occupation in 1970; average income of the occupation; percentage of women in the occupation; and type of occupation, a factor with levels "bc" (blue collar), "prof" (professional or managerial), and "wc" (white collar).<sup>1</sup> The first three predictors are from the 1971 Canadian Census. Row names for the Prestige data frame are the occupation titles. See help ("Prestige") for a bit more information about the data set.

[1](#) As is standard, the levels of type are alphabetized by default; see the discussion of factors in [Chapter 2](#).

A histogram of income may be drawn by the command

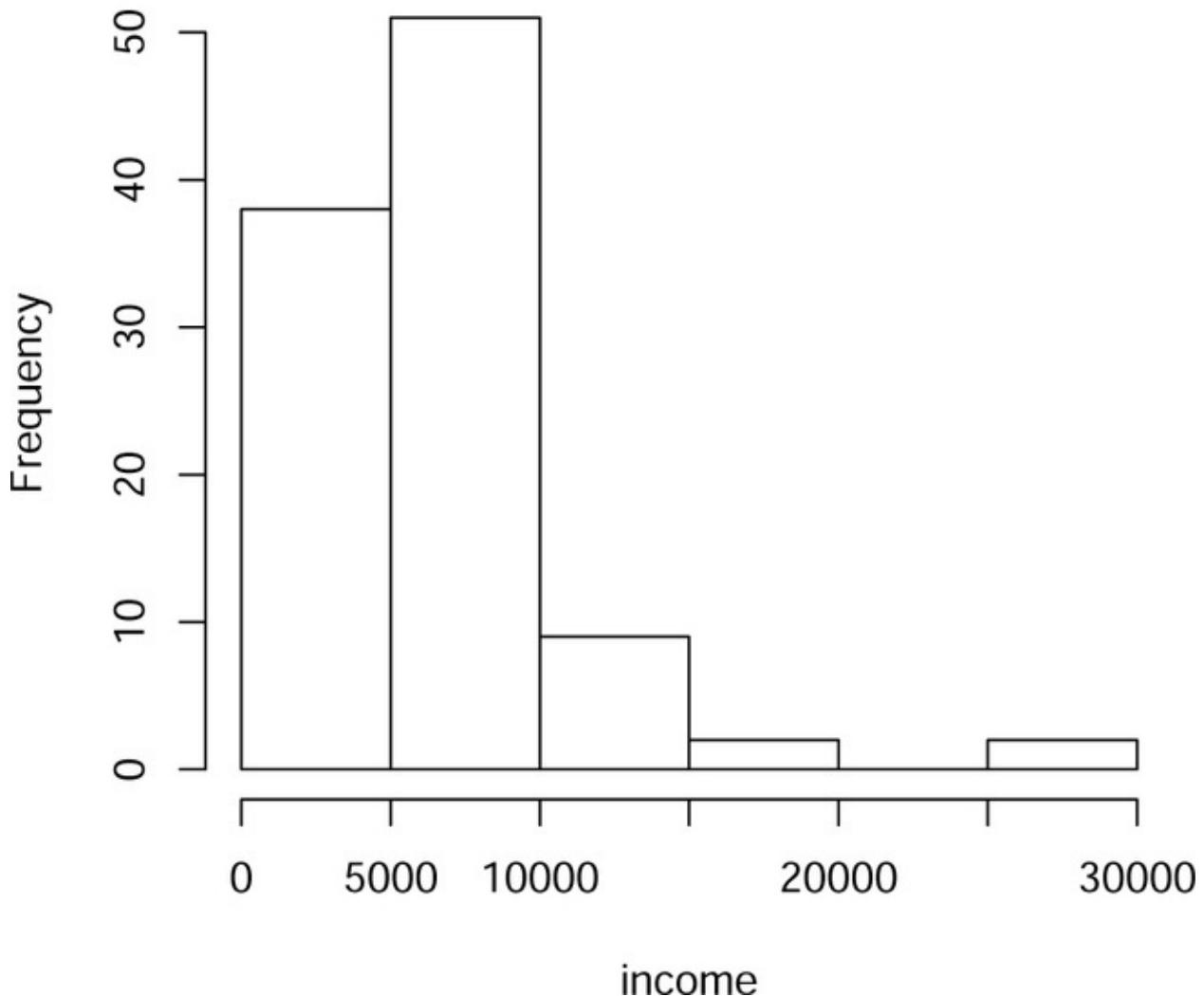
**with (Prestige, hist (income))**

We call the hist () function with no arguments beyond the name of the variable to be graphed. We use the with () function, described in [Section 1.5.1](#), to provide the data frame within which the variable income resides. The default histogram, in [Figure 3.1](#), has bins of equal width. The height of each bar is equal to the number of cases, or *frequency*, in the corresponding bin. Thus, there are 38 cases with income between \$0 and \$5,000, 51 cases with income between \$5,000 and \$10,000, and so on. Frequency histograms are only used with bins of equal width, which is the default for the hist () function. A *density histogram* is more general, where the height of each bar is determined so that the *area* of the bar is equal to the proportion of the data in the corresponding bin, and the bins need not be of equal width. The hist () function draws density histograms if the argument freq is set to FALSE or if the breaks argument is used to define bins of

unequal width.

**Figure 3.1** Default histogram of income in the Canadian occupational-prestige data.

## Histogram of income



The shape of the histogram is determined in part by the number of bins and the location of their boundaries. With too few bins, the histogram can hide interesting features of the data, while with too many bins, the histogram is very rough, displaying spurious features of the data. The default method used by `hist()` for selecting the number of bins, together with the effort to locate nice cut-points between the bins, can produce too few bins. An alternative rule, proposed by D. Freedman and Diaconis (1981), sets the target number of bins to Other

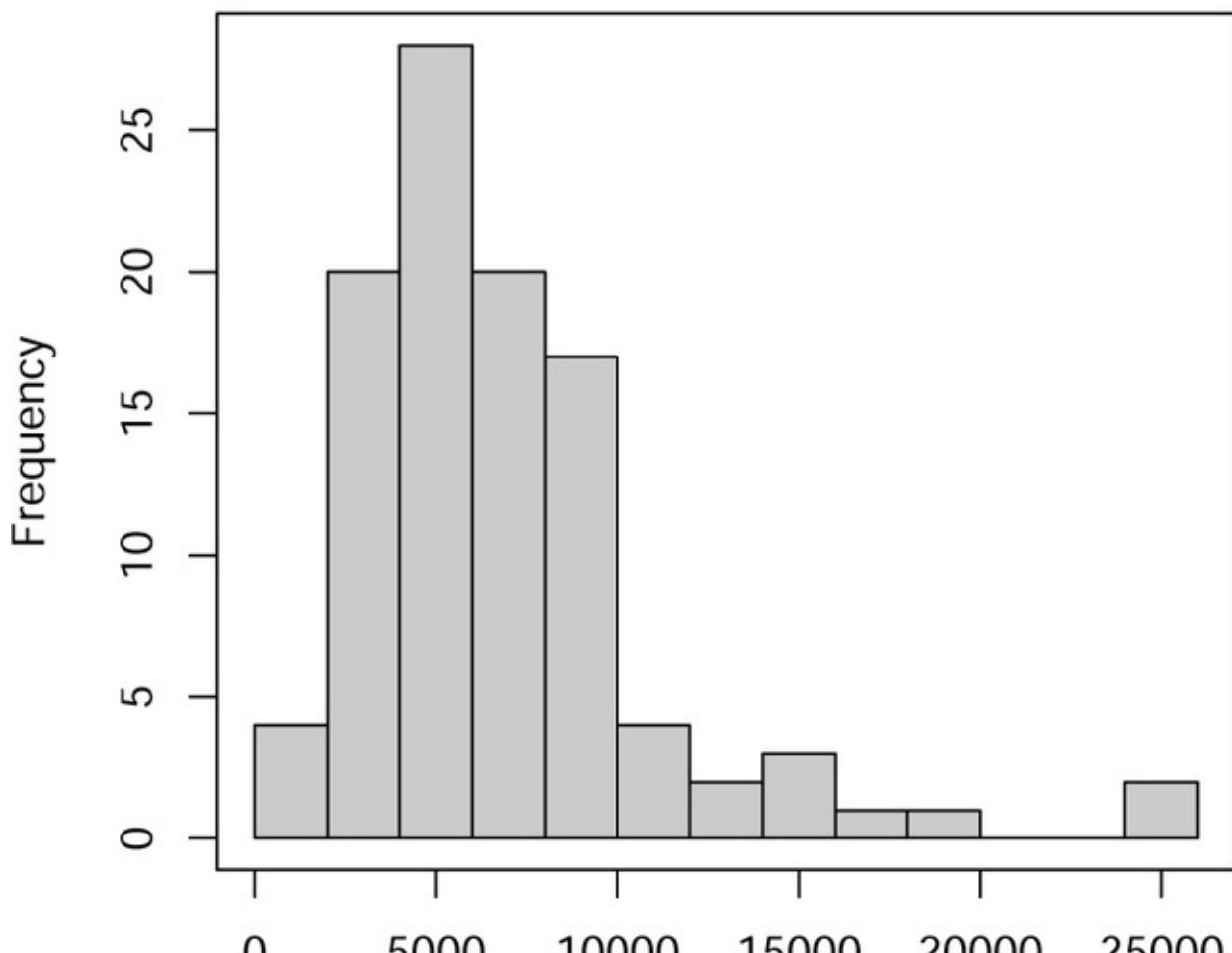
$$\left\lceil \frac{n^{1/3}(\max - \min)}{2(Q_3 - Q_1)} \right\rceil$$

where  $n$  is the number of cases in the data set,  $\max - \min$  is the range of the variable,  $Q_3 - Q_1$  is its interquartile range, and the ceiling brackets indicate rounding up to the next integer. Applying this rule to income in the Canadian occupational-prestige data produces the histogram in [Figure 3.2](#):

```
with (Prestige, hist (income, breaks="FD", col="gray", main="Average
Income in 1970 (dollars)"))
box ()
```

**Figure 3.2** Revised histogram of income.

## Average Income in 1970 (dollars)



income

Setting `col="gray"` specifies the color of the histogram bars (see [Section 9.1.4](#)), and the main argument specifies the title of the graph. The `box()` function draws a frame around the histogram and is included only for aesthetics. Both histograms suggest that the distribution of income has a single mode near \$5,000 and is skewed to the right, with several occupations that had comparatively large 1970 incomes.

As with most of the graphics functions in R, `hist()` has a dizzying array of arguments that can change the appearance of the graph:

**`args ("hist.default")`**

```
function (x, breaks = "Sturges", freq = NULL,  
probability = !freq, include.lowest = TRUE, right = TRUE,  
density = NULL, angle = 45, col = NULL, border = NULL,
```

```
main = paste ("Histogram of", xname), xlim = range (breaks),
ylim = NULL, xlab = xname, ylab, axes = TRUE, plot = TRUE,
labels = FALSE, nclass = NULL, warn.unused = TRUE, ...)
NULL
```

The `args()` command displays all the arguments of a function. We ask for the arguments of `hist.default()` rather than of `hist()` because `hist()` is a generic function (see [Section 1.7](#)), and it is the default method that actually draws the graph.<sup>2</sup> Here are some of the key arguments; a more complete description is available from `help("hist")`:

- The `breaks` argument is used to specify the cut-points for the bins. We can choose these values directly (e.g., `breaks=c(0, 5000, 10000, 15000, 20000, 25000)`), supply the target number of bins desired (e.g., `breaks=10`), or set `breaks` to the name of a rule that determines the number of equal-size bins (e.g., `breaks="FD"`). If the value of `breaks` results in unequal bin widths, then the function draws a density histogram.
- The `xlab`, `ylab`, and `main` arguments are used in most graphical functions in R to label the horizontal and vertical axes and to specify a title for the graph; for example, `main="Average Income in 1970 (dollars)"` sets the title for the plot. If we don't supply these arguments, then `hist()` constructs labels that are often reasonable.
- The `col` argument, also a common argument for graphics functions, sets the color of the bars.

[2](#) The command `args("hist")` is disappointingly uninformative because the generic `hist()` function defines only the minimal arguments that any `hist()` method must have. In this case, `args("hist.default")` is much more useful. If the result of `args("some-function")` proves uninformative, try `help("some-function")`. See [Sections 1.7](#) and [10.9](#) on object-oriented programming in R for a detailed explanation of generic functions and their methods.

You may be familiar with *stem-and-leaf displays*, which are histograms that encode the numeric data directly in their bars. We believe that stem-leaf displays, as opposed to more traditional histograms, are primarily useful for what Tukey (1977) called *scratching down numbers*—that is, paper-and-pencil methods for visualizing small data sets. That said, stem-and-leaf displays may be constructed by the standard R `stem()` function; a more sophisticated version, corresponding more closely to Tukey's original stem-and-leaf display, is provided by the `stem.leaf()` function in the **aptpack** package.

If you are looking for fancy three-dimensional effects and other *chart junk*, an apt term coined by Tufte (1983), which are often added by graphics programs to clutter up histograms and other standard graphs, you will have to look

elsewhere: The basic graphics functions in R and functions in the **car** package eschew chart junk.

### 3.1.2 Density Estimation

By in effect smoothing the histogram, *nonparametric density estimation* often produces a more satisfactory representation of the distribution of a numeric variable than a traditional histogram. Unlike a histogram, a nonparametric density estimate is continuous and so it doesn't dissect the range of a numeric variable into discrete bins. The *kernel density estimate* at the value  $x$  of a variable  $X$  is defined as

Other

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)$$

where  $\hat{p}(x)$  is the estimated density at the point  $x$ , the  $x_i$  are the  $n$  observations on the variable, and  $K$  is a *kernel function*, generally a symmetric, single-peaked density function, such as the normal density. The quantity  $h$  is called the *half-bandwidth*, and it controls the degree of smoothness of the density estimate: If  $h$  is too large, then the density estimate is smooth but biased as an estimator of the true density, while if  $h$  is too small, then bias is low but the estimate is too rough and the variance of the estimator is large. The bandwidth of a density estimate is the continuous analog of the bin width of a histogram.

Methods for selecting the bandwidth are a common topic in nonparametric statistics. The key difficulty is that for  $x$ -values with many nearby cases, one wants to use a small value of  $h$ , while for  $x$ -values with few close neighbors, a larger value of  $h$  is desired. The `adaptiveKernel()` function in the **car** package employs an algorithm that uses different bandwidths depending on the local observed density of the data, with smaller bandwidths in dense regions and larger bandwidths in sparse regions.

For example, the following code generates [Figure 3.3](#):

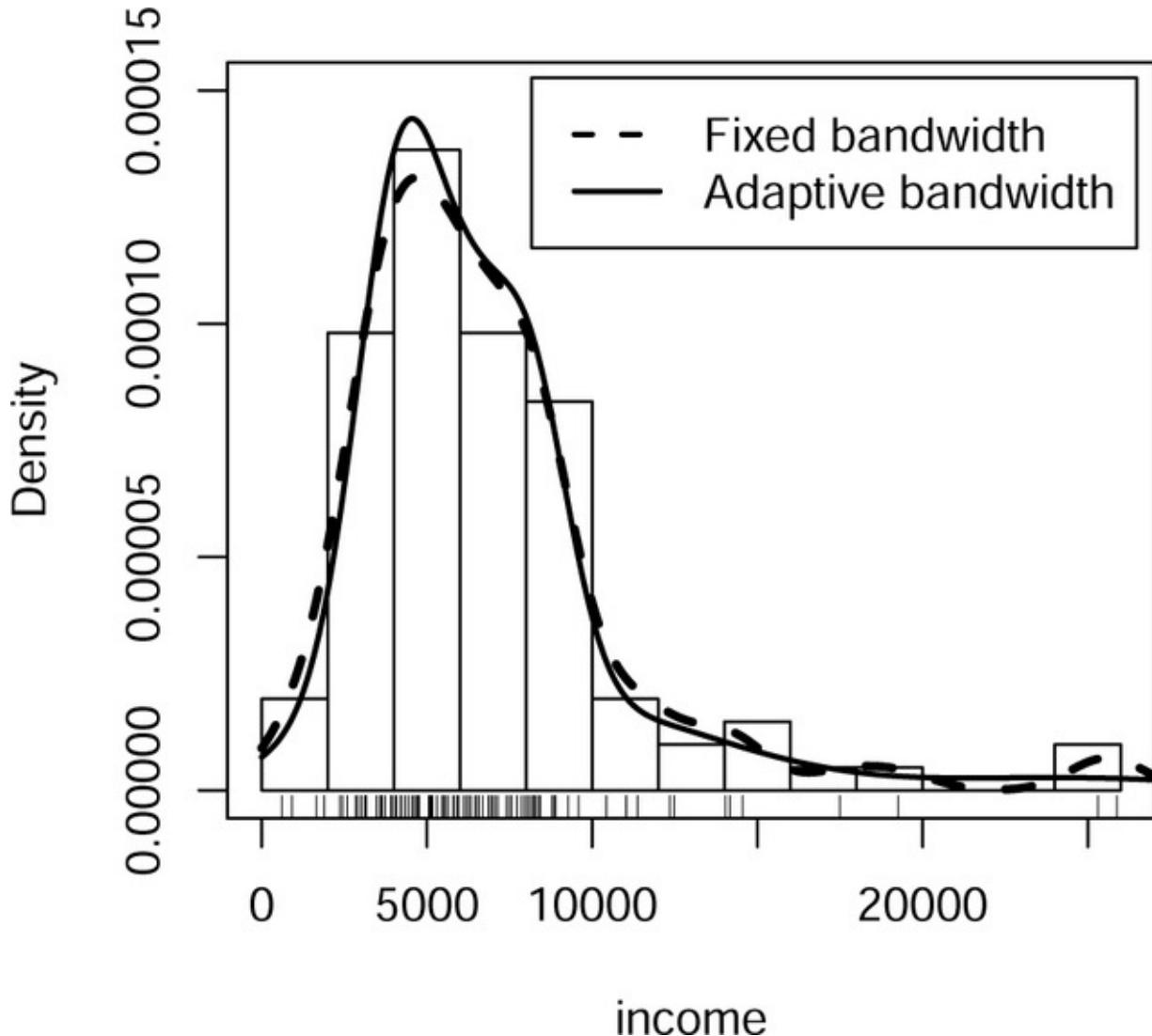
```
with(Prestige, {
  hist(income, freq=FALSE, ylim=c(0, 1.5e-4),
    breaks="FD", main="")
  lines(density(income, from=0), lwd=3, lty=2)
  lines(adaptiveKernel(income, from=0), lwd=2, lty=1)
```

```

rug(income)
legend("topright", c("Fixed bandwidth", "Adaptive bandwidth"),
      lty=2:1, lwd=2, inset=.02)
box()
})

```

**Figure 3.3** Nonparametric fixed-bandwidth and adaptive-bandwidth kernel density estimates for the distribution of income in the Prestige data set; a density histogram of income is also shown. The rug-plot at the bottom of the graph shows the location of the income values.



Three estimates of the density of income are shown, using a histogram, a fixedkernel density estimate, and an adaptive-kernel density estimate. As in [Figure 3.2](#), we use `breaks="FD"` to determine the number of bars in the histogram, and we employ default bandwidths for the two kernel density

estimates. For a positively skewed distribution, like the distribution of income in the example, the default bandwidth can be too small, suppressing detail in the distribution. The `adjust` argument to `density()` or `adaptiveKernel()` can be used to modify the default bandwidth. The default value of the argument is `adjust=1`; setting `adjust=0.5`, for example, makes the bandwidth half as wide as the default.<sup>3</sup>

[3](#) We invite the reader to experiment with the value of the `adjust` argument in this example.

The command to draw the graph in [Figure 3.3](#) is relatively complicated and thus requires some explanation:

- The `with()` function is used as usual to pass the data frame `Prestige` to the second argument. Here the second argument is a *compound expression* consisting of all the commands between the initial `{` and the final `}`.
- The call to `hist()` draws a histogram in density scale, so the areas of all the bars in the histogram sum to 1.
- The argument `main=""` suppresses the title for the histogram.
- The `ylim` argument sets the range for the y-axis to be large enough to accommodate the adaptive-kernel density estimate. The value `1.5e-4` is in scientific notation,  $1.5 \times 10^{-4} = 0.00015$ .
- The fixed-bandwidth and adaptive-bandwidth kernel estimates are computed, respectively, by `density()` and `adaptiveKernel()`. In each case, the result returned by the function is then supplied as an argument to the `lines()` function to add the density estimate to the graph.
- The argument `from=0` to both `density()` and `adaptiveKernel()` ensures that the density estimates don't go below `income = 0`.
- The call to `rug()` draws a *rug-plot* (one-dimensional scatterplot) of the data at the bottom of the graph.
- The remaining two commands add a legend and a frame around the graph.<sup>4</sup>

[4](#) For more information about customizing R graphs, see [Chapter 9](#).

Both nonparametric density estimates and the histogram suggest a mode at around \$5,000, and all three show that the distribution of income is right-skewed. The fixed-bandwidth kernel estimate has more wiggle at the right where data are sparse, and the histogram is rough in this region, while the adaptive-kernel estimator is able to smooth out the density estimate in the low-density

region.

The **car** package has an additional function, `densityPlot ()`, that both computes and draws density estimates with either a fixed or adaptive kernel. Taking all defaults, for example, we get an adaptive-kernel estimate:

```
densityEstimate (~ income, data=Prestige) # graph not shown
```

Many graphics functions in the **car** package allow you to use a formula to specify the variable or variables to be plotted—in this case, the *one-sided formula* `~ income`. If you use a formula for the variables in a graph, then the data and subset arguments may be included to supply the name of a data frame and the rows in the data frame to be used in the graph. To get the fixed-bandwidth kernel estimator, set the argument `method="kernel"` in the call to `densityPlot ()`. For the remaining arguments, see help ("densityPlot").

### 3.1.3 Quantile-Comparison Plots

We may want to compare the distribution of a variable with a theoretical reference distribution, such as the normal distribution. A *quantile-comparison plot*, or *quantile-quantile plot (QQ-plot)*, provides an effective graphical means of making the comparison, plotting the ordered data on the vertical axis against the corresponding quantiles of the reference distribution on the horizontal axis. If the data conform to the reference distribution, then the points in the quantile-comparison plot should fall close to a straight line, within sampling error.

R provides the `qqnorm ()` function for making quantile-comparison plots against the normal distribution, but we prefer the `qqPlot ()` function in the **car** package. By default, `qqPlot ()` compares the data with the normal distribution. [Figure 3.4](#) shows a normal QQ-plot for income in the Prestige data:

```
qqPlot(~ income, data=Prestige, id=list(n=3))
```

general.managers

2

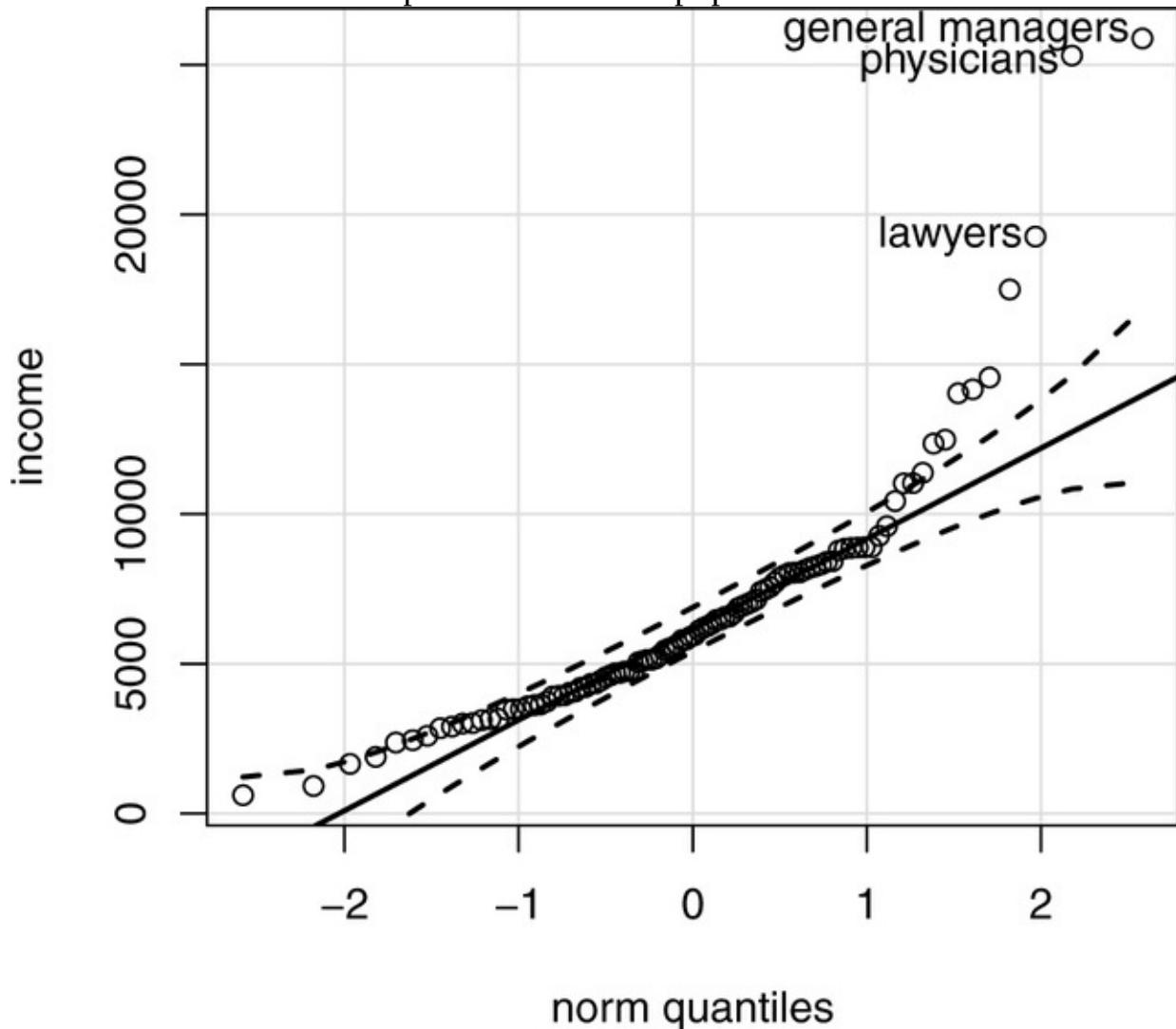
physicians

24

lawyers

17

**Figure 3.4** Normal quantile-comparison plot for income. The broken lines give a pointwise 95% confidence envelope around the fitted solid line. Three points were labeled automatically. Because many points, especially at the right of the graph, are outside the confidence bounds, we have evidence that the distribution of income is not like a sample from a normal population.



If the values of income behave like a sample from a normal distribution, then the points should lie close to a straight line. The straight line shown on the plot is determined empirically by drawing a line through the pairs of first and third quartiles of income and the comparison normal distribution; this behavior can be changed with the `line` argument. The dashed lines on the graph show an approximate 95% pointwise confidence envelope around the reference line, assuming that the data are a normal sample. We expect most of the points to fall within the envelope most of the time.<sup>5</sup> In the example, the largest values of

income are clearly outside the confidence envelope, and the overall pattern of the points is curved, not straight, casting doubt on the assumption of normality. That the points are above the reference line at both ends of the distribution is indicative of a positive skew. The QQ-plot agrees with the histogram and kernel density estimates in [Figure 3.3](#): Normal densities are symmetric, and the QQ-plot and density estimates all suggest positive skewness.

[5](#) Base R includes the function `shapiro.test()`, which tests the null hypothesis that the sampled values were drawn from a normal distribution versus the alternative that the population is not normally distributed; the test is based essentially on the square of the correlation between the  $x$  and  $y$  coordinates of the points in the normal QQ-plot.

As is true for most of the graphics functions in the `car` package, the `id` argument to `qqPlot()` supports both interactive and automatic marking of extreme points. In the example, we set `id=list(n=3)` to label the three most extreme points in the QQ-plot—the three points most remote from the center of the data. More generally, the definition of “extreme” depends on context.[6](#)

[6](#) See [Section 3.5](#) for a more complete discussion of point labeling in the `car` package.

The `qqPlot()` function can be used to plot the data against *any* reference distribution for which there are quantile and density functions in R, which includes just about any distribution that you may wish to use. Simply specify the root word for the distribution. For example, the root for the normal distribution is "norm", with density function `dnorm()` and quantile function `qnorm()`. The root for the chi-square distribution is "chisq", with density and quantile functions `dchisq()` and `qchisq()`. Root words for some other commonly used distributions are "binom" and "pois" for the binomial and Poisson distributions, respectively (which, as discrete distributions, have probability-mass functions rather than density functions), "f" for the  $F$ -distribution, "t" for the  $t$ -distribution, and "unif" for the uniform distribution.

In addition to density and quantile functions, R also provides cumulative distribution functions, with the prefix `p`, and pseudo-random-number generators, with prefix `r`. For example, `pnorm()` gives cumulative probabilities for the normal distributions, while `rnorm()` generates normal random variables. [Table 3.1](#) summarizes the principal arguments to these probability functions.

**Table 3.1**

| <i>Distribution</i> | <i>Density or Mass Function</i> | <i>Quantile Function</i> |
|---------------------|---------------------------------|--------------------------|
| normal              | dnorm(x, mean=0, sd=1)          | qnorm(p, mean=0, sd=1)   |
| chi-square          | dchisq(x, df)                   | qchisq(p, df)            |
| F                   | df(x, df1, df2)                 | qf(p, df1, df2)          |
| t                   | dt(x, df)                       | qt(p, df)                |
| binomial            | dbinom(x, size, prob)           | qbinom(p, size, prob)    |
| Poisson             | dpois(x, lambda)                | qpois(p, lambda)         |
| uniform             | dunif(x, min=0, max=1)          | qunif(p, min=0, max=1)   |

| <i>Distribution</i> | <i>Distribution Function</i> | <i>Random Number Function</i> |
|---------------------|------------------------------|-------------------------------|
| normal              | pnorm(q, mean=0, sd=1)       | rnorm(n, mean=0, sd=1)        |
| chi-square          | pchisq(q, df)                | rchisq(n, df)                 |
| F                   | pf(q, df1, df2)              | rf(n, df1, df2)               |
| t                   | pt(q, df)                    | rt(n, df)                     |
| binomial            | pbinom(q, size, prob)        | rbinom(n, size, prob)         |
| Poisson             | ppois(q, lambda)             | rpois(n, lambda)              |
| uniform             | runif(q, min=0, max=1)       | runif(n, min=0, max=1)        |

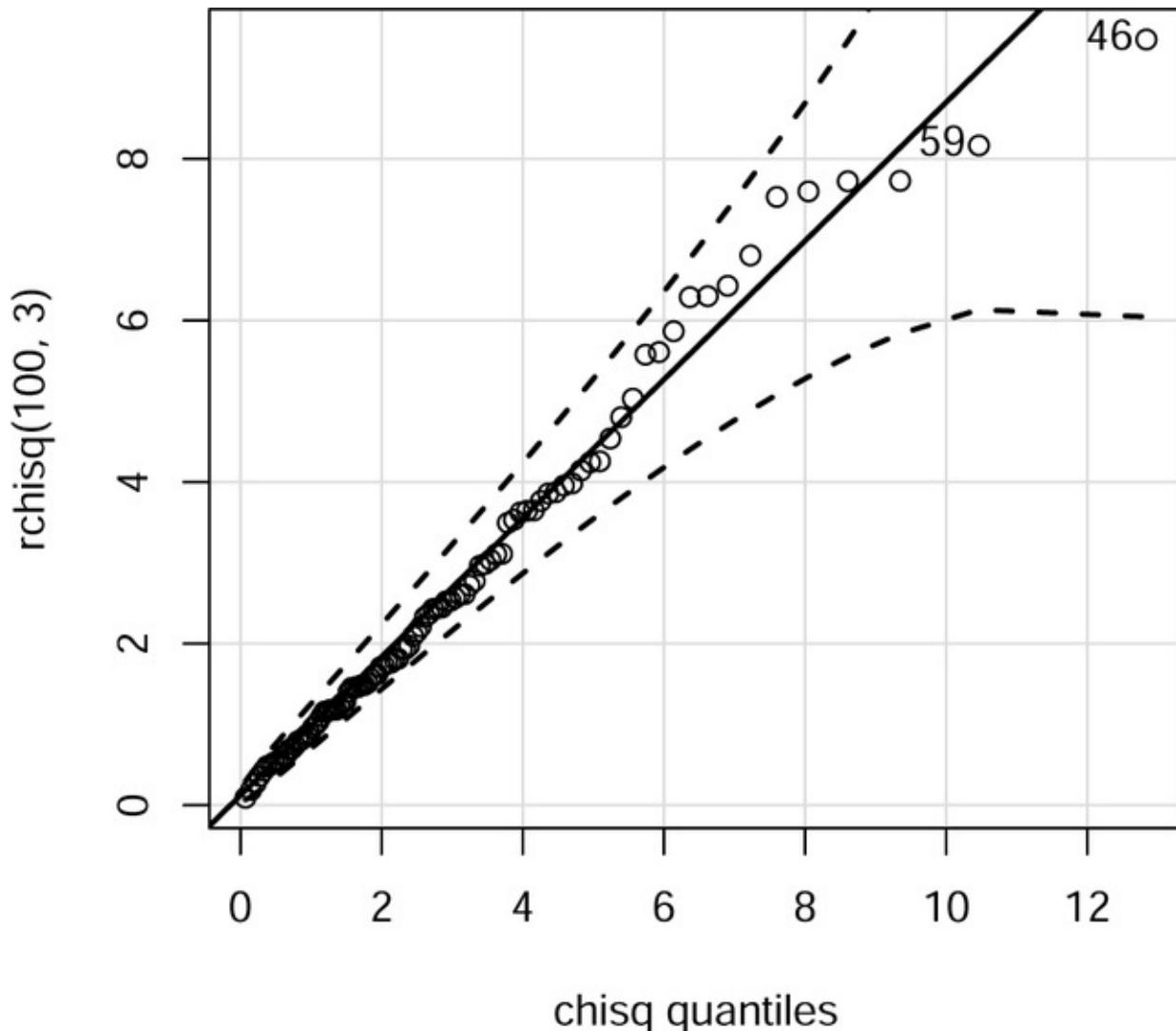
To illustrate, we use the `rchisq()` function to generate a random sample from the chi-square distribution with three *df* and then plot the sample against the distribution from which it was drawn, producing [Figure 3.5](#):<sup>7</sup>

```
set.seed(124) # for reproducibility
```

```
qqPlot(rchisq(100, 3), distribution = "chisq", df=3)
```

```
[1] 46 59
```

**Figure 3.5** Quantile-comparison plot of a sample of size  $n = 100$  from the  $\chi^2(3)$  distribution against the distribution from which the sample was drawn.



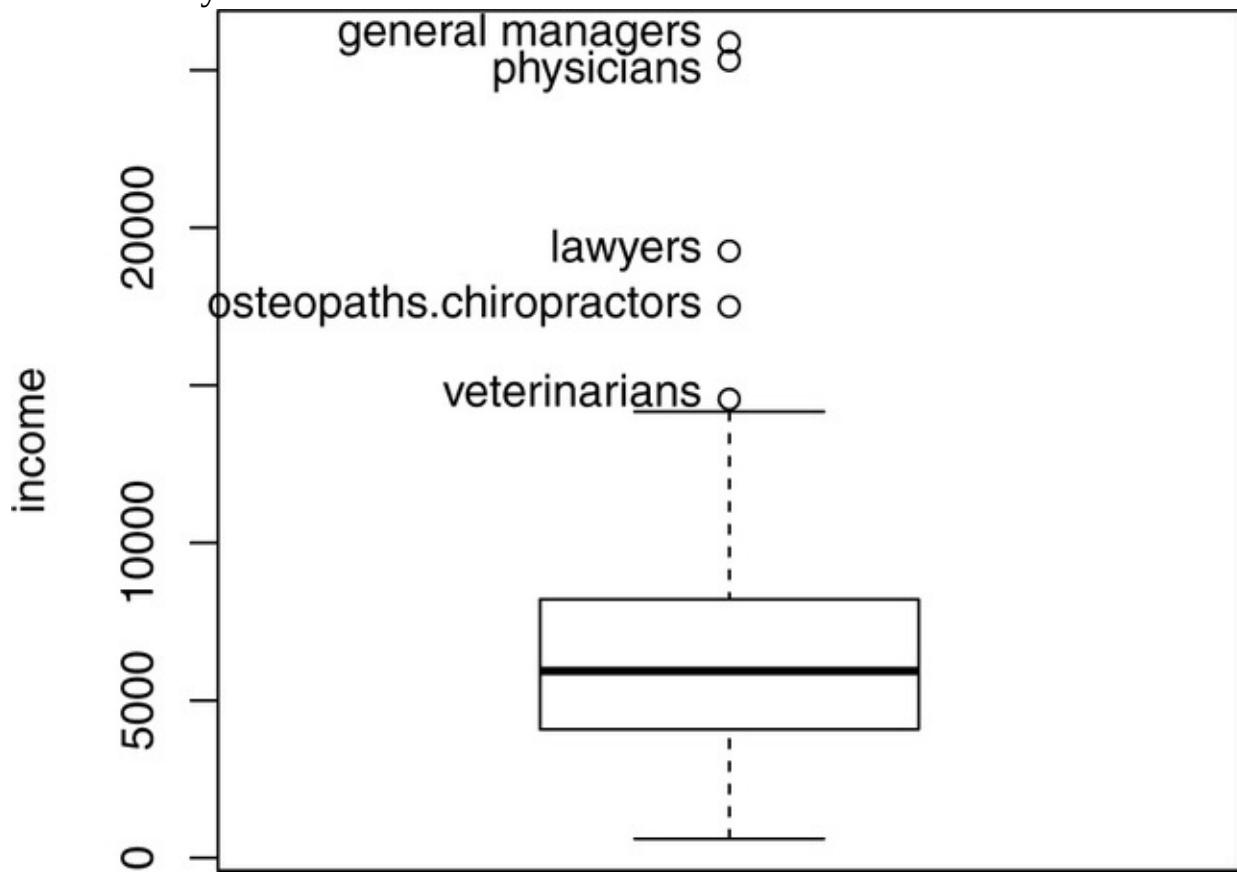
[7](#) We set the seed for R’s random-number generator to an arbitrary known value to make the example reproducible—if you use the same seed, you’ll get exactly the same sampled values: See [Section 10.7](#) on conducting random simulations in R.

The argument `df=3` to `qqPlot()` is passed by it to the `dchisq()` and `qchisq()` functions. The points should, and do, closely match the straight line on the graph, with the fit a bit worse for the larger values in the sample. The confidence envelope suggests that these deviations for large values are to be expected, as they reflect the greater variability of sampled values in the long right tail of the  $\chi^2(3)$  density function.

### 3.1.4 Boxplots

The final univariate display that we describe is the *boxplot*. Although boxplots are most commonly used, as in [Section 3.2.2](#), to compare distributions among groups, they can also be drawn to summarize the distribution of a numeric variable in a single sample, providing a quick check of symmetry and the presence of outliers. [Figure 3.6](#) shows a boxplot for income, produced by the `Boxplot()` function in the `car` package. This function is a convenient front end to the basic R `boxplot()` function, adding automatic identification of outlying values (by default, up to 10), points for which are shown individually in the boxplot.<sup>8</sup>

**Figure 3.6** Boxplot of income. Several outlying cases were labeled automatically.



<sup>8</sup> Points identified as outliers are those beyond the *inner fences*, which are 1.5 times the interquartile range below the first quartile and above the third quartile. This rule for identifying outliers was suggested by Tukey (1977, chap. 2).

```
Boxplot(~ income, data=Prestige)
```

```
[1] "general.managers"           "lawyers"  
[3] "physicians"                 "veterinarians"  
[5] "osteopaths.chiropractors"
```

The variable to be plotted, `income`, is given in a one-sided formula, with data drawn from the `Prestige` data set. The names shown in the output are the cases that are labeled on the graph and are drawn from the row names of the `Prestige` data set.

## 3.2 Examining Relationships

### 3.2.1 Scatterplots

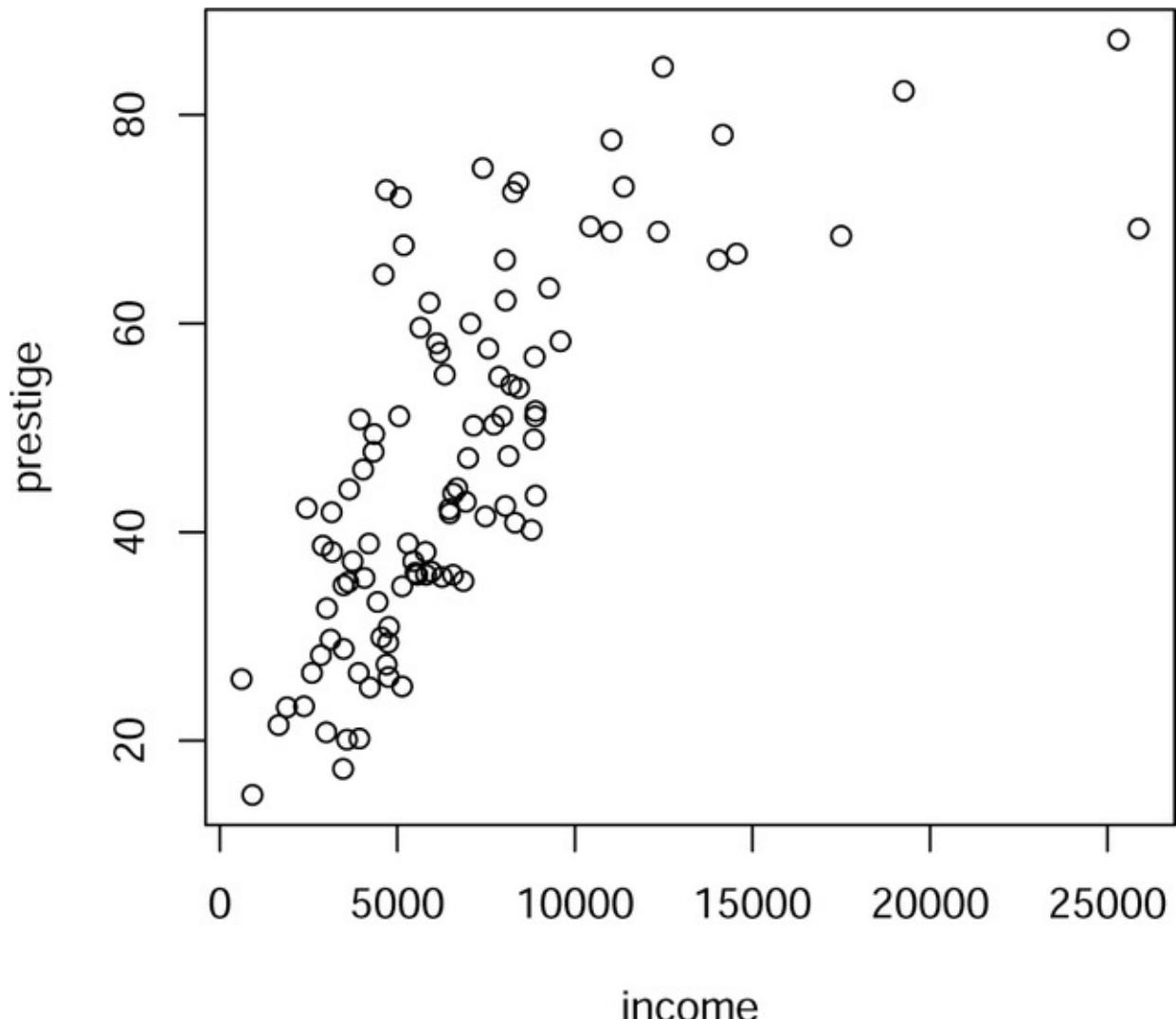
A *scatterplot* is the familiar graph of points with one quantitative variable on the horizontal or  $x$ -axis and a second quantitative variable on the vertical or  $y$ -axis. Understanding and using scatterplots is at the heart of regression analysis.<sup>9</sup> There is typically an asymmetric role of the two axes, with the  $y$ -axis reserved for a response variable and the  $x$ -axis for a predictor.

[9](#) We emphasize *using* scatterplots because we find that in practice, the common advice that one should plot one's data prior to statistical modeling is a rule that it often honored in the breach.

The generic `plot()` function is the basic tool in R for drawing graphs in two dimensions. What this function produces depends on the classes of its first one or two arguments.<sup>10</sup> If the first two arguments to `plot()` are numeric vectors, then we get a scatterplot, as in [Figure 3.7](#):

***with (Prestige, plot (income, prestige))***

**Figure 3.7** Simple scatterplot of prestige versus income for the Canadian occupational-prestige data.



[10](#) The behavior of generic functions such as `plot()` is discussed in [Sections 1.7](#) and [10.9](#); more information about the `plot()` function is provided in [Section 3.2.3](#) and in [Chapter 9](#) on R graphics.

The first argument to `plot()` is the *x*-axis variable and the second argument is the *y*-axis variable. The scatterplot in [Figure 3.7](#) is a *summary graph* for the regression problem in which *prestige* is the response and *income* is the predictor.<sup>11</sup> As our eye moves from left to right across the graph, we see how the distribution of *prestige* changes as *income* increases. In technical terms, we are visualizing the *conditional distributions* of *prestige* given the value of *income*. The overall story here is that as *income* increases, so does *prestige*, at least up to about \$10,000, after which the value of *prestige* stays more or less fixed on average at about 80.

[11](#) See R. D. Cook and Weisberg (1999, chap. 18) for a general treatment of summary graphs in regression and the related concept of *structural dimension*.

We write  $E(\text{prestige}|\text{income})$  to represent the mean value of prestige as the value of income varies and call this the *conditional mean function* or the *regression function*. The qualitative statements in the previous paragraph therefore concern the regression function. The *variance function*,  $\text{Var}(\text{prestige}|\text{income})$ , traces the conditional variability in prestige as income changes—that is, the variability of  $y$  in vertical strips in the plot.

Scatterplots are useful for studying the mean and variance functions in the regression of the  $y$ -variable on the  $x$ -variable. In addition, scatterplots can help identify *outliers*, points that have values of the response far different from the expected value, and *leverage points*, cases with extremely large or small values on the  $x$ -axis. How these ideas relate to multiple regression is a topic discussed in [Chapter 8](#).

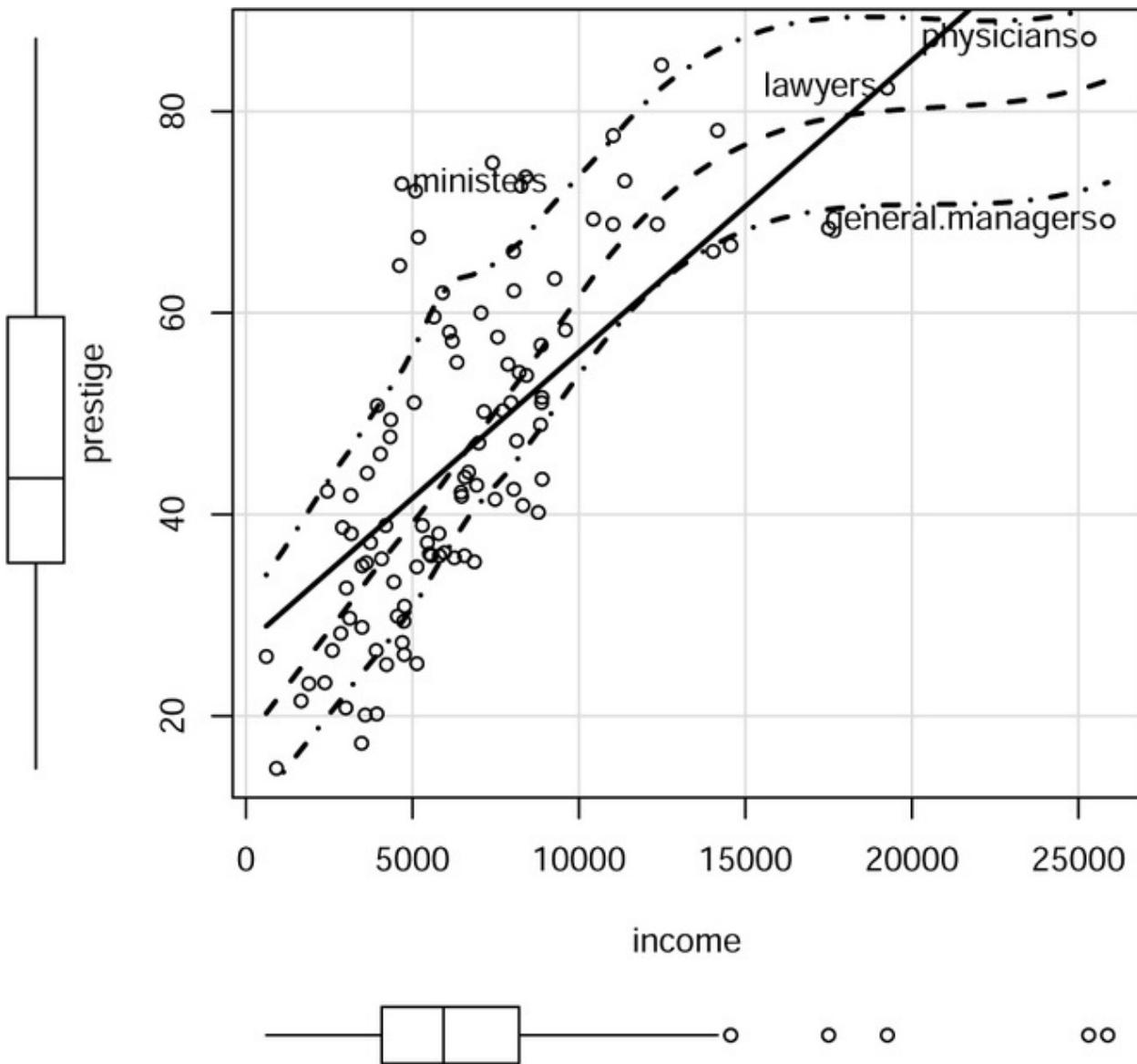
## Enhanced Scatterplots

The `scatterplot()` function in the `car` package can draw scatterplots with a wide variety of enhancements and options.[12](#) A basic application of the function is shown in [Figure 3.8](#):

```
scatterplot(prestige ~ income, data=Prestige, id=list(n=4))
```

|                  |         |           |
|------------------|---------|-----------|
| general.managers | lawyers | ministers |
| 2                | 17      | 20        |
| physicians       |         |           |
| 24               |         |           |

**Figure 3.8** Enhanced scatterplot of prestige versus income produced by the `scatterplot()` function. Four points were identified using the `id` argument.



[12](#) In Version 3 of the **car** package, associated with the third edition of the *R Companion*, we have consolidated the arguments to `scatterplot()` and other plotting functions to simplify their use.

- Points correspond to the pairs of (income, prestige) values in the `Prestige` data frame, which is supplied in the `data` argument.
- The thick solid straight line in the scatterplot is the simple linear regression of prestige on income fit by ordinary least squares.
- The dashed line is fit using a nonparametric *scatterplot smoother*, and it provides an estimate of the mean function that does not depend on a parametric regression model, linear or otherwise.[13](#)
- The two dash-dotted lines provide a nonparametric estimate of the square

root of the variance function (i.e., the conditional standard deviation of the response), based on separately smoothing the positive and negative residuals from the fitted nonparametric mean function.

- Also shown on each axis are *marginal boxplots* of the plotted variables, summarizing the univariate distributions of  $x$  and  $y$ .
- The only optional argument we used in drawing [Figure 3.8](#) is `id=list(n=4)`, to identify by row name the four most extreme cases, where by default “extreme” means farthest from the point of means using *Mahalanobis distances*.<sup>14</sup>

[13](#) The use of scatterplot smoothers in the **car** package is discussed in [Section 3.6](#).

[14](#) Unlike simple Euclidean distances, which are inappropriate when the variables are scaled in different units, Mahalanobis distances take into account the variation and correlation of  $x$  and  $y$ .

All these features, and a few others that are turned off by default, can be modified by the user, including the color, size, and symbol for points; the color, thickness, and type for lines; and inclusion or exclusion of the least-squares fit, the mean smooth, variance smooth, and marginal boxplots. See help ("scatterplot") for the available options.

The least-squares line shows the estimated mean function assuming that the mean function is a straight line, while the regression smoother estimates the mean function without the assumption of linearity. In [Figure 3.8](#), the least-squares line cannot match the obvious curve in the mean function that is apparent in the smooth fit, so modeling the relationship of prestige to income by simple linear regression is likely to be inappropriate.<sup>15</sup>

[15](#) We use a smoother here, and in the **car** package more generally, as a plot enhancement, designed to help us extract information from a graph. *Nonparametric regression*, in which smoothers are substituted for more traditional regression models, is described in an online appendix to the *R Companion*.

## Conditioning on a Categorical Variable

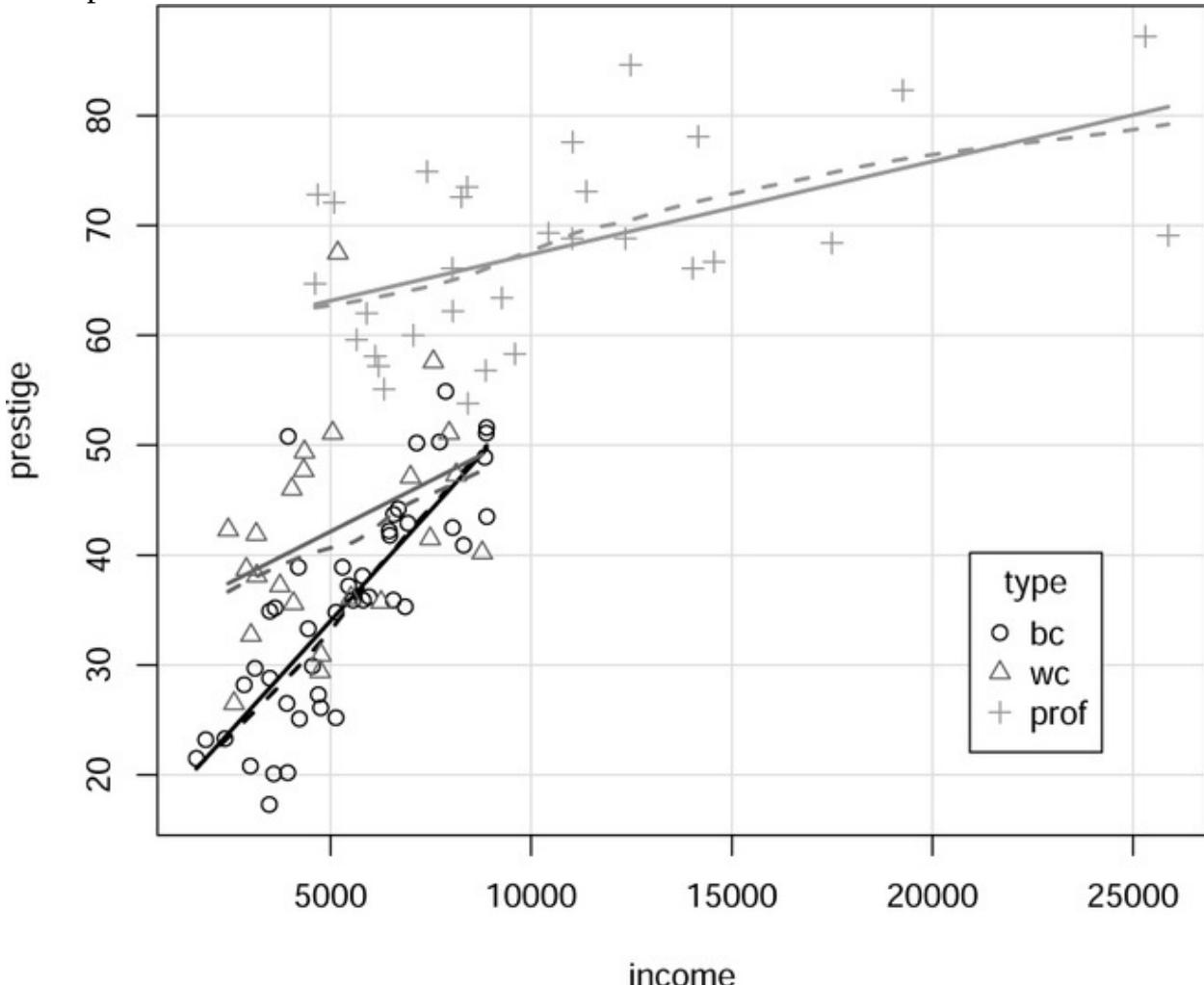
Introducing the categorical variable type, type of occupation, we can examine

the relationship between prestige and income separately for each level of the factor type. Because type has only three levels, we can draw a single graph by using distinct colors and symbols for points in the different levels, fitting separate smoothers to each group, as illustrated in [Figure 3.9](#).<sup>16</sup> Before drawing the graph, we reorder the levels of type, which are initially alphabetical:

```
Prestige$type <- factor (Prestige$type, levels=c ("bc", "wc", "prof"))
```

```
scatterplot (prestige ~ income | type, data=Prestige, legend=list
(coords="bottomright", inset=0.1), smooth=list (span=0.9))
```

**Figure 3.9** Scatterplot of prestige versus income, coded by type of occupation. The span of the loess smoother is set to 0.9.



[16](#) The graphs in this text are in monochrome; by default, the scatterplot () function uses different colors for points and lines in different groups.

The variables for the *coded scatterplot* are given in a formula as  $y \sim x | g$ , which is interpreted as plotting  $y$  on the vertical axis,  $x$  on the horizontal axis, and marking points according to the value of  $g$  (or “ $y$  versus  $x$  given  $g$ ”). The legend for the graph, automatically generated by the scatterplot () function, is placed by default above the plot; we specify the legend argument to move the legend to the lower-right corner of the graph. We select a larger *span*, *span*=0.9, for the scatterplot smoothers than the default (*span*=2/3) because of the small numbers of cases in the occupational groups:[17](#)

```
xtabs (~type, data=Prestige)
```

| type |    |      |
|------|----|------|
| bc   | wc | prof |
| 44   | 23 | 31   |

[17](#) The default smoother employed by scatterplot () is the *loess* smoother. The *span* in *loess* is the fraction of the data used to determine the fitted value of  $y$  at each  $x$ . A larger *span* therefore produces a smoother result, and too small a *span* produces a fitted curve with too much wiggle. The trick is to select the smallest *span* that produces a sufficiently smooth regression mean function. The default *span* of 2/3 works well most of the time but not always. Scatterplot smoothers are discussed in [Section 3.6](#).

[Figure 3.9](#) allows examination of three regression functions simultaneously:  $E(\text{prestige}|\text{income}, \text{type} = "bc")$ ,  $E(\text{prestige}|\text{income}, \text{type} = "wc")$ , and  $E(\text{prestige}|\text{income}, \text{type} = "prof")$ . The nonlinear relationship in [Figure 3.8](#) has disappeared, and we now have three reasonably linear regressions with different slopes. The slope of the relationship between prestige and income looks steepest for blue-collar occupations and least steep for professional and managerial occupations.

## Jittering Scatterplots

Variables with integer or other discrete values often result in uninformative

scatterplots. For an example, consider the `Vocab` data set in the `carData` package:

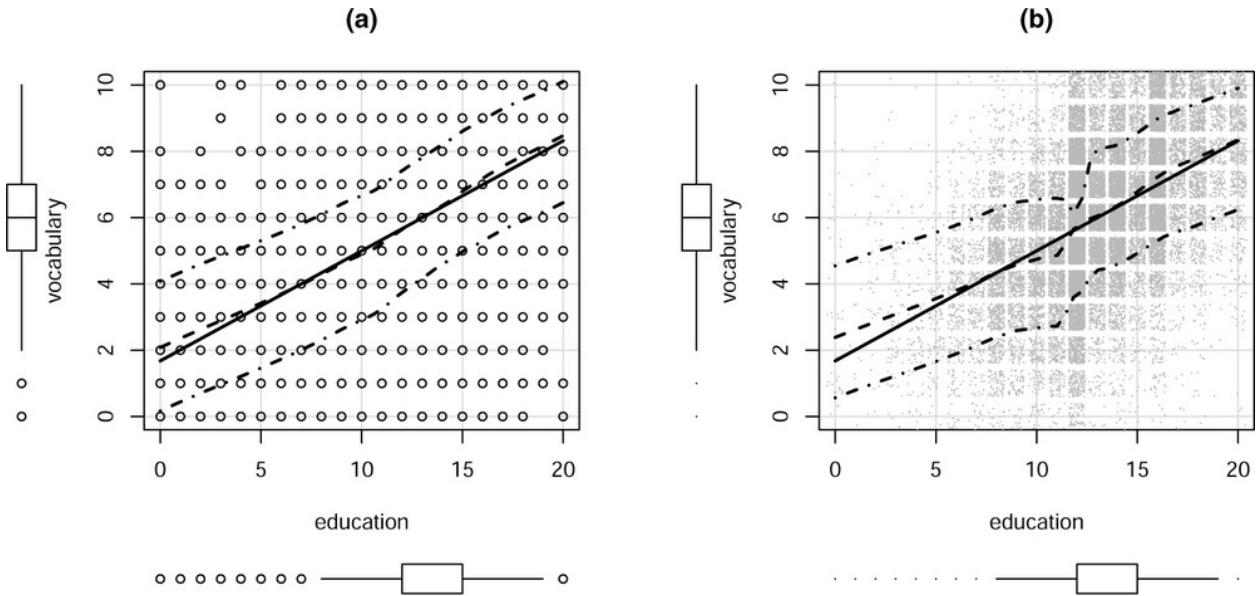
```
brief(Vocab) # a few cases

30351 x 4 data.frame (30346 rows omitted)
  year    sex education vocabulary
  [n]    [f]      [n]        [n]
19740001 1974 Male       14         9
19740002 1974 Male       16         9
19740003 1974 Female     10         9
...
20162865 2016 Female     14         9
20162866 2016 Female     14         5
```

The data are from the U.S. General Social Surveys, 1974–2016, conducted by the National Opinion Research Center. The response of interest is the respondent's vocabulary score on a 10-word vocabulary test, with 11 distinct possible values between zero and 10, and the predictor education is the respondent's years of education, with 21 distinct possible values between zero and 20 years. A scatterplot with default options is shown in [Figure 3.10 \(a\)](#):

```
scatterplot (vocabulary ~ education, data=Vocab, main = "(a)")
```

**Figure 3.10** Scatterplots of vocabulary by education: (a) using defaults, (b) jittering the points.



The only nondefault argument, `main="a"`, sets the title for the graph.

There are only  $11 \times 21 = 231$  possible plotting positions, even though there are more than 30,000 cases. Nearly all of the possible plotting positions are consequently heavily overplotted, producing a meaningless rectangular grid of dots. The plot enhancements in [Figure 3.10 \(a\)](#) provide some useful information. From the marginal boxplots, we see that about 50% of the values of education are concentrated between 12 and 15 years. This boxplot does not show a line for the median, and so the median must be on a boundary of the box, either at 15 years or, more likely, at 12 years, for high school graduates with no college. Thus, education is negatively skewed, with a long left (lower) tail. The distribution of the response variable, vocabulary, is more symmetric, with the median number of correct answers equal to six. The least-squares line and the default loess smooth seem to agree very closely, so the mean of vocabulary appears to increase linearly with years of education. The conditional variability of vocabulary also appears to be constant across the plot, because the variability smooths are parallel to the loess smooth.

An improved version of this graph is shown in [Figure 3.10 \(b\)](#):

```
scatterplot(vocabulary ~ education, data=Vocab,
            jitter=list(x=2, y=2), cex=0.01, col="darkgray",
            smooth=list(span=1/3, col.smooth="black",
                        col.var="black"),
            regLine=list(col="black"), main="(b)")
```

We use several optional arguments to scatterplot () to enhance the original graph:

- The argument jitter=list (x=2, y=2) adds small random numbers to the  $x$  and  $y$  coordinates of each plotted point. The values  $x=2$  and  $y=2$  specify the degree of jittering relative to the default amount, in this case twice as much jittering. The amounts of jitter used were determined by trial and error to find the choices that provide the most information.
- The argument cex=0.01 reduces the size of the circles for the plotted points to 1% of their normal size, and col="darkgray" sets their color to gray, more appropriate choices when plotting more than 30,000 points. As a consequence of jittering and using smaller and lighter plotting symbols, we see clearly that the density of points for education = 12, high school graduates, is higher than for other years, and that the data for education < 8 are very sparse.
- We use the smooth argument to set the span for the default loess smoother to 1/3, half the default value of 2/3. Setting a smaller span uses less data to estimate the fitted curve at each value of education, allowing us to resolve greater detail in the regression function (see footnote 17 on page 139). Here we observe a dip in the regression function when education  $\approx$  11, so individuals who just missed graduating from high school perform somewhat worse than expected by a straight-line fit on the vocabulary test. Similarly, there is a small positive bulge in the regression function corresponding to college graduation, approximately 16 years of education.
- The specifications col.smooth="black" and col.var="black" set the color of the loess and variability lines, making them more visible in the graph; the default is the color of the points, now gray.
- Finally, as before, the main argument sets the title of the graph.

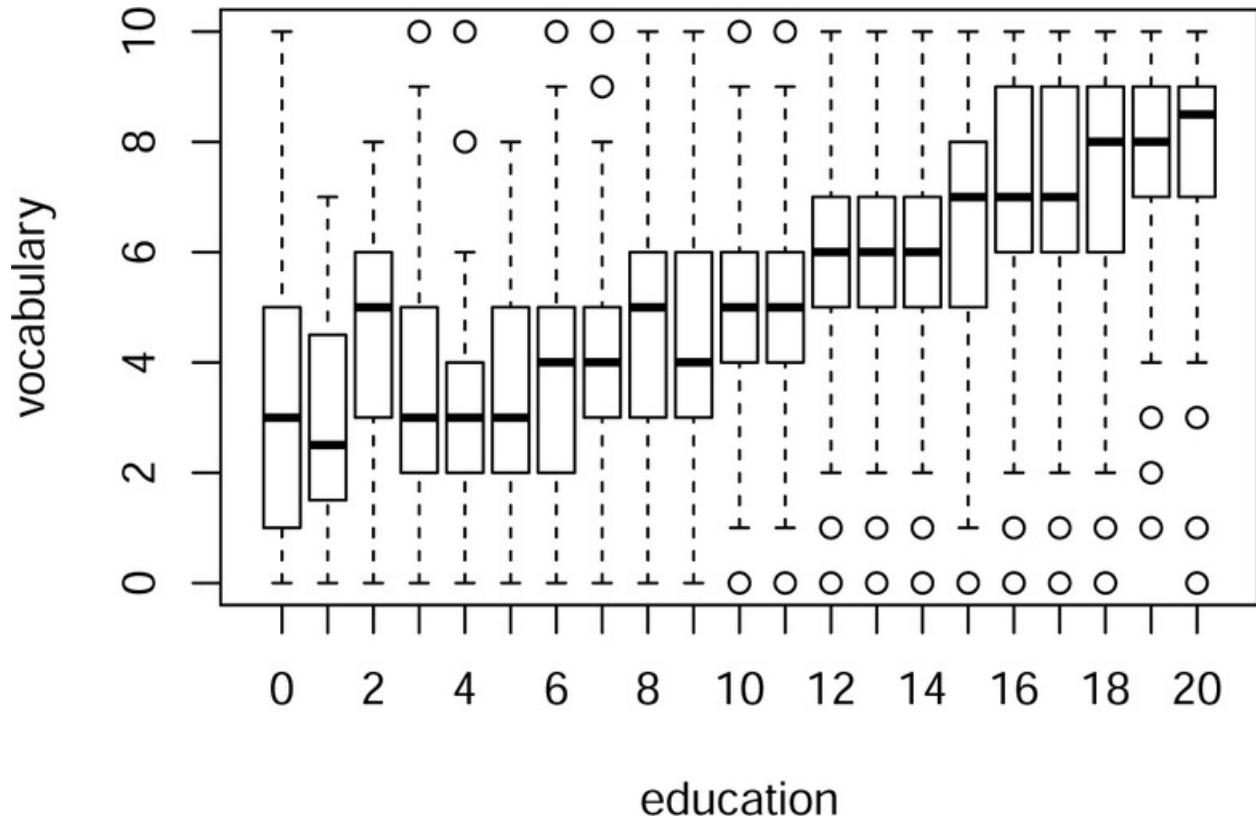
### 3.2.2 Parallel Boxplots

We can further explore the relationship between vocabulary and years of education in the Vocab data frame using parallel boxplots, as shown in [Figure](#)

[3.11](#), produced by the command

**Boxplot (vocabulary ~ education, data=Vocab, id=FALSE)**

**Figure 3.11** Boxplots of vocabulary separately for each value of years of education.



This command draws a separate boxplot for vocabulary for all cases with the same value of education, so we condition on the value of education. Unlike most **car** graphics functions, **Boxplot** sets automatic point identification as the default, but that is very distracting in this example, and so we specify `id=FALSE` to suppress point marking. The formula has the response variable on the left of the `~` and a discrete conditioning variable—typically, but not necessarily, a factor—on the right. In this example, the conditioning predictor `education` is a discrete numeric variable. For the subsamples with larger sample sizes,  $8 \leq \text{education} \leq 18$ ,<sup>18</sup> rather than a steady increase in vocabulary with education, there appear to be jumps every 2 or 3 years, at years 10, 12, 15, and 18.

<sup>18</sup> Enter the command `xtabs (~ education, data=Vocab)` to examine the

distribution of education.

A more typical example uses data from Ornstein (1976) on interlocking directorates among 248 major Canadian corporations:

### ***brief(Ornstein)***

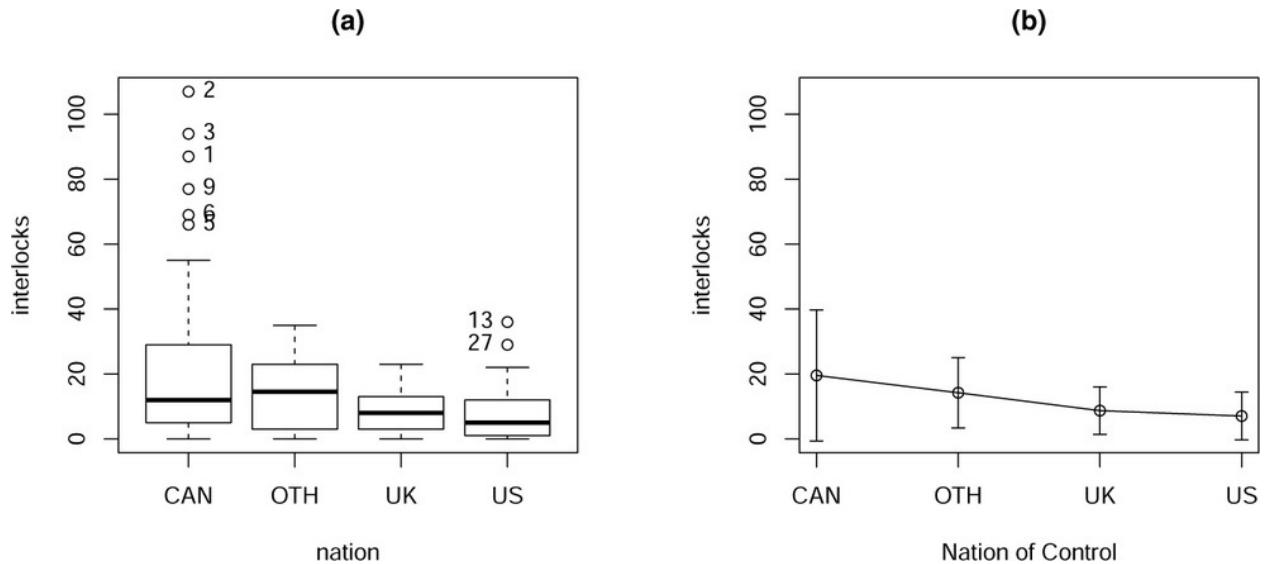
| 248 x 4 data.frame (243 rows omitted) |        |        |        |            |
|---------------------------------------|--------|--------|--------|------------|
|                                       | assets | sector | nation | interlocks |
|                                       | [i]    | [f]    | [f]    | [i]        |
| 1                                     | 147670 | BNK    | CAN    | 87         |
| 2                                     | 133000 | BNK    | CAN    | 107        |
| 3                                     | 113230 | BNK    | CAN    | 94         |
| ...                                   | ...    | ...    | ...    | ...        |
| 247                                   | 119    | AGR    | CAN    | 6          |
| 248                                   | 62     | MIN    | US     | 0          |

The variables in the data set include the assets of each corporation, in millions of dollars; the corporation's sector of operation, a factor with 10 levels; the factor nation, indicating the country in which the firm is controlled, with levels "CAN" (Canada), "OTH" (other), "UK", and "US"; and interlocks, the number of interlocking directorate and executive positions maintained between each company and others in the data set. [Figure 3.12 \(a\)](#) shows a boxplot of the number of interlocks for each level of nation:

***Boxplot (interlocks ~ nation, data=Ornstein, main = "(a)" )***

```
[1] "1" "2" "3" "5" "6" "9" "13" "27"
```

**Figure 3.12 (a)** Parallel boxplots of interlocks by nation of control, for Ornstein's interlocking-directorate data. (b) A mean/standard deviation plot of the same data.



Because the names of the companies are not given in the original data source, the points are labeled by case numbers. The firms are in descending order by assets, and thus the identified points are among the largest companies.

A more common plot in the scientific literature is a graph of group means with error bars showing  $\pm 1$  SD around the means. This plot can be drawn conveniently using the `plotCI()` function in the **plotrix** package (Lemon, 2017), as shown, for example, in [Figure 3.12 \(b\)](#):<sup>19</sup>

```

library ("plotrix")

means <- Tapply (interlocks ~ nation, mean, data=Ornstein) sds <- Tapply
(interlocks ~ nation, sd, data=Ornstein) plotCI (1:4, means, sds,
xaxt="n", xlab="Nation of Control",
ylab="interlocks", main="(b)",
ylim=range (Ornstein$interlocks))

lines (1:4, means)

axis (1, at=1:4, labels = names (means))

```

<sup>19</sup> These plots are sometimes drawn with intervals of  $\pm 1$  standard error rather than  $\pm 1$  SD, and sometimes error bars are added to bar charts rather than to a

point plot of means. We discourage the use of bar charts for means because interpretation of the length of the bars, and therefore the visual metaphor of the graph, depends on whether or not a meaningful origin (zero) exists for the measured variable and whether or not the origin is included in the graph. The error bars can also lead to misinterpretation, because neither standard-error bars nor standard-deviation bars are the appropriate measure of variation for comparing means between groups: They make no allowance or correction for multiple testing, among other potential problems.

The Tapply () function in the **car** package, described in [Section 10.5](#), adds a formula interface to the standard tapply () function. The first call to Tapply () computes the mean of interlocks for each level of nation, and the second computes within-nation standard deviations. The basic graph is drawn by plotCI (). The first argument to this function specifies the coordinates on the horizontal axis, the second the coordinates on the vertical axis, and the third the vector of SDs. The standard graphical argument xaxt="n" suppresses the x-axis tick marks and labels, and the ylim argument is used here to match the vertical axis of panel (b) to that of panel (a). The lines () function joins the means, and the axis () function labels the horizontal axis with the names of the groups. The first argument to axis () specifies the side of the graph where the axis is to be drawn: side=1 (as in the example) is below the graph, 2 at the left, 3 above, and 4 at the right. See help ("axis") for details and [Chapter 9](#) for an extended discussion of customizing R graphs.

The parallel boxplots in [Figure 3.12 \(a\)](#) and the mean/SD plot in [Figure 3.12 \(b\)](#) purport to provide similar information, but the impressions one gets from the two graphs are very different. The boxplots allow us to identify outliers and recognize skewness, with a few larger values of interlocks at some levels of nation. The mean/SD graph is misleading, and rather than showing the outliers, the graph inflates both the means and the SDs, particularly for Canada, and disguises the skewness that is obvious in the boxplots. Both graphs, however, suggest that the variation in interlocks among firms within nations is greater than the differences among nations.

### 3.2.3 More on the plot () Function

Suppose that x and y are numeric variables, that g is a factor, and that m is an object produced, for example, by one of the many statistical-modeling functions

in R, such as lm () for fitting linear models, discussed in [Chapter 4](#), or glm () for fitting generalized linear models, discussed in [Chapter 6](#). Then:

- Assuming that x and y reside in the global environment, plot (y ~ x) or plot (x, y) produces a basic scatterplot with y on the vertical axis and x on the horizontal axis. If x and y are two variables in the data frame D, then we can enter the command plot (y ~ x, data=D), plot (D\$x, D\$y), or with (D, plot (x, y)). If we employ a formula to specify the plot, then we can use the data argument, but we can't use the data argument if the plotted variables are given as two arguments.
- plot (x) produces a scatterplot with x on the *vertical* axis and case numbers on the horizontal axis, called an *index plot*.
- plot (y ~ g) is the same as boxplot (y ~ g), using the standard R boxplot () function, not the Boxplot () function in the **car** package.
- What plot (m) does depends on the *class* of the object m.<sup>20</sup> For example, if m is a linear-model object created by a call to lm (), then plot (m) draws several graphs that are commonly associated with linear regression models fit by least squares. In contrast, plot (density (x)) draws a plot of the density estimate for the numeric variable x.

[20](#) The class-based object-oriented programming system in R and its implementation through generic functions such as plot () are explained in [Sections 1.7](#) and [10.9](#).

The plot () function can take many additional optional arguments that control the appearance of the graph, the labeling of its axes, the fonts used, and so on. We can set some of these options globally with the par () function or just for the current graph via arguments to the plot () function. We defer discussion of these details to [Chapter 9](#), and you can also look at help ("par") for a description of the various graphics parameters.

Plots can be built up sequentially by first creating a basic graph and then adding to it. For example, a figure similar to [Figure 3.10 \(b\)](#) (on page 140) can be drawn using the following commands:

```
plot (jitter (vocabulary, factor=2) ~ jitter (education, factor=2), cex=0.01,  
col="darkgray", data=Vocab)
```

```
abline (lm (vocabulary ~ education, data=Vocab), lwd=2, lty=2)
```

```
with (Vocab, loessLine (education, vocabulary, var=TRUE,  
smoother.args=list (span=1/3)))
```

This sequence of commands uses plot () to draw a jittered scatterplot, and then calls the abline () and loessLine () functions (the latter from the **car** package) to add linear-least-squares and nonparametric-regression lines to it. The resulting plot isn't shown; we invite the reader to enter these commands line by line to create the graph.

## 3.3 Examining Multivariate Data

Scatterplots provide summaries of the conditional distribution of a numeric response variable given a numeric predictor. When there are  $m > 1$  predictors, we would like to be able to draw a plot in  $m + 1$  dimensions of the response versus all of the predictors simultaneously. Because the media on which we draw graphs—paper and computer displays—are two-dimensional, and because in any event we can only perceive objects in three spatial dimensions, examining multivariate data is intrinsically more difficult than examining univariate or bivariate data.

### 3.3.1 Three-Dimensional Plots

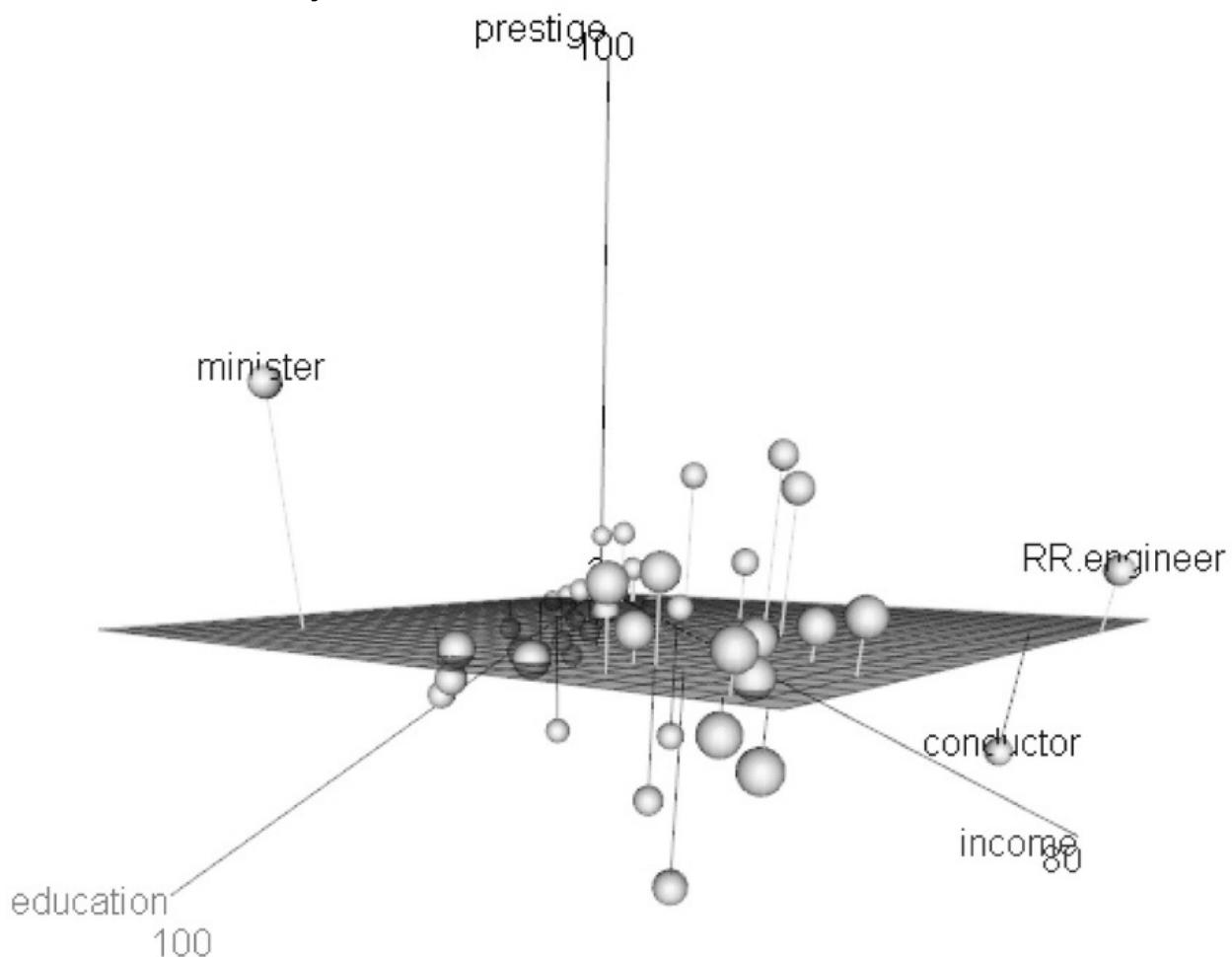
Perspective, motion, and illumination can convey a sense of depth, enabling us to examine data in three dimensions on a two-dimensional computer display. The most effective software of this kind allows the user to rotate the display, to mark points, and to plot surfaces such as regression mean functions.

The **rgl** package (Adler & Murdoch, 2017) links R to the OpenGL three-dimensional graphics library often used in animated films. The scatter3d () function in the **car** package uses **rgl** to provide a three-dimensional generalization of the scatterplot () function. For example, the command

```
scatter3d (prestige ~ income + education, data=Duncan, id=list (n=3))
```

produces [Figure 3.13](#), which is a three-dimensional scatterplot for Duncan's occupational-prestige data. The graph shows the least-squares regression plane for the regression of the variable on the vertical or  $y$ -axis, prestige, on the two variables on the horizontal (or  $x$ - and  $z$ -) axes, income and education; three cases (minister, conductor, and railroad engineer) are identified as the most unusual based on their Mahalanobis distances from the centroid (i.e., the point of means) of the three variables. The three-dimensional scatterplot can be rotated by left-clicking and dragging with the mouse. Color is used by default, with perspective, sophisticated lighting, translucency, and fog-based depth cueing. The overall effect is much more striking on the computer screen than in a static monochrome printed graph.

**Figure 3.13** Three-dimensional scatterplot for Duncan's occupational-prestige data, showing the least-squares regression plane. Three unusual points were labeled automatically.



The `scatter3d()` function can also plot other regression surfaces (e.g., non-

parametric regressions), can identify points interactively and according to other criteria, can plot concentration ellipsoids, and can rotate the plot automatically. Because three-dimensional dynamic graphs depend on color, perspective, motion, and so on for their effectiveness, we refer the reader to the help file for `scatter3d()` and to the examples therein.

There are also facilities in R for drawing static three-dimensional graphs, including the standard R `persp()` function, and the `cloud()` and `wireframe()` functions in the **lattice** package. The **rggobi** package links R to the GGobi system for visualizing data in three and more dimensions (Swayne, Cook, & Buja, 1998; D. Cook & Swayne, 2009).

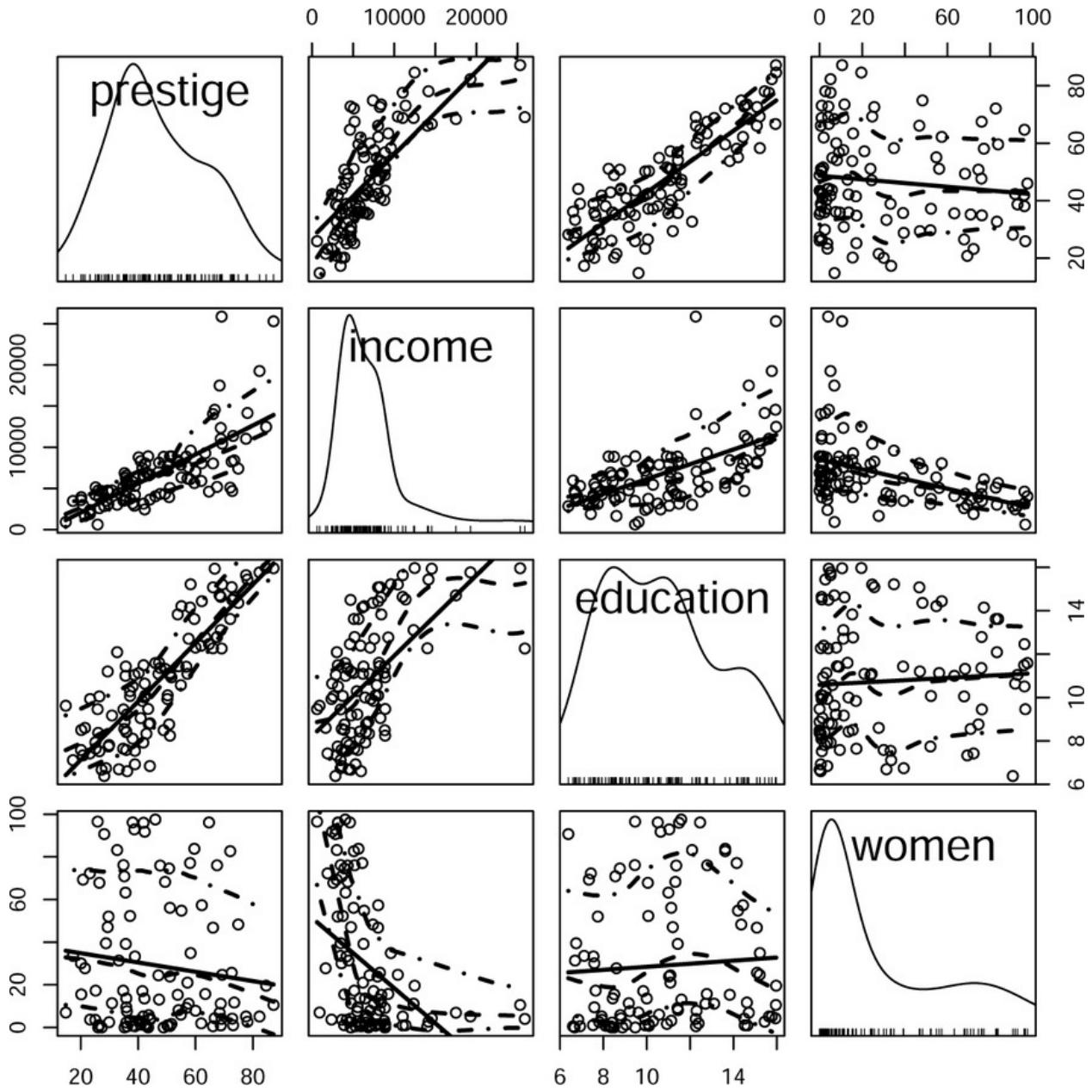
### 3.3.2 Scatterplot Matrices

*Scatterplot matrices* are graphical analogs of correlation matrices, displaying bivariate scatterplots of all pairs of numeric variables in a data set as a two-dimensional graphical array. Because the panels of a scatterplot matrix are just two-dimensional scatterplots, each panel is the appropriate summary graph for the regression of the  $y$ -axis variable on the  $x$ -axis variable. The `pairs()` function in R and the `splom()` function in the **lattice** package draw scatterplot matrices.

The `scatterplotMatrix()` function in the **car** package uses `pairs()` to draw a basic scatterplot matrix but adds a number of plot enhancements. The `scatterplotMatrix()` function bears the same relationship to `pairs()` that `scatterplot()` bears to `plot()`. An example, using the Canadian occupational-prestige data, appears in [Figure 3.14](#):

```
scatterplotMatrix (~ prestige + income + education + women,  
data=Prestige)
```

**Figure 3.14** Scatterplot matrix for the Canadian occupational-prestige data, with density estimates on the diagonal.



The first argument to `scatterplotMatrix()` is a *one-sided formula*, specifying the variables to be plotted separated by + signs. We interpret this formula as “plot prestige, income, education, and women.” In [Figure 3.8](#) (on page 137), we previously constructed the scatterplot of prestige versus income, and this scatterplot is also in the first row and second column of [Figure 3.14](#), summarizing the regression of prestige on income. In contrast, the plot of income versus prestige, in the second row, first column of the scatterplot matrix, is for the regression of income on prestige. The `scatterplotMatrix()` function also provides univariate information about the plotted variables in the diagonal panels. The default, to use adaptive-kernel density estimates, is shown in [Figure](#)

### 3.14.

As in scatterplot (), mean and variability smoothers and a least-squares regression line are added by default to each panel of a scatterplot matrix and are controlled respectively by optional smooth and regLine arguments. The id argument for marking points is also the same for both functions, except that interactive point-marking isn't supported for scatterplot matrices. The diagonal argument to scatterplotMatrix () controls the contents of the diagonal panels. The scatterplot () and scatterplotMatrix () functions have many other arguments in common: See help ("scatterplotMatrix") for details.

## 3.4 Transforming Data

The way we measure and record data may not necessarily reflect the way the data should be used in a regression analysis. For example, automobile fuel usage in the United States is measured by fuel *efficiency*, in miles per gallon. In most of the rest of the world, fuel usage is measured by fuel *consumption*, in liters per 100 km, which is the *inverse* of miles per gallon times a factor to account for the change in the units of measurement from gallons to liters and from miles to kilometers. More fuel-efficient cars have larger values of miles per gallon and smaller values of liters per 100 km. If we are interested in a regression in which fuel usage is either the response or a predictor, the choice between these two scales is not obvious a priori.

Similar ideas apply when a variable is a percentage or fraction, such as the percentage of an audience that responds to an advertising campaign. An increase from 2% to 4% is a substantial change, whereas an increase from 40% to 42%, still an increase of 2%, is much less substantial. Percentages may need to be transformed for this reason.

Learning to use transformations effectively is part of the subtle craft of data analysis. Good computational tools, such as those described in this section, can help choose effective transformations.

### 3.4.1 Logarithms: The Champion of Transformations

Suppose that we are interested in the variable annual salary, measured in dollars. An increase in salary from, say, \$20,000 to \$22,000 is an increase of 10%, a

substantial increase to the person receiving it, whereas an increase from \$120,000 to \$122,000, still \$2,000 but only a 1.67% increase, is much less consequential. Logarithmic scale corresponds to viewing variation through relative or percentage changes, rather than through absolute changes. Varshney and Sun (2013) argue that in many instances humans perceive relative changes rather than absolute changes, and so logarithmic scales are often theoretically appropriate as well as useful in practice.<sup>21</sup>

<sup>21</sup> If you are unfamiliar with logarithms, often abbreviated as “logs,” see the brief review in footnote 4 (page 7) and the complementary readings cited at the end of the chapter.

You may encounter or choose to use *natural logs* to the base  $e \approx 2.718$ , *common logs* to the base 10, and in this book we often use logs to the base 2. For many important aspects of statistical analyses, such as fitted values, tests, and model selection, the base of logarithms is inconsequential, in that identical conclusions will be reached regardless of the base selected. The interpretation of regression coefficients, however, may be simpler or more complicated depending on the base. For example, increasing the log base-2 of  $x$  by 1 implies doubling  $x$ , but increasing the natural log of  $x$  by 1 implies multiplying  $x$  by  $e$ . Different bases for logarithms differ only by multiplication by a constant. For example,  $\log(x) = \log_2(x)/\log_2(e) \approx 0.692 \log_2(x)$ , and thus converting a base-2 logarithm to a natural logarithm requires only multiplication by (approximately) 0.692. In the *R Companion*, we use natural logarithms unless we specifically state otherwise, so notationally  $\log(x)$  always means the natural logarithm of  $x$ , while  $\log_2(x)$  is the base-2 logarithm of  $x$ , and  $\log_{10}(x)$  is the common logarithm.

The R functions `log()`, `log10()`, and `log2()` compute the natural, base-10, and base-2 logarithms, respectively:

```

c("natural log"=log(7), "base-2 log"=log2(7),
  "common log"=log10(7))

natural log  base-2 log  common log
1.9459      2.8074      0.8451

log2(7)/log2(exp(1)) # again the natural logarithm

[1] 1.9459

```

The first of these commands employs the `c()` function to collect the results returned by several commands in a single line of output. The arguments to `c()` are arbitrary and are just used to label the values of the resulting vector; we put these arguments in quotes because they contain spaces and a dash and are therefore not standard R names.

The *exponential function* `exp()` computes powers of  $e$ , and thus  $\exp(1) = e$ . Exponentiating is the inverse of taking logarithms, and so the equalities  $x = \exp(\log(x)) = \log(\exp(x))$  hold for any positive number  $x$  and base  $b$ :

`exp(log(50))`

[1] 50

`log(exp(50))`

[1] 50

`10^(log10(50))`

[1] 50

`log10(10^50)`

[1] 50

Although you will find little use for bases other than  $e$ , 10, and 2 in applied statistics, the `log()` and `logb()` functions in R can be used to compute logs to any base:

```
c (log10(7), log (7, base=10), logb (7, 10))
```

```
[1] 0.8451 0.8451 0.8451
```

The various log functions in R can be applied to numbers, numeric vectors, numeric matrices, and numeric data frames. They return NA for missing values, NaN (Not a Number) for negative values, and -Inf ( $-\infty$ ) for zeros.

One use for logarithms is to enhance visual perception when viewing a one-dimensional summary of a variable in a histogram or density estimate. As an example, the Ornstein data set, introduced in [Section 3.2.2](#), includes measurements of the assets of  $n = 248$  large Canadian companies. Adaptive-kernel density plots of assets and their logs are shown in [Figure 3.15](#):

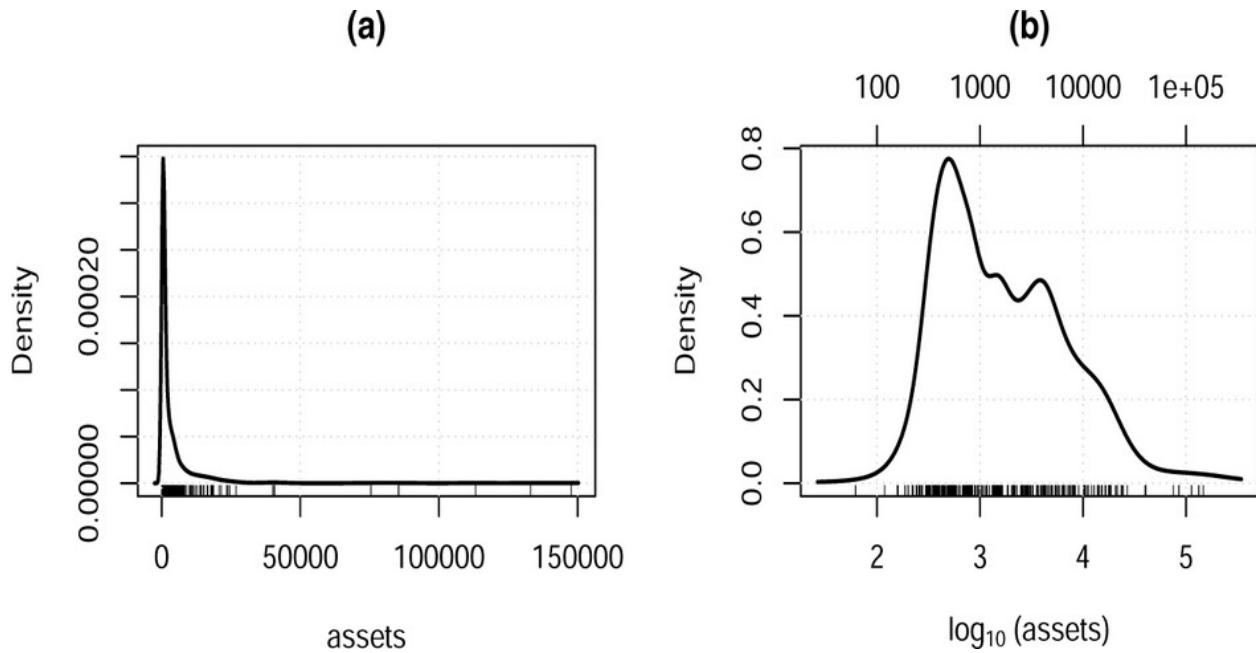
```
par (mfrow=c (1, 2), mar=c (5, 4, 6, 2) + 0.1)
```

```
densityPlot (~ assets, data=Ornstein, xlab="assets", main=(a))
```

```
densityPlot (~ log10(assets), data=Ornstein, adjust=0.65, xlab=expression (log[10]~"(assets)"), main=(b))
```

```
basicPowerAxis (0, base=10, side="above", at=10^(2:5), axis.title="")
```

**Figure 3.15** Distribution of assets (millions of dollars) in the Ornstein data set (a) before and (b) after log transformation.



The command `par(mfrow=c(1, 2))` sets the global graphics parameter "mfrow", which divides the graphics window into an array of subplots with one row and two columns. We also set the "mar" (margins) graphics parameter to increase space at the top of the plots and call the `basicPowerAxis()` function in the `car` package to draw an additional horizontal axis at the top of panel (b) showing the original units of assets.<sup>22</sup> The `xlab` argument in the second command is set equal to an *expression*, allowing us to typeset 10 as the subscript of log.<sup>23</sup>

<sup>22</sup> See [Chapter 9](#) on R graphs.

<sup>23</sup> For more on how to typeset mathematical notation in R graphs, see `help("plotmath")` and Murrell and Ihaka (2000). Additional examples appear in [Chapter 9](#).

[Figure 3.15 \(a\)](#) is a typical distribution of a variable that represents the size of objects, what Tukey (1977) calls an *amount*. In this case, “size” is measured in millions of dollars, and focusing on percentage variation between values makes more sense than focusing on additive variation. Most of the data values are reasonably similar, but a few values are very large, and the distribution is consequently positively skewed. Working with a variable like this as either a response or as a predictor is difficult because the few large values become overly influential and the many small values become uninformative.

Logarithms spread out the small values and compress the large ones, producing

the more symmetric distribution in [Figure 3.15 \(b\)](#). The log transformation does not achieve perfect symmetry, and there is a suggestion that the distribution of the log-transformed variable has two, or possibly even three, modes, a property of the data that is disguised by the skew in [Figure 3.15 \(a\)](#). Nevertheless the log-transformed data are far better behaved than the untransformed data.<sup>24</sup>

[24](#) When as, in [Figure 3.15 \(b\)](#), there are multiple modes, the default bandwidth for the adaptive-kernel density estimator can be too large, suppressing detail in the distribution; setting `adjust=0.65`, a value determined by trial and error, decreases the bandwidth; see the discussion of density estimation in [Section 3.1.2](#). Adjusting the bandwidth doesn't substantially improve the density estimate of the untransformed data in [Figure 3.15 \(a\)](#).

Logarithms are sufficiently important in data analysis to suggest a rule: For any strictly positive variable with no fixed upper bound whose values cover two or more orders of magnitude (that is, powers of 10), replacing the variable by its log is likely to be helpful. Conversely, if the range of a variable is considerably less than an order of magnitude, then transformation by logarithms, or indeed any simple transformation, is unlikely to make much of a difference. Variables that are intermediate in range may or may not benefit from a log or other transformation.

Because of their ability to render many nonlinear relationships more nearly linear, logs can also be very helpful in transforming scatterplots. The UN data set in the **carData** package, for example, contains data obtained from the United Nations on several characteristics of 213 countries and other geographic areas recognized by the UN around 2010; the data set includes the infant mortality rate (`infantMortality`, infant deaths per 1,000 live births) and the per-capita gross domestic product (`ppgdp`, in U.S. dollars) of most of the countries.<sup>25</sup>

[25](#) There are only 193 countries in the UN data set with both variables observed. The R commands used here automatically reduce the data to the 193 fully observed cases on these two variables.

The graph in [Figure 3.16 \(a\)](#) simply plots the data as provided:

```
scatterplot (infantMortality ~ ppgdp, data=UN,
```

```
xlab="GDP per Capita",
```

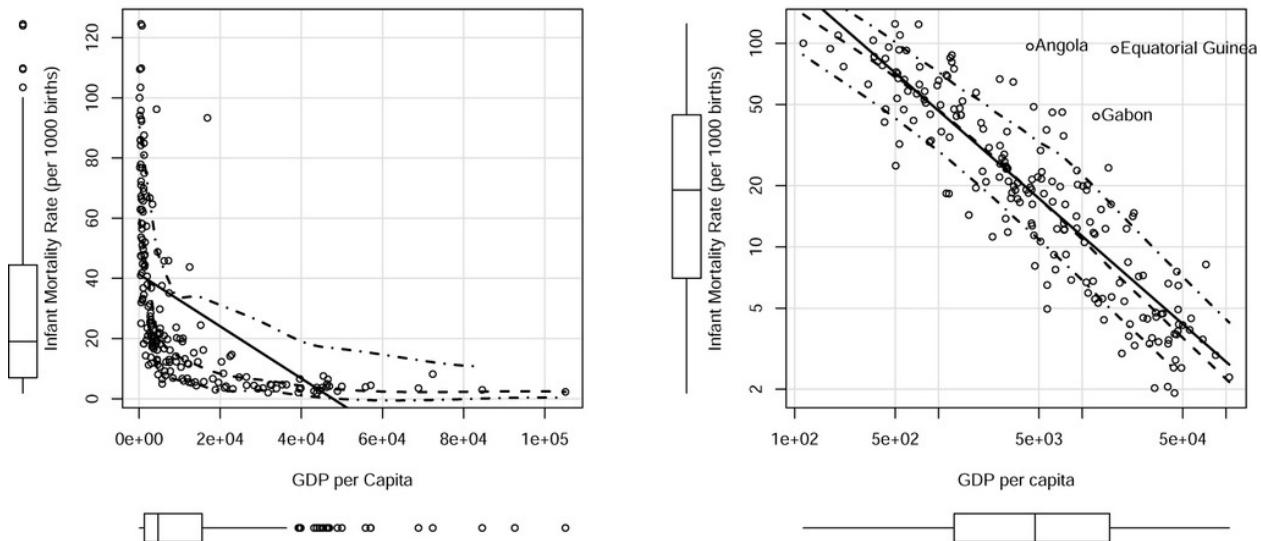
*ylab= "Infant Mortality Rate (per 1000 births)",*

*main= "(a)"*)

**Figure 3.16** Infant mortality rate and gross domestic product per capita, from the UN data set: (a) untransformed data and (b) both variables log-transformed.

(a)

(b)



The dominant feature of the scatterplot is that many of the places are very poor, as is clear in the marginal boxplot of ppgdp at the bottom of the graph. Infant mortality is also highly positively skewed. There is consequently very little visual resolution in the scatterplot, as nearly all the data points congregate at the far left of the graph, particularly in the lower-left corner. The loess smooth suggests that average infantMortality declines with ppgdp, steeply at first and then at a decreasing rate. Variation, shown by the variability smooths, is also larger among low-income places than among higher-income places.

In [Figure 3.16 \(b\)](#), we view the same data after taking logs of both variables:

```

scatterplot(infantMortality ~ ppgdp, data=UN,
            xlab="GDP per capita",
            ylab="Infant Mortality Rate (per 1000 births)",
            main="(b)", log="xy", id=list(n=3))

```

|        |                   |       |
|--------|-------------------|-------|
| Angola | Equatorial Guinea | Gabon |
| 4      | 54                | 62    |

The argument `log="xy"`, which also works with `plot()` and `scatterplot.Matrix()`, uses logs on both axes but labels the axes in the units of the original variables. The optional argument `id=list(n=3)` causes the three most “unusual” points according to the Mahalanobis distances of the data points to the means of the two variables to be labeled with their row names. These are three African countries that have relatively large infant mortality rates for their levels of per-capita GDP. The names of the identified points and their row numbers in the data set are printed in the R console.

The log-log plot tells a much clearer story than the original scatterplot, as the points now fall close to a straight line with a negative slope, suggesting that increasing `ppgdp` is associated with decreasing `infantMortality`. Because the variability smooths are essentially parallel to the loess smoother, conditional variation among countries is nearly constant across the plot, although as noted a few of the points are relatively far from the general linear pattern of the data. The marginal boxplots in [Figure 3.16](#) show that the univariate distributions of the log-transformed variables are much more symmetric than the distributions of the untransformed variables. Although it isn’t necessarily true that linearizing transformations also stabilize variation and make marginal distributions more symmetric, in practice, this is often the case.

From [Figure 3.16 \(b\)](#), we can posit a relationship between `infantMortality` and `ppgdp` given by a regression model of the form

Other

$$(3.1) \quad \log(\text{infantMortality}) = \beta_0 + \beta_1 \log(\text{ppgdp}) + \varepsilon$$

where  $\varepsilon$  is an additive *error*, a random variable with mean zero, constant variance, and a distribution that is independent of `infantMortality` given `ppgdp`.

Because we use natural logarithms in Equation 3.1, we can exponentiate both sides to get

Other

$$\begin{aligned}\text{infantMortality} &= \exp(\beta_0 + \beta_1 \log(\text{ppgdp}) + \varepsilon) \\ &= \exp(\beta_0) \times \text{ppgdp}^{\beta_1} \times \exp(\varepsilon) \\ &= \alpha_0 (\text{ppgdp}^{\beta_1}) \times \delta\end{aligned}$$

where we define  $\alpha_0 = \exp(\beta_0)$ , and  $\delta = \exp(\varepsilon)$  is a *multiplicative error* with typical value (actually, geometric mean)  $\exp(0) = 1$ . Additive errors in the log scale imply multiplicative errors in the original scale. According to this model, and with negative  $\beta_1$ , an increase in ppgdp is associated with a proportional decrease in infantMortality.

Fitting the linear regression model to the log-transformed data produces the following results, using the `brief()` function in the **car** package for a compact summary of the fitted model:

```
brief(lm(log(infantMortality) ~ log(ppgdp), data=UN))
```

|            | (Intercept) log(ppgdp) |         |
|------------|------------------------|---------|
| Estimate   | 8.104                  | -0.6168 |
| Std. Error | 0.211                  | 0.0247  |

Residual SD = 0.528 on 191 df, R-squared = 0.766

Larger values of ppgdp are associated with smaller values of infantMortality, as indicated by the negative sign of the coefficient for  $\log(\text{ppgdp})$ . For a hypothetical increase in ppgdp in a country of 1%, to  $1.01 \times \text{ppgdp}$ , the expected infantMortality would be (approximately, because we are ignoring the error  $\delta$ )

Other

$$\alpha_0 (1.01 \times \text{ppgdp})^{\beta_1} = 1.01^{\beta_1} \times \alpha_0 (\text{ppgdp}^{\beta_1})$$

For the estimated slope  $b_1 = -0.617$ , we have  $1.01^{-0.617} = 0.994$ , and so the estimated infantMortality would be 0.6% smaller—a substantial amount. Put another way, if we compare pairs of countries that differ by 1% in their ppgdp, on average the country with the 1% higher ppgdp will have a 0.6% lower infantMortality. Because we used the same base for the logs of both  $y$  and  $x$ , the percentage change in infant mortality corresponding to a 1% increase in per-capita GDP is approximately equal to the estimated regression coefficient,  $b_1 = -0.617$ . Economists call a coefficient such as  $\beta_1$  in a log-log regression an *elasticity*.

### 3.4.2 Power Transformations

The log-transformation rule introduced in [Section 3.4.1](#) appears to imply that numeric variables should generally be represented by the variable as measured or by its logarithm. Other transformations may be more appropriate in some problems. An example is the choice between fuel economy and fuel consumption mentioned at the beginning of [Section 3.4](#), suggesting using either the variable as measured or its inverse.

In regression analysis, transformations can help to achieve several goals:

- The first is the goal of approximate linearity. As we saw in [Figure 3.16](#) (page 152) for the UN data, the scatterplot of infant mortality versus percapita GDP has an approximately linear mean function in log-log scale lacking in the plot in arithmetic scale. Particularly in multiple linear regression, data with approximately linear mean functions for all 2D plots of the response and regressors, and among the regressors, are very useful (see, e.g., R. D. Cook & Weisberg, 1999, chap. 19).
- Transformations can also help to achieve more nearly constant conditional variation, as reflected in [Figure 3.16 \(b\)](#) by the variability smooths nearly parallel to the mean smooth in the scatterplot.
- Third, as we will see in examples later in the book, transformations can induce simpler structure and consequently models that are more easily understood.
- Finally, we can consider transformations toward normality, so that the transformed data are as close to normally distributed as possible. This can be useful for statistical methods such as factor analysis that depend on multivariate normality for their justification.

These goals can be in conflict, but in many problems achieving one of them helps with the others as well. In this section, we discuss systematic methodology for transforming one variable, or a group of variables, to be as close to normal or multivariate normal as possible. Later in the book (in [Section 8.4.1](#)), we will apply the methodology introduced here to transforming the response in linear regression problems.

We define a *simple power transformation* as the replacement of a variable  $x$  by  $x^\lambda$ , where, in the current general context,  $x$  can be either a response variable or a numeric predictor variable, and where  $\lambda$  is a power to be selected based on the data, usually in the range  $[-1, 1]$  but occasionally in the range  $[-3, 3]$ . The simple power transformations require that all elements of  $x$  be strictly positive,  $x > 0$ .

This use of power transformations is distinct from *polynomial regression*. In polynomial regression, the highest-order power, say  $p$ , is a small positive integer, usually 2 or 3, and rather than *replacing* a predictor  $x$  by  $x^p$ , we also include all lower-order powers of  $x$  in the regression model. For example, for  $p = 3$ , we would include the regressors  $x$ ,  $x^2$ , and  $x^3$ . Polynomial regression in R can be accomplished with the `poly()` function, discussed in [Section 4.4.1](#).

The power  $\lambda = -1$  is the reciprocal or inverse transformation,  $1/x$ . If, for example,  $x$  represents the time to completion of a task (e.g., in hours), then  $x^{-1}$  represents the rate of completion (tasks completed per hour). The power  $\lambda = 1/3$  can convert a volume measure (e.g., in cubic meters) to a linear measure (meters), which may be appropriate in some problems. Likewise,  $\lambda = 3$  can convert a linear measure (meters) to a volume (cubic meters);  $\lambda = 1/2$ , an area (square meters) to a linear measure (meters); and  $\lambda = 2$ , a linear measure (meters) to an area (square meters).

Selecting a transformation from a family of possible transformations was first proposed in a seminal paper by Box and Cox (1964).<sup>26</sup> Conspicuously absent from the power family is the all-important log transformation, and, to remedy this omission, Box and Cox introduced a family of *scaled power transformations* defined by

Other

(3.2)

$$T_{BC}(x, \lambda) = x^{(\lambda)} = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \text{when } \lambda \neq 0 \\ \log(x) & \text{when } \lambda = 0 \end{cases}$$

[26](#) *Box and Cox* is also the title of an operetta by Gilbert and Sullivan and an 1847 farce by John Maddison Morton, although the operetta and the play have nothing whatever to do with regression or statistics.

- For  $\lambda \neq 0$ , the scaled power transformations are essentially  $x^\lambda$ , because the scaled-power family only subtracts 1 and divides by the constant  $\lambda$ .
- One can show that as  $\lambda$  approaches zero,  $T_{BC}(x, \lambda)$  approaches  $\log(x)$ , and so the scaled-power family includes the log transformation when  $\lambda = 0$ , even though the literal zeroth power,  $x^0 = 1$ , is useless as a transformation.
- Also, the scaled power  $T_{BC}(x, \lambda)$  preserves the order of the  $x$ -values, while the basic powers preserve order only when  $\lambda$  is positive and *reverse* the order of  $x$  when  $\lambda$  is negative.

The scaled power family is generally called the *Box-Cox power family*. Usual practice is to use the Box-Cox power family to estimate the  $\lambda$  parameter of the transformation, but then to use in statistical modeling the equivalent simple power  $x^\lambda$  if the selected value of  $\lambda$  is far from zero, and  $\log(x)$  for  $\lambda$  close to zero. The details of the estimation method, which is similar to the method of maximum likelihood, are given in the complementary readings.

As a one-dimensional example, consider estimating a normalizing transformation of infantMortality in the UN data. The **car** function **powerTransform()** finds the transformation:

```
p1 <- powerTransform(infantMortality ~ 1, data=UN,
                      family="bcPower")
summary(p1)
```

```

bcPower Transformation to Normality
      Est Power Rounded Pwr Wald Lwr Bnd Wald Upr Bnd
Y1     0.0468          0       -0.0879       0.1814

Likelihood ratio test that transformation parameter is equal to 0
(log transformation)
      LRT df   pval
LR test, lambda = (0) 0.46446  1 0.496

Likelihood ratio test that no transformation is needed
      LRT df   pval
LR test, lambda = (1) 172.81  1 <2e-16

```

The first argument to `powerTransform()` is a formula, with the variable to be transformed on the left of the `~` and the variables, if any, for conditioning on the right. In this instance, there are no conditioning variables, indicated by `1` on the right, so we are simply transforming `infantMortality` unconditionally. The argument `family="bcPower"`, to specify the Box-Cox power family, isn't strictly necessary because "bcPower" is the default, but we supply the argument here for clarity. The first part of the output gives information about the estimated power. The value under `Est Power` is the point estimate found using the procedure suggested by Box and Cox. The second value, labeled `Rounded Pwr`, is a rounded version of the estimate, which we will discuss shortly. The remaining two values are the bounds of the 95% Wald confidence interval for  $\lambda$ , computed as  $\pm z_{\alpha/2} \hat{\sigma}_{\lambda}$ , where  $\hat{\sigma}_{\lambda}$  is the asymptotic standard error of the estimated power.

The point estimate here is `.0468`. The confidence interval includes  $\lambda = 0$ , which in turn suggests using a log-transform because of our preference for simple transformations. The value in the output labeled `Rounded Pwr` is the first value among `{1, 0, -1, .5, .33, -.5, -.33, 2, -2}` that is included in the confidence interval for  $\lambda$ ; if none of these values are in the confidence interval, then `0` is repeated in this column. In the example, zero is included in the confidence interval, suggesting the use in practice of the log transformation.

The remainder of the output provides tests concerning  $\lambda$ , first testing  $H_0: \lambda = 0$  for the log transformation and then testing  $H_0: \lambda = 1$ , which is equivalent to no transformation. The test for the log transformation has a very large  $p$ -value, indicating that the log transformation is consistent with the data, while the tiny  $p$ -value for  $\lambda = 1$  indicates that leaving `infantMortality` untransformed is

inconsistent with the goal of making the variable normally distributed. You can test for any other value of  $\lambda$ ; for example:

```
testTransform(p1, lambda=1/2)
```

|                         | LRT    | df | pval     |
|-------------------------|--------|----|----------|
| LR test, lambda = (0.5) | 41.958 | 1  | 9.32e-11 |

The very small  $p$ -value for the test of  $\lambda = 1/2$  suggests that we shouldn't use this transformation.

You can add a variable to the UN data set corresponding to a transformation by using the transform () function; for example:

```
UN <- transform(UN, infantMortality.tran=infantMortality^p1$lambda)
```

The updated UN data set contains all the variables in the original data set plus an additional variable called infantMortality.tran.<sup>27</sup> The value of p1\$lambda is the estimated power transformation parameter, while the value of p1\$roundlam is the rounded estimate.

<sup>27</sup> Updating the UN data set in this manner makes a copy of the data set in the global environment; it does not alter the UN data set in the **carData** package. See [Section 2.3.3](#).

## Multivariate Transformations

The multivariate extension of Box-Cox power transformations was provided by Velilla (1993) and is also implemented by the powerTransform () function in the **car** package. Given variables  $x_1, \dots, x_k$ , powerTransform () provides estimates of the power transformation parameters  $\lambda_1, \dots, \lambda_k$  such that the transformed variables are as close to multivariate normal as possible.

As an example, we return to the Canadian occupational-prestige data. A useful early step in any regression analysis is considering transformations toward linearity and normality of all the quantitative predictors, without at this point

including the response. The scatterplot matrix in [Figure 3.14](#) (page 147) suggests nonlinear relationships between some of the predictors and between the predictors and the response. We will not consider power-transforming women because it is a percentage, bounded between zero and 100, and because, from [Figure 3.14](#), women is at best weakly related to the other predictors and to the response. We will try to transform income and education toward bivariate normality:

```
summary(p2 <- powerTransform(cbind(income, education) ~ 1,
                                data=Prestige, family="bcPower"))
```

bcPower Transformations to Multinormality

|           | Est    | Power | Rounded | Pwr  | Wald | Lwr | Bnd     | Wald | Upr    | Bnd |
|-----------|--------|-------|---------|------|------|-----|---------|------|--------|-----|
| income    | 0.2617 |       |         | 0.33 |      |     | 0.0629  |      | 0.4604 |     |
| education | 0.4242 |       |         | 1.00 |      |     | -0.3663 |      | 1.2146 |     |

Likelihood ratio test that transformation parameters  
are equal to 0 (all log transformations)

|                         | LRT   | df | pval   |
|-------------------------|-------|----|--------|
| LR test, lambda = (0 0) | 7.694 | 2  | 0.0213 |

Likelihood ratio test that no transformations are needed

|                         | LRT    | df | pval     |
|-------------------------|--------|----|----------|
| LR test, lambda = (1 1) | 48.873 | 2  | 2.44e-11 |

Let  $\lambda = (\lambda_1, \lambda_2)'$  represent the vector of power transformation parameters.

Looking at the likelihood-ratio tests, untransformed variables, corresponding to  $\lambda = (1, 1)'$ , and all logarithms, corresponding to  $\lambda = (0, 0)'$ , both have small  $p$ -values, implying that neither of these choices is appropriate. The suggested procedure is to use the rounded values of the power parameters (1/3, 1) in further work. To verify that this choice is reasonable, we test the hypothesis that  $\lambda = (1/3, 1)'$ :

```
testTransform(p2, lambda=c(1/3, 1))
LRT df pval
LR test, lambda = (0.33 1) 2.4474 2 0.294
```

The nominal  $p$ -value is about 0.3, but interpreting this result is complicated by the use of the observed data to select the null hypothesis. As with a one-dimensional transformation, the value of and of the rounded estimates are stored respectively in `p2$lambda` and `p2$roundlam`.

Because we generally favor logarithms, let's also try  $\lambda = (0, 1)'$ :

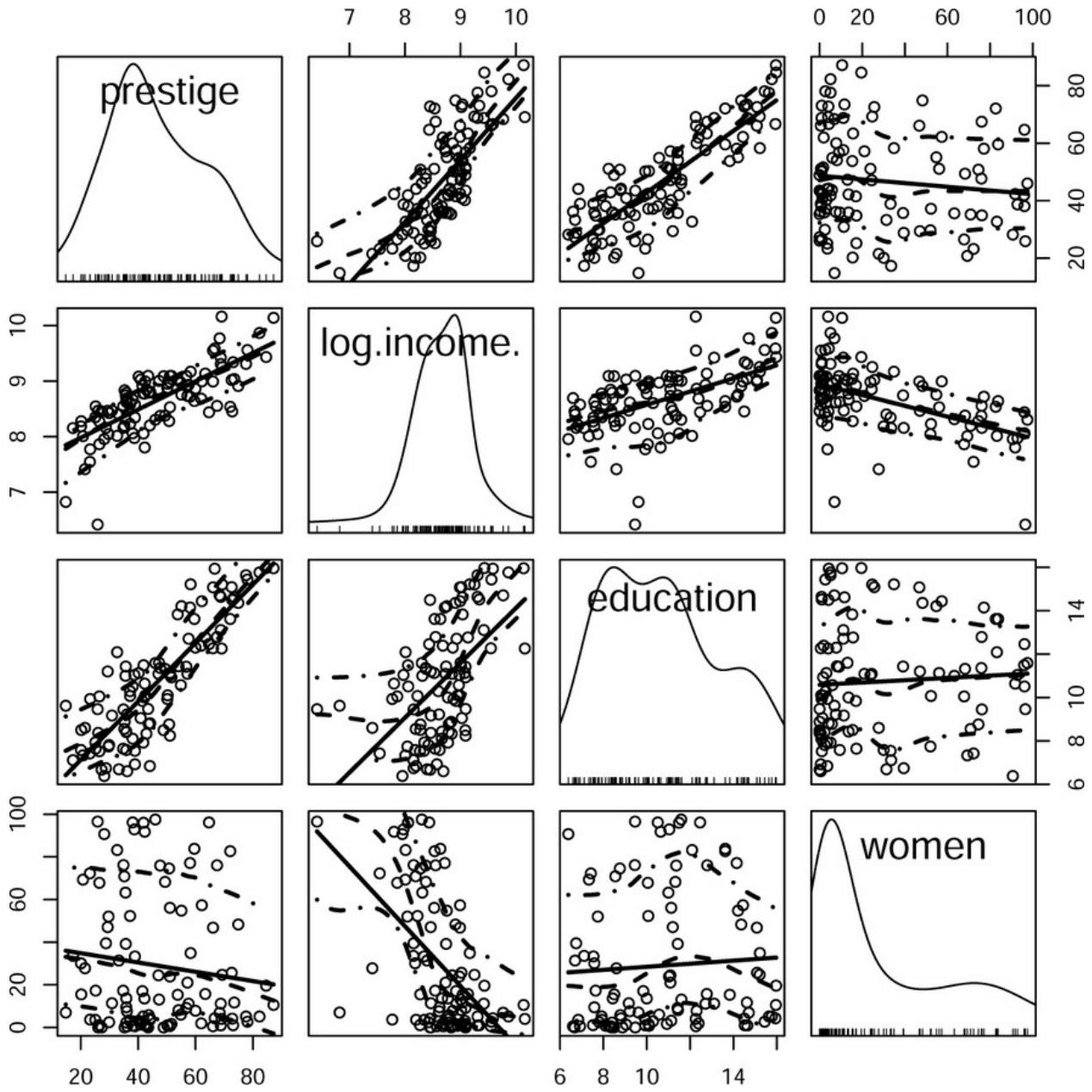
```
testTransform(p2, lambda=c(0, 1))
LRT df pval
LR test, lambda = (0 1) 9.2782 2 0.00967
```

Interpretation of the  $p$ -values is again difficult because the data were used to generate the hypothesis. The relatively small  $p$ -value suggests that the choice of  $\lambda = (1/3, 1)$  is to be preferred to  $\lambda = (0, 1)$ . A prudent approach would be to repeat the analysis employing both cube root and log transformations of income to see if any noteworthy differences result.

Applying the log transformation of income produces the scatterplot matrix in [Figure 3.17](#):

```
scatterplotMatrix (~ prestige + log (income) + education + women,
smooth=list(span=0.7), data=Prestige)
```

**Figure 3.17** Scatterplot matrix for variables in the Prestige data set, with log-transformed income. The `scatterplotMatrix()` function converts the variables on the diagonal panel to standard R names, changing “`log (income)`” to “`log.income`.”.



The panels in this scatterplot matrix show little nonlinearity, although in the plot of prestige versus log (income) the two occupations with the lowest incomes appear not to fit the pattern in the rest of the data. We invite the reader to redraw the graph using the cube root of income.

## Conditioning on Groups

The variable type in the Prestige data set is a factor that divides occupations into blue-collar ("bc"), white-collar ("wc"), and professional/managerial ("prof")

categories. Examination of the data suggests that the distributions of the numeric variables in the data set vary by occupational type. We can then seek transformations of the numeric variables that make their within-group distributions as close as possible to multivariate normal with common covariance structure (i.e., similar variances and covariances within types of occupations). For example, for the joint distribution of income and education within levels of type, we have

```
summary(p3 <- powerTransform(cbind(income, education) ~ type,
data=Prestige))
bcPower Transformations to Multinormality
      Est Power Rounded Pwr Wald Lwr Bnd Wald Upr Bnd
income     -0.0170          0     -0.2738      0.2398
education   0.7093          1      0.1318      1.2867

Likelihood ratio test that transformation parameters
are equal to 0 (all log transformations)
      LRT df    pval
LR test, lambda = (0 0) 5.7228  2 0.0572

Likelihood ratio test that no transformations are needed
      LRT df    pval
LR test, lambda = (1 1) 64.02  2 1.25e-14

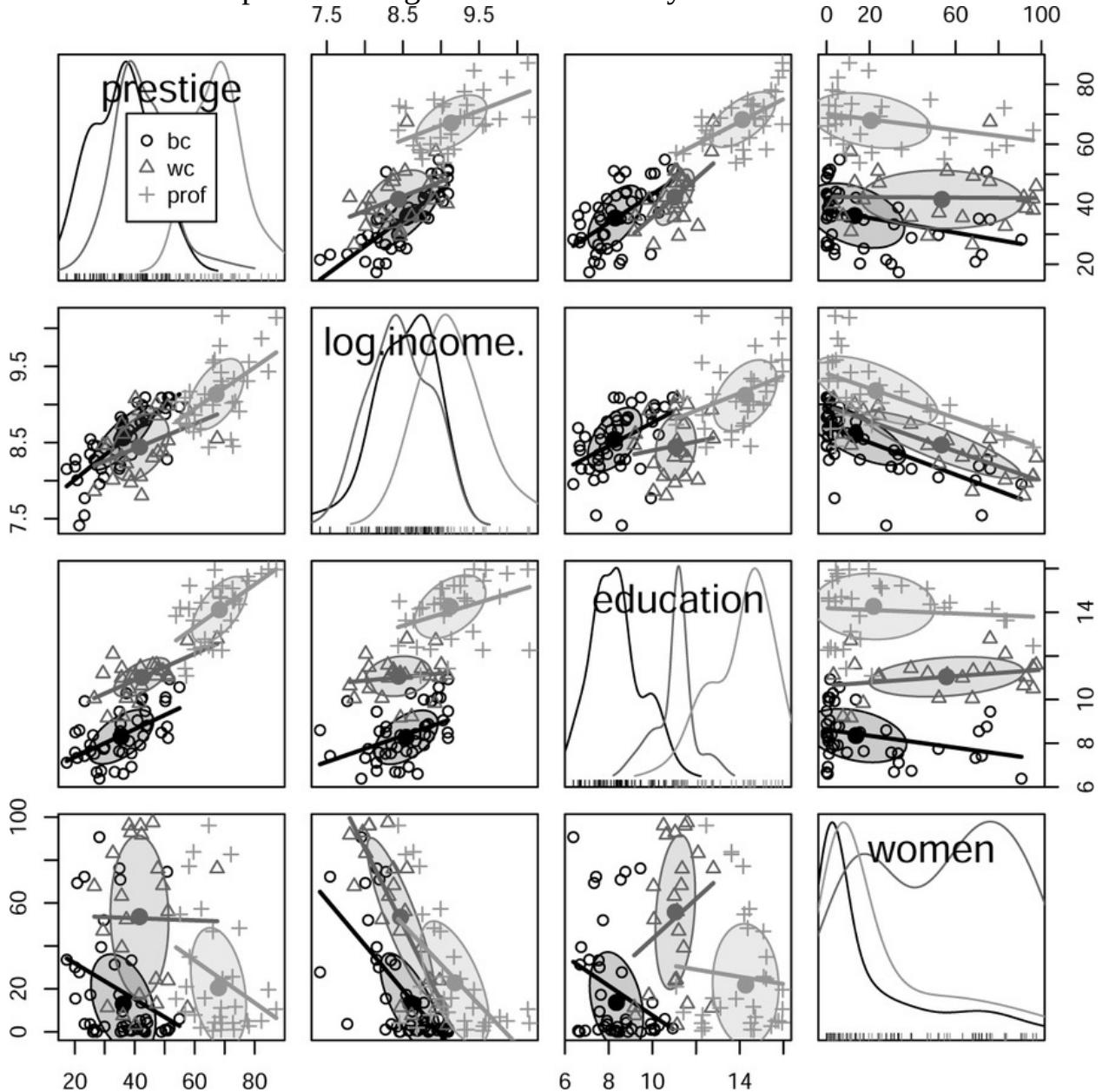
testTransform(p3, c(0, 1))
      LRT df    pval
LR test, lambda = (0 1) 0.98381  2 0.611
```

After conditioning on type, the log transformation of income is clearly indicated, while education is untransformed, producing the revised scatterplot matrix in [Figure 3.18](#):

```
scatterplotMatrix (
~ prestige + log (income) + education + women | type,
```

```
data=Prestige, smooth=FALSE, ellipse=list (levels=0.5),
legend=list (coords="center"))
```

**Figure 3.18** Scatterplot matrix with log-transformed income and conditioning on type of occupation, for the Prestige data set. The ellipse in each panel is a 50% concentration ellipse assuming bivariate normality.



We suppress the loess smooths with the argument `smooth=FALSE` to minimize clutter in the graph and move the legend from its default position in the top-right

corner of the first diagonal panel to the center of the panel. Approximate within-group linearity is apparent in most of the panels of [Figure 3.18](#).

We set the argument `ellipse=list(levels=0.5)` to get separate 50% concentration ellipses for the groups in the various off-diagonal panels of the plot (see, e.g., Friendly, Monette, & Fox, 2013): If the data in a panel are bivariately normally distributed, then the ellipse encloses approximately 50% of the points. The size of the ellipse in the vertical and horizontal directions reflects the SDs of the two variables, and its tilt reflects their correlation. The within-group variability of prestige, log (income), education, and women is therefore reasonably similar across levels of type; the tilts of the ellipses in each panel are somewhat more variable, suggesting some differences in within-group correlations. The least-squares lines in the panels also show some differences in within-group slopes.

The plotted points are the same in [Figures 3.17](#) and [3.18](#)—only the symbols used for the points, and the fitted lines, are different. In particular, the two points with low income no longer appear exceptional when we control for type.<sup>28</sup>

[28](#) A small difference between the two graphs is that type is missing for four occupations, and thus [Figure 3.18](#) has only 98 of the 102 points shown in [Figure 3.17](#).

## Transformations With a Few Negative or Zero Values

We may wish to apply the Box and Cox transformation methodology to a variable with a small number of zero or even negative values. For instance, a few selfemployed individuals in a data set may incur expenses that cancel out their earnings or that result in a net loss. To take another example, 28 of the 240 companies in the Ornstein data set maintained zero interlocking directors and officers with other firms; the numbers of interlocks for the remaining companies were positive, ranging between 1 and 107.

A simple expedient when there are some nonpositive values of a variable  $x$  is to add a small constant, say  $\delta$ , called a *start* by Mosteller and Tukey (1977), to  $x$ , so that  $x + \delta$  is always positive. The Box-Cox procedure is then applied to  $x + \delta$  rather than to  $x$  itself. The difficulty with this approach is that if  $\delta$  is chosen so  $\min(x) + \delta$  is too small, then the procedure can produce outliers or influential cases, making results unreliable. If  $\delta$  is chosen to be too large, then power transformations of  $x + \delta$  may prove ineffective. The resulting fitted model,

including parameter estimates, is potentially hard to interpret, and results may not be applicable to a different data set.

As an alternative, suppose that we let . The *Box-Cox family with negatives* (Hawkins & Weisberg, 2017) is defined to be the Box-Cox power family applied to  $z(x,y)$  rather than to  $x$ :  $T_{BCN}(x, \lambda, y) = T_{BC}(z(x,y), \lambda)$ . If  $x$  is strictly positive and  $\gamma = 0$ , then  $z(x, 0) = x$  and the transformation reduces to the regular Box-Cox power family. The Box-Cox family with negatives is defined for any  $\gamma > 0$  when  $x$  has nonpositive entries. The important improvement of this method over just adding a start is that for any  $\gamma$  and large enough  $x$ , the transformed values with power  $\lambda$  can be interpreted as they would be if the Box-Cox family had been used and no nonpositive data were present.

We illustrate using `powerTransform()` with the "bcnPower" family (which invokes the `bcnPower()` function in the `car` package) for the number of interlocking directors in Ornstein's data:

```
p4 <- powerTransform(interlocks ~ 1, data=Ornstein,
                      family="bcnPower")
summary(p4)
```

bcnPower transformation to Normality

Estimated power, lambda

|    | Est | Power  | Rounded | Pwr | Wald | Lwr | Bnd | Wald   | Upr | Bnd    |
|----|-----|--------|---------|-----|------|-----|-----|--------|-----|--------|
| Y1 |     | 0.2899 |         |     | 0.33 |     |     | 0.2255 |     | 0.3543 |

Location gamma was fixed at its lower bound

|    | Est | gamma | Std Err. | Wald | Lower Bound | Wald | Upper Bound |    |
|----|-----|-------|----------|------|-------------|------|-------------|----|
| Y1 |     | 0.1   |          | NA   |             | NA   |             | NA |

Likelihood ratio tests about transformation parameters

|                       |         | LRT | df | pval |
|-----------------------|---------|-----|----|------|
| LR test, lambda = (0) | 86.829  | 1   | 0  |      |
| LR test, lambda = (1) | 328.124 | 1   | 0  |      |

Employing the "bcnPower" family estimates both the power  $\lambda$  and the shift  $\gamma$  transformation parameters using Box and Cox's maximum-likelihood-like method. As is often the case with the "bcnPower" family, the estimate of the shift  $\gamma$  is very close to the boundary of zero, in which case `powerTransform()` fixes  $\gamma = 0.1$  and then estimates the power  $\lambda$ . In the example,  $\gamma$  is close to the cube root. The tests shown in the output are for the fixed value of  $\gamma = 0.1$ ; had  $\gamma$  been estimated, the tests would have been based on the log-likelihood profile averaging over  $\gamma$ . We will return to this example in [Section 8.4.1](#).

### 3.4.3 Transformations and Exploratory Data Analysis

Tukey (1977) and Mosteller and Tukey (1977) promoted a collection of methods that can be used to understand data, mostly without requiring models, and that have come to be known as *exploratory data analysis* (or *EDA*). In particular, Tukey and Mosteller advocated graphical examination of data, at a time when computing was generally much slower than now, much more expensive, and much more difficult. As a result, many of Mosteller and Tukey's proposed methods are based on simple ideas that can even be carried out on the proverbial “back of an envelope.” The **car** package implements a few EDA methods for selecting a transformation. These methods are of some continuing utility, and they can also help us to understand how, why, and when transformations work.

Mosteller and Tukey characterize the family of power transformations  $x^\lambda$  as a *ladder of powers and roots*, with no transformation,  $\lambda = 1$ , as the central rung of the ladder. Transformations like  $\lambda = 1/2$  (square root),  $\lambda = 0$  (taken, as in the Box-Cox family, as the log transformation), and  $\lambda = -1$  (the inverse transformation) entail moving successively down the ladder of powers, and transformations like  $\lambda = 2$  (squaring) and  $\lambda = 3$  (cubing) entail moving up the ladder.

#### Transforming for Symmetry

Transformation down the ladder of powers serves to spread out the small values of a variable relative to the large ones and consequently can serve to correct a positive skew. Negatively skewed data are much less common but can be made more symmetric by transformation *up* the ladder of powers.

The `symbox()` function in the **car** package implements a suggestion of Tukey and Mosteller to induce approximate symmetry by trial and error<sup>29</sup>—for

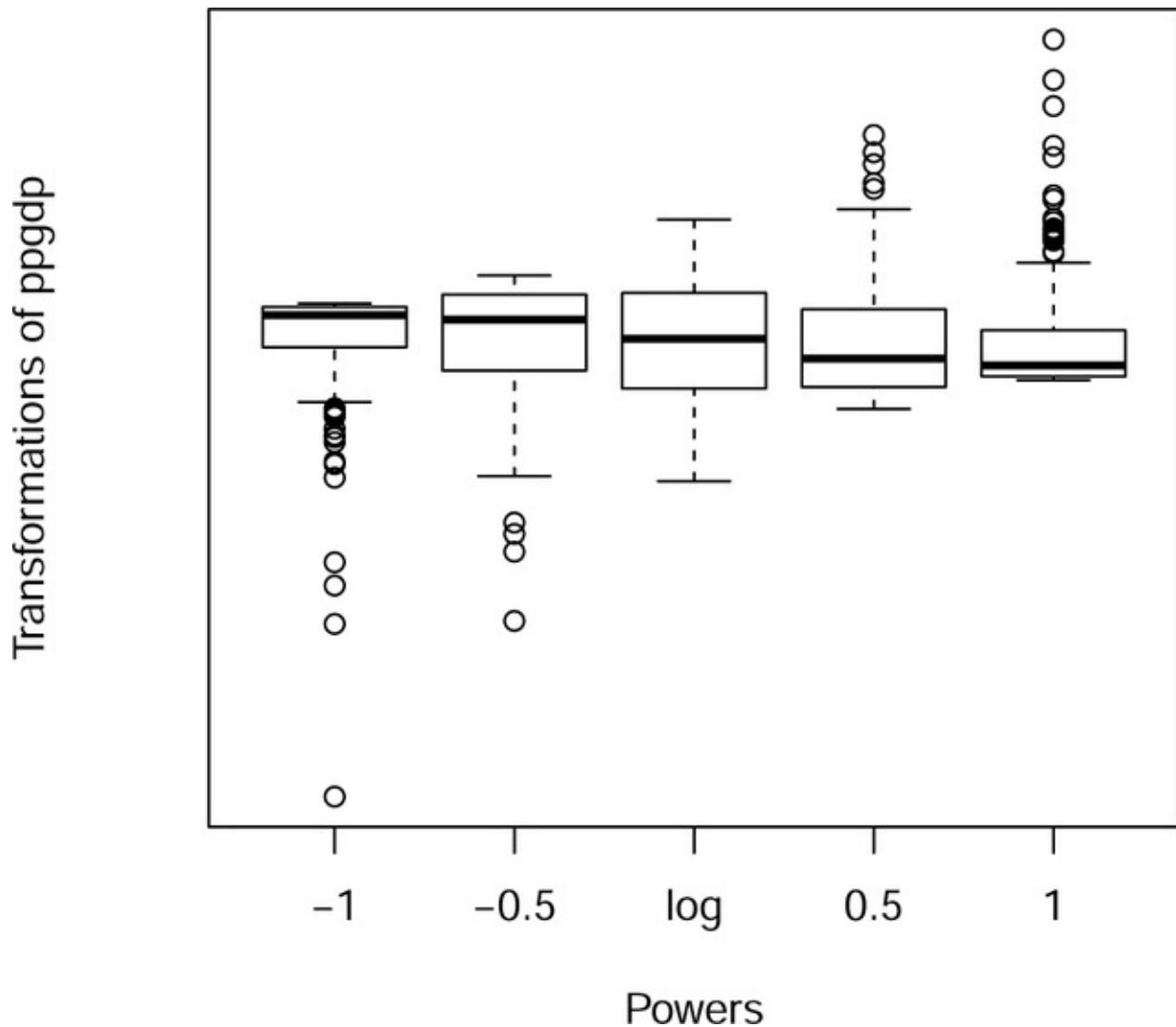
example, for `ppgdp` (per-capita GDP) in the UN data:

**`symbox (~ ppgdp, data=UN)`**

[29](#) The `symbox ()` function was inspired by a SAS macro designed by Michael Friendly.

The resulting graph is shown in [Figure 3.19](#). By default, `symbox ()` uses the function `bcPower ()` and displays boxplots of the transformed variable for several transformations down the ladder of powers; here, the ubiquitous log transformation of `ppgdp` does the best job of making the distribution of the variable symmetric.

**Figure 3.19** Boxplots of various power transformations of per-capita GDP in the United Nations data.

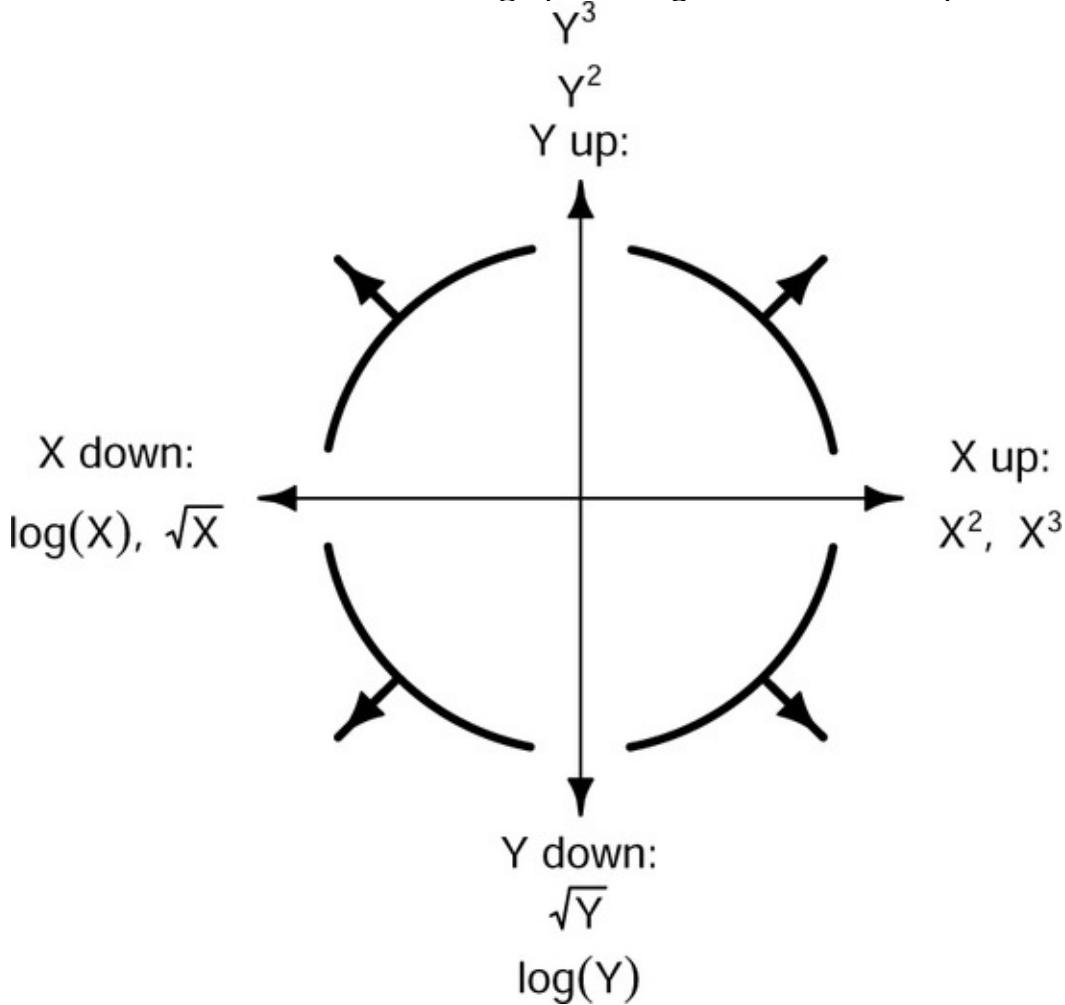


## Transforming Toward Linearity

If a relationship is nonlinear but monotone (i.e., strictly increasing or strictly decreasing) and simple (in the sense that the curvature of the regression function is relatively constant), then Mosteller and Tukey's *bulging rule* (Mosteller & Tukey, 1977; also Tukey, 1977), illustrated in [Figure 3.20](#), can guide the selection of linearizing power transformations of the two variables. When the bulge of the regression function in a scatterplot points *down* and to the *right*, for example—as suggested by the line in the lower-right quadrant of [Figure 3.20](#)—we move  $y$  down the ladder of powers,  $x$  up the ladder of powers, or both, to straighten the relationship between the two variables. The bulging rule generalizes to the other quadrants of [Figure 3.20](#).

**Figure 3.20** Mosteller and Tukey's *bulging rule* for finding linearizing

transformations: When the bulge points *down*, transform  $y$  *down* the ladder of powers; when the bulge points *up*, transform  $y$  *up*; when the bulge points *left*, transform  $x$  *down*; when the bulge points *right*, transform  $x$  *up*.



Source: Fox (2016, Figure 4.7).

To illustrate, in [Figure 3.16 \(a\)](#) (page 152), the nonlinear relationship between infantMortality and ppgdp is simple and monotone; the bulge in the scatterplot points down and to the left, suggesting the transformation of infant-Mortality or ppgdp, or both, down the ladder of powers. In this case, the log transformation of both variables produces a nearly linear relationship, as shown in [Figure 3.16 \(b\)](#).

## Transforming to Equalize Spread

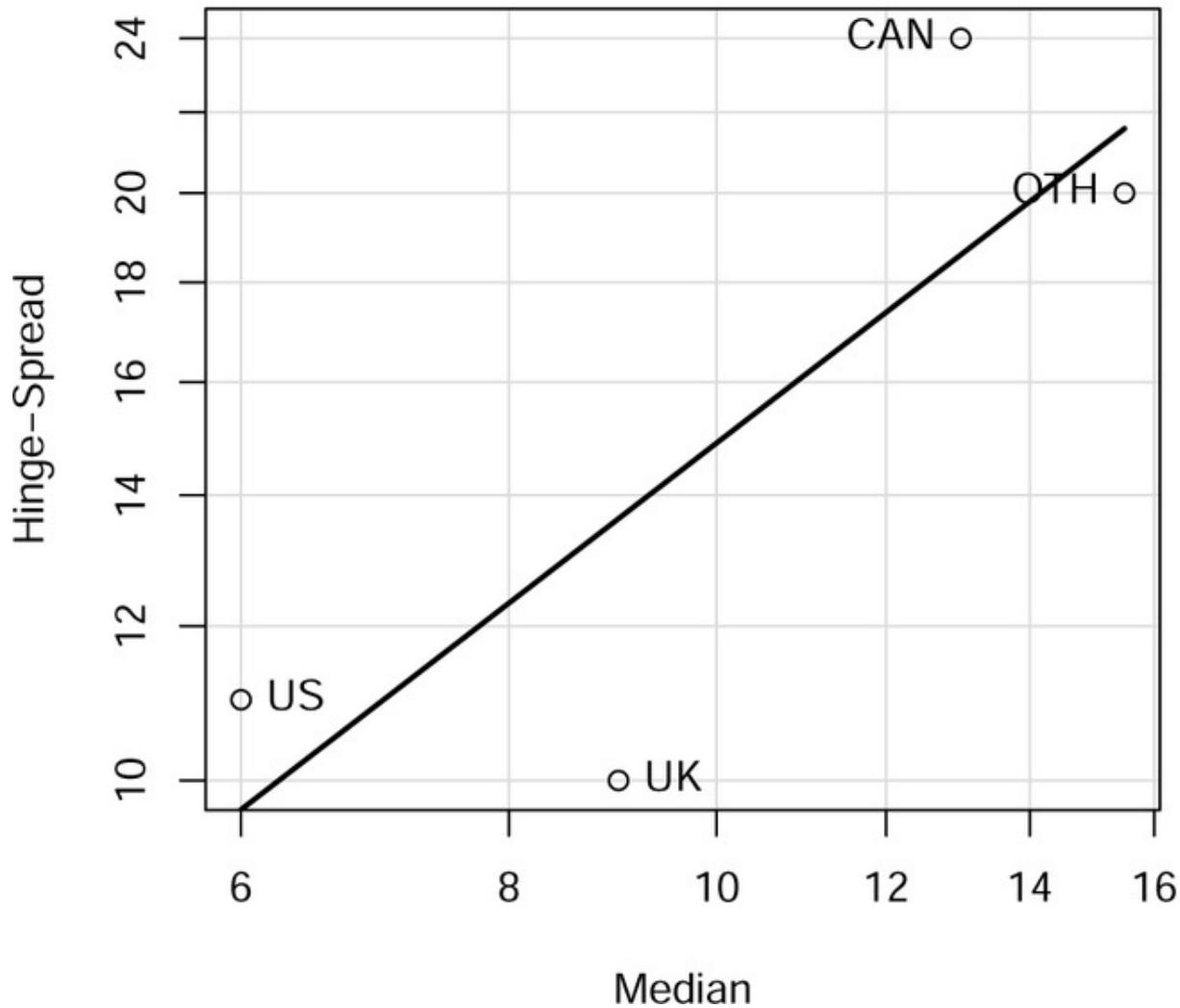
In [Figure 3.11](#) (page 142), we examined the relationship between number of interlocks and nation of control among the 248 large Canadian corporations in Ornstein's interlocking-directorate data. That graph revealed an association between *level* and *spread*: Nations with a relatively high level of interlocks (Canada, other) evidence more variation than nations with fewer interlocks on average (United States, United Kingdom). The term *level* is meant to imply an average or typical value, such as the mean, and *spread* is meant to imply variability, as measured, for example, by the SD. These terms are used in exploratory data analysis in place of more technical terms such as mean and SD because level and spread may be measured using other statistics that are resistant to outlying values.

A *spread-level plot* (Tukey, 1977) is a scatterplot of the logarithm of the interquartile range, which measures spread, versus the logarithm of the within-group median, which measures level. Interquartile ranges and medians are insensitive to a few outliers, and so the spread-level plot provides a robust representation of the dependence of spread on level.

Using the `spreadLevelPlot()` function in the **car** package produces a graph, such as [Figure 3.21](#) for the Ornstein data:

**Figure 3.21** Spread-level plot for the relationship between number of interlocks and nation of control in Ornstein's interlocking-directorate data.

## Spread–Level Plot for interlocks + 1 by nation



```
spreadLevelPlot(interlocks + 1 ~ nation, data=Ornstein)
```

|     | LowerHinge | Median | UpperHinge | Hinge-Spread |
|-----|------------|--------|------------|--------------|
| US  | 2          | 6.0    | 13         | 11           |
| UK  | 4          | 9.0    | 14         | 10           |
| CAN | 6          | 13.0   | 30         | 24           |
| OTH | 4          | 15.5   | 24         | 20           |

Suggested power transformation: 0.15345

If there is an association between spread and level in the spread-level plot, Tukey suggested fitting a straight line to the plot, with estimated slope  $b$ , and then transforming the response variable  $y$  by the power transformation  $y^{1-b}$ ; if  $b \approx 1$ , then a log transformation is used. Because Tukey's method requires a strictly positive response variable, we add a start of 1 to interlocks to avoid zero values. In addition to drawing a graph, the `spreadLevelPlot()` function prints a table showing the first quartile (Tukey's *lower hinge*), median, third quartile (*upper hinge*), and interquartile range (*hinge spread*) in each group, along with the suggested power transformation of  $\text{interlocks} + 1$ . In the example, the suggested power is  $p = 0.15$ , close to the log transformation.

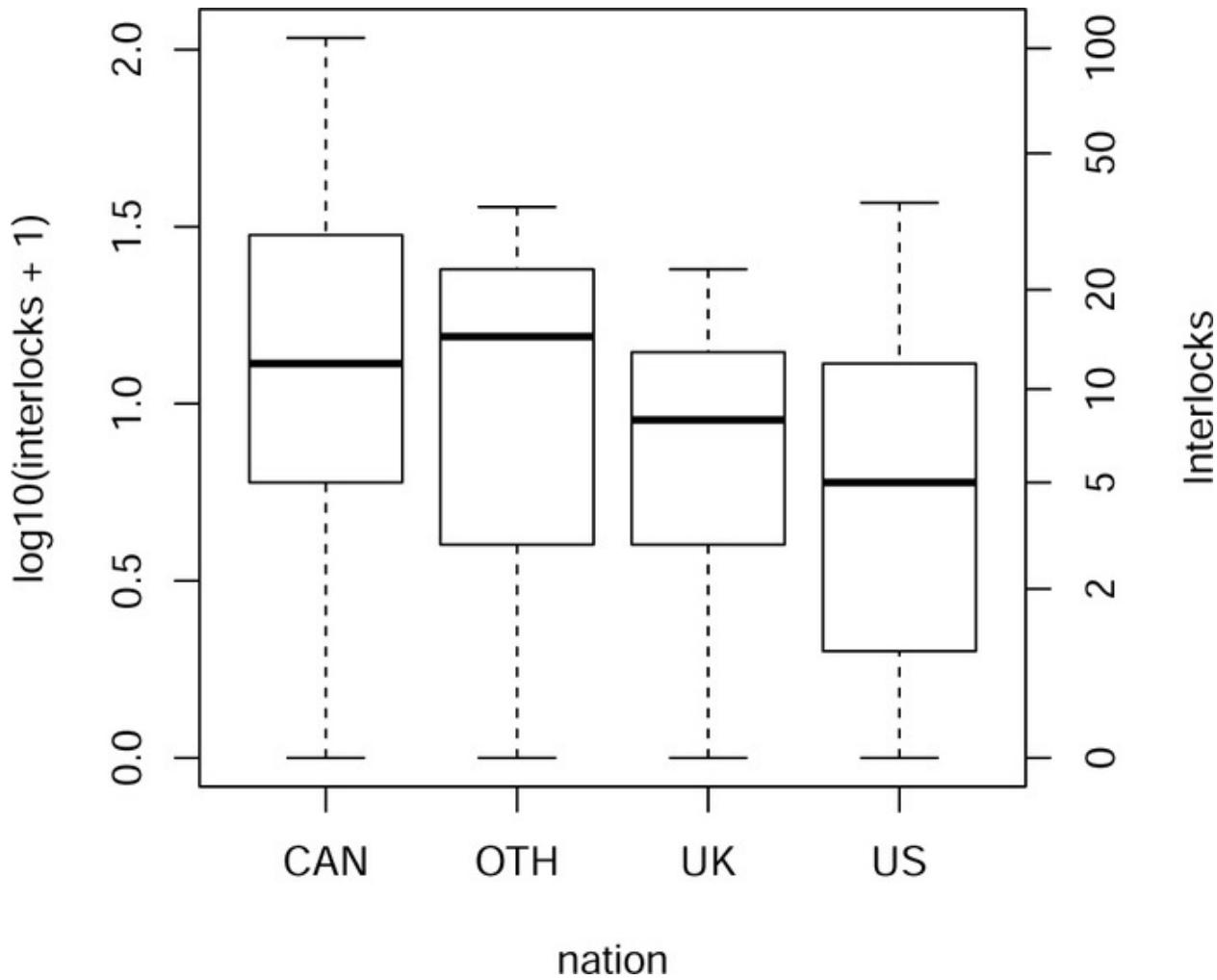
[Figure 3.22](#) shows parallel boxplots for the log-transformed data:

```
par (mar=c (5.1, 4.1, 4.1, 4.1))
```

```
Boxplot (log10(interlocks + 1) ~ nation, data=Ornstein)
```

```
basicPowerAxis (power=0, base=10, at=c (1, 3, 6, 11, 21, 51, 101), start=1,
axis.title="Interlocks")
```

**Figure 3.22** Parallel boxplots for  $\log_{10}(\text{interlocks} + 1)$  by nation of control.



The spreads in the transformed data for the four groups are much more similar than the spreads in the untransformed data (shown in [Figure 3.11](#) on page 142).

Many global graphics parameters in R are set or queried by the `par()` function. The "mar" setting is for the plot margins; see `help("par")` for details.<sup>30</sup> We specify two arguments in the call to `Boxplot()`: a formula with the base-10 logarithm of `interlocks + 1` on the left-hand side and the factor `nation` on the right-hand side, and the argument `data=Ornstein`, supplying the data frame in which these variables reside. The remaining commands make the graph more elaborate, first by increasing the right-side margin of the plot and then by adding a second axis on the right, labeled in the original units of `interlocks`, produced by the `basicPowerAxis()` function in the **car** package. The argument `power=0` to `basicPowerAxis()` specifies the log transformation; `base=10` gives the base used for the logs; `at=c(1, 3, 6, 11, 21, 51, 101)` indicates where the tick marks are to appear on the `interlocks + 1` scale; and `start=1` gives the start that was used, so

that the tick labels can be adjusted to the original interlocksscale.<sup>31</sup>

<sup>30</sup> Graphics parameters are also discussed in [Section 9.1.2](#).

<sup>31</sup> The functions `bcPowerAxis()`, `bcnPowerAxis()`, and `probabilityAxis()` in the `car` package may be used similarly to produce axes on the untransformed scale corresponding to Box-Cox, Box-Cox with negatives, and logit transformations.

### 3.4.4 Transforming Restricted-Range Variables

Variables with ranges bounded both below and above may require different transformations because of ceiling or floor effects, in which values tend to accumulate near the boundaries of the variable. The most common kinds of restricted-range variables are percentages and proportions, but other variables, such as the number of questions correct on a test of fixed length, are also “disguised proportions.”

Because small changes in a proportion near zero or 1 are potentially much more important than small changes near 0.5, promising transformations spread out both large and small proportions relative to proportions near 0.5. If a proportion has a more limited observed range, say between 0.25 and 0.75, transformations are unlikely to be helpful. The idea is to transform proportions so the transformed values behave more like a sample from a normal distribution. Usual methods of statistical inference for normally distributed data can then be used with the transformed proportions as the response.

Statistical models specifically designed for proportion data—for example, logistic-regression and loglinear models (discussed in [Chapter 6](#))—are the preferred methods for working with proportions, and so transformation for normality is now rarely used with proportion response variables if the counts on which the proportions are based are available. Transforming predictor variables that are expressed as proportions or percents may prove useful.

Suppose that  $u_i$  is a sample count of number of “successes” for the  $i$ th of  $n$  cases and that  $N_i$  is the corresponding number of trials. Each  $u_i$  can have its own probability of success  $\theta_i$ . From the binomial distribution, the standard deviation of the sample proportion  $x_i = u_i/N_i$  is , and therefore varies depending on the

value of  $\theta_i$ . If we transform  $x_i = u_i / N_i$  using the *arcsine square root*,

Other

$$T_{\text{asinsqrt}}(x) = \sin^{-1}(\sqrt{x})$$

computed in R as, for example,

```
asin (sqrt (seq (0, 1, length=11)))
```

```
[1] 0.00000 0.32175 0.46365 0.57964 0.68472 0.78540 0.88608
```

```
[8] 0.99116 1.10715 1.24905 1.57080
```

then the SD of the transformed variable is approximately for any value of  $\theta_i$ , and hence,  $T_{\text{asinsqrt}}(x)$  is a variance-stabilizing transformation.

Logistic regression uses a different transformation of a proportion called the *logit* or *log-odds* transformation,

Other

$$T_{\text{logit}}(x) = \text{logit}(x) = \log\left(\frac{x}{1-x}\right)$$

The logit transformation is not defined for sample proportions exactly equal to zero or 1, however.<sup>32</sup> We can get around this limitation if we want to use sample logits in a graph or statistical analysis by remapping proportions from the interval [0, 1] to [0.025, 0.975], for example, taking the logit of  $0.025 + 0.95 \times x$  rather than the logit of  $x$ . The `logit()` function in the **car** package takes care of remapping proportions or percentages when there are zeros or ones—or 0% or 100% for percentage data—printing a warning if remapping is required:

```

logit(seq(0.1, 0.9, 0.1))
[1] -2.19722 -1.38629 -0.84730 -0.40547  0.00000  0.40547
[7]  0.84730  1.38629  2.19722

logit(seq(0, 1, 0.1))
[1] -3.66356 -1.99243 -1.29505 -0.80012 -0.38467  0.00000
[7]  0.38467  0.80012  1.29505  1.99243  3.66356
Warning message:
In logit(seq(0, 1, 0.1)) : proportions remapped to (0.025, 0.975)

```

[32](#) This restriction does not affect logistic regression, where it is the *probability* of success $\theta$ , and not the observed sample *proportion* of successes  $x$ , that is transformed.

Even better, if we have access to the original data from which the proportions are calculated, we can avoid proportions of zero or 1 by computing *empirical logits*,  $\log[(u + 1/2)/(N + 1)]$ , where, as before,  $u$  is the number of successes in  $N$  trials.

To illustrate, we apply the logit and arcsine-square-root transformations to the distribution of the gender composition of occupations (percentage women) in the Prestige data set:

```
par(mfrow=c(1, 3))
```

```
densityPlot (~ women, data=Prestige, from=0, to=100, main="(a)  
Untransformed")
```

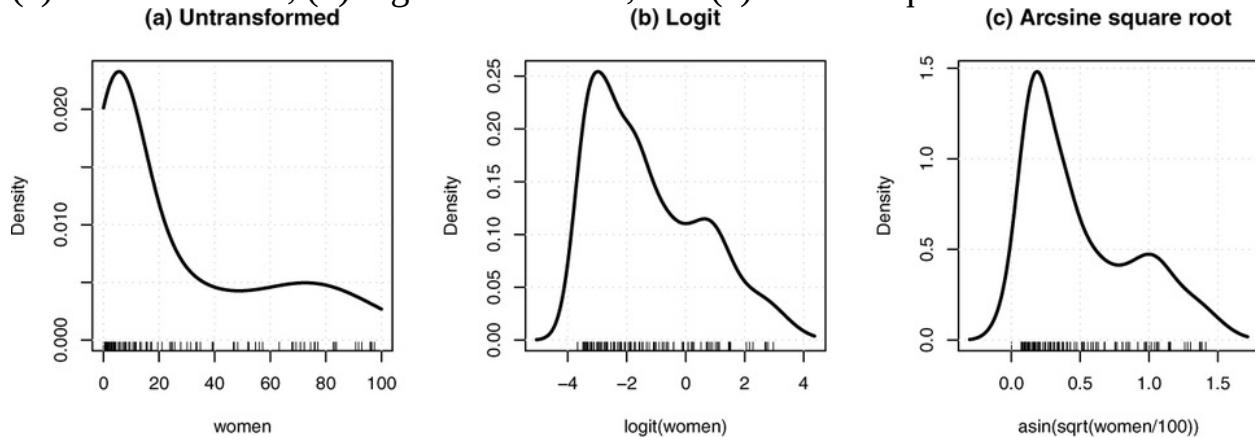
```
densityPlot (~ logit(women), data=Prestige, adjust=0.7, main="(b)  
Logit")
```

```
densityPlot (~ asin(sqrt(women/100)), data=Prestige, adjust=0.7,  
main="(c) Arcsine square root")
```

The resulting density plots are shown in [Figure 3.23](#). The density plot for the untransformed percentages is confined to the domain zero to 100. The untransformed data stack up near the boundaries, especially near zero. The logit-transformed data appear better behaved, and the arcsine-square-root transformed data are similar. We adjust the bandwidth of the density estimators (see [Section](#)

[3.1.2](#)) for the transformed data to resolve the two peaks in the distribution more clearly. As mentioned previously, the default bandwidth is often too large when there are multiple modes.

**Figure 3.23** Distribution of women in the Canadian occupational-prestige data: (a) untransformed, (b) logit transformed, and (c) arcsine square root transformed.



## 3.4.5 Other Transformations

The transformations that we have discussed thus far are hardly a complete catalog, and not all transformations are selected empirically. Here are a few other possibilities:

- People's height in cm and weight in kg are often combined in the *body mass index*,  $BMI = \text{weight}/(\text{height}^2)$ , which is intended to approximate body composition in units of  $\text{kg}/\text{cm}^2$ .
- A variable like `month.number` can be replaced in a regression model by two regressors,  $\sin(\text{month.number}/12)$  and  $\cos(\text{month.number}/12)$ , to model seasonal time trends.
- In a study of highway accident rates, a variable giving the number of traffic signals in a highway segment is converted to the number of signals per mile by dividing by the length of the segment. More generally, measures of total size often need to be converted to size per unit, for example converting GDP to a per-capita basis by dividing by population.

The upshot of these examples is that one should think carefully about how variables are expressed in the substantive context in which they are used in research.

## 3.5 Point Labeling and Identification

Identifying extreme points can be especially valuable in graphs used for model building and diagnostics. The `identify()` function in base R, which we describe in [Section 3.5.1](#), has a clumsy interface for interactive marking of interesting points. Most of the graphics functions in the `car` package include an `id` argument that permits both interactive and automatic marking of points. We describe what the `id` argument does in [Section 3.5.2](#).

### 3.5.1 The `identify()` Function

The `identify()` function is called after drawing a plot to add labels to the plot at points selected interactively using the mouse. For example, [Figure 2.1](#) (page 75) could have been drawn using

***with (Freedman, plot (density, crime))***

To identify points on the resulting scatterplot, enter the following command at the prompt in the R console:

***with (Freedman, identify (density, crime, row.names (Freedman)))***

The call to the `identify()` function usually has at least three arguments: the horizontal coordinates of the plotted points, the vertical coordinates of the points, and a vector of labels for the points. If the third argument is absent, then the points are labeled by number. The `identify()` function has several other arguments that can modify the size and location of the labels that are printed: See help ("`identify`").

Left click near a plotted point in the RStudio *Plots* tab to label the point; a “pin” icon flashes momentarily near the point, but point labels aren’t shown until you exit from point identification mode. Labeling points interactively continues until you explicitly stop it. The message *Locator active (Esc to finish)*, at the upper left of the RStudio *Plots* tab, reminds you of this fact. A warning: Having to exit from point identification mode is a potential source of trouble. You can’t enter

additional commands until you press the Esc key or the *Finish* button at the upper right of the *Plots* pane. Furthermore, because `identify()` is interactive, you can't use it conveniently in R Markdown documents.

### 3.5.2 Automatic Point Labeling

The graphics functions in the **car** package employ a common general mechanism for point identification, using the `showLabels()` function to identify potentially noteworthy points. Point identification is controlled by the `id` argument, which takes the values `TRUE`, `FALSE`, or a detailed list of specifications; point labeling can be either automatic or interactive depending on how the `id` argument is set. In most cases, the user only needs to set `id=TRUE` or to specify the number of points to be identified—for example, `id=list(n=5)`. Setting `id=FALSE` (the default for most **car** graphics functions) suppresses point labeling, and `id=TRUE` is in most cases equivalent to `id=list(n=2)`.

The specifications that can be included in the `id` list are

`labels` By default, points are labeled with a data frame's row labels or by row numbers if there are no row labels. You can use whatever point labels you like by setting `labels` to a character vector with the same number of elements as there are data points.

`n` the number of points to label.

`cex` the relative size of the labels; `cex=0.5`, for example, produces labels half the default size.

`col` the color of the labels; if not set, the color is determined automatically. `location` where the labels are drawn. The default is `location="lr"` to draw labels to the left of points in the right half of the graph and to the right of points in the left half. Another option is `location="ab"`, to draw labels above points below the middle of the graph and below points above the middle. Finally, `location="avoid"` tries to avoid overplotting labels.

`method` Setting `method="identify"` enables interactive point identification, as with `identify()`. Several automatic methods are available for determining how interesting points are to be identified. For example, in `scatterplot()`, the default is `method="mahal"`, in which the Mahalanobis distance of each point to the centroid (point of averages) is computed, and the identified points are those with the largest Mahalanobis distances. For `residualPlot()`, the default is `method="r"`, selecting noteworthy points according to the

absolute values of their vertical coordinates (i.e., the residuals) and labeling the points with the largest values. There are many other options for the method argument; see help ("showLabels") for their description and the help pages for particular **car** graphics functions for how they use showLabels () .

You can also call the showLabels () function directly to add labels to most 2D plots created by the plot () or pairs () functions. See help ("showLabels") for usage details.

## 3.6 Scatterplot Smoothing

Like point identification, graphics functions in the **car** package, such as scatterplot (), scatterplotMatrix (), and crPlots (), handle scatterplot smoothing in a generalized manner. Scatterplot smoothers add a nonparametric regression line to a scatterplot to aid in interpretation.

There are three scatterplot smoothers provided by the **car** package: loessLine (), which is the default smoother and uses the loess () function (employing local polynomial regression: Cleveland, Grosse, & Shyu, 1992); gam-Line (), which uses the gam () function in the **mgcv** package (employing a smoothing spline: Wood, 2006); and quantregLine (), which uses the rqss () function in the **quantreg** package (employing nonparametric quantile regression: Koenker, 2018). All of these functions can compute variability smoothers as well as smoothers of the regression function.

Functions in the **car** package, such as scatterplot (), that use scatterplot smoothers have a smooth argument, which takes the values TRUE (the default), FALSE (to suppress the smoother), or a list of detailed specifications. The smooth list can include the element smoother, which specifies the smoother to be used (e.g., smoother=quantregLine). The other elements of the list vary by smoother. For example, for the default loessLine () smoother, specifying smooth=list (span=0.9, degree=1), or equivalently smooth=list (smoother=loessLine, span=0.9, degree=1), establishes a span of 0.9 (the default is span=2/3) and fits local linear regressions (the default, degree=2, fits local quadratic regressions). See help ("ScatterplotSmoothers") for details.

## 3.7 Complementary Reading and References

- Mosteller and Tukey (1977) and Tukey (1977) were very influential in convincing applied statisticians of the necessity of looking at their data through a variety of graphs.
- Most of the material in this chapter on examining data is covered in Fox (2016, chap. 3). Power transformations ([Section 3.4.2](#)) are discussed in Fox (2016, chap. 4).
- Plotting in general and scatterplot matrices in particular are also covered in Weisberg (2014, chap. 1). Much of the material in [Section 3.4](#) is covered in Weisberg (2014, chaps. 7–8), although some of the discussion in this section on transformations to multivariate normality conditioning on other variables is new material.
- Estimation of transformations in the Box-Cox family is described more completely in Fox (2016, Section 12.5.1) and Weisberg (2014, Section 8.1.3, Appendix A.12).
- There is a vast literature on density estimation, including Bowman and Azzalini (1997) and Silverman (1986).
- The loess smoother used in the scatterplot () function and elsewhere in the **car** package was introduced by Cleveland (1979) and is but one of many scatterplot smoothers, including regression and smoothing splines, kernel regression, and others (see, e.g., Fox, 2000). Jittering was apparently proposed by Chambers et al. (1983).
- The OpenGL graphics standard is discussed at <http://www.opengl.org>. R. D. Cook and Weisberg (1999) include an extended treatment of the role of three-dimensional plots in regression.
- The Wikipedia article on logarithms (at <http://en.wikipedia.org/wiki/Logarithms>) provides a nice introduction to the topic. Fox (2009b) describes logarithms, powers, and other mathematics for social statistics.

# 4 Fitting Linear Models

John Fox & Sanford Weisberg

*Linear models* are central to applied statistics. They are frequently used in research, and provide the basis for many other classes of statistical models, such as the generalized linear models discussed in [Chapter 6](#).

This chapter explains how to specify and fit linear models in R. Many of the topics taken up in the chapter apply generally to statistical-modeling functions in R.

- [Section 4.1](#) reviews the structure of the linear model and establishes basic notation. As usual in this book, we assume general familiarity with the statistical material.
- [Section 4.2](#) explains how to fit simple and multiple linear regression models in R using the `lm()` function.
- [Section 4.3](#) introduces *predictor effect plots* as a means of visualizing the fit of a linear model to data.
- [Section 4.4](#) shows how nonlinear relationships can be represented in a linear model by *polynomial* and *regression spline* terms.
- [Sections 4.5](#) and [4.6](#) explain, respectively, how to include categorical predictors (factors) and interactions in linear models fit by `lm()`.
- [Section 4.7](#) provides more detail on incorporating factors in linear and other statistical models in R.
- [Section 4.8](#) describes how `lm()` handles models in which some regressors are perfectly collinear.
- Finally, [Section 4.9](#) systematically describes the arguments of the `lm()` function.

## 4.1 The Linear Model

A statistical model is a collection of assumptions that has sufficient structure to support estimating interesting quantities, to use past data to predict future values, and to perform many other tasks. We assume general familiarity with the standard linear regression model but briefly summarize the essential characteristics of the model, many of which are assumptions made about consequences of the process generating the data. Our approach to linear regression has some nonstandard elements, and we will integrate discussion of these into the chapter.

We have a set  $\mathbf{u} = (u_1, \dots, u_m)$  of  $m$  *predictors* or *explanatory variables* and a *response variable*  $y$  observed on each of  $n$  units or cases.<sup>1</sup> In many data sets, there may be missing values.<sup>2</sup> The predictors can be numeric variables, such as

age or score on an exam; qualitative categorical variables, such as gender, country of origin, or treatment group; or ordinal categorical variables, such as a subject's assessment of an item on a 5-point scale from strongly disagree to strongly agree. The predictors are converted into *regressor variables*, or *regressors* for short, which are numeric variables that appear directly in the model.

[1](#) If you are unfamiliar with vector notation, simply think of  $\mathbf{u} = (u_1, \dots, u_m)$  as the collection of predictor variables and similarly for other vectors in this section.

[2](#) How missing data are represented in R is described in [Section 2.3.2](#). When there is more than a small proportion of missing data, simply ignoring missing values can be seriously problematic. One principled approach to missing data, multiple imputation, is described in an online appendix to the *R Companion*.

- A numeric predictor often corresponds to just one regressor given by the predictor itself, but it could require a transformation to another scale, such as logarithms, or generate several regressors, for example, if it is to be used in a polynomial of degree higher than 1 or in a regression spline (see [Section 4.4](#)).
- A qualitative predictor, or *factor*, with  $q$  distinct categories or *levels* typically requires  $q - 1$  regressors (see [Sections 4.5](#) and [4.7](#)).
- Additional regressor variables can be generated to represent *interactions*, which are functions of more than one predictor (see [Section 4.6](#)). Interactions allow the effect of a predictor on a response to depend on the values of other predictors.
- In all, the  $m$  predictors in  $\mathbf{u}$  generate  $k + 1$  regressors, which we generically name  $\mathbf{x} = (x_0, x_1, \dots, x_k)$ , where  $x_0 = 1$  represents the *regression constant* or *intercept* if, as is usually the case, the model includes an intercept. The regressors are determined by the predictors, so if we know all the predictors, we know all the regressors and vice versa.

The assumptions of the linear model are as follows:

- The response in a linear model is a numeric variable that is at least nominally continuous. Qualitative response variables and count response variables require other regression models, some of which are described in [Chapter 6](#).
- *Independence*: The observations of the variables for one case are independent of the observations for all other cases. If cases are dependent, linear mixed-effects models, discussed in [Chapter 7](#), may be more appropriate.<sup>3</sup>

- *Linearity*: The dependence of the response on the predictors is through the conditional expected value or *mean function*,

Other

(4.1)

$$E(y|\mathbf{x}) = E(y|x_1, \dots, x_k) = \beta_0 + \beta_1 x_1 + \dots + \beta_k x_k$$

The vertical bar “|” is read as *given*, and thus  $E(y|x_1, \dots, x_k)$  is the conditional expectation of the response  $y$  given fixed values of the regressors or, equivalently, given fixed values of the predictors. The quantity  $\eta(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \dots + \beta_k x_k$  on the right-hand side of Equation 4.1 is called the *linear predictor*, where  $\mathbf{x} = (x_1, \dots, x_k)$  is the vector of regressors. Equation 4.1 specifies a linear regression model because its right-hand side is a linear combination of the parameters, that is, the  $\beta$ s.

If the mean function is wrong, then the parameters, and any conclusions drawn from fitting the model to data, may be invalid. The **car** package contains many functions that can help you decide whether the assumption of linearity is reasonable in any given problem, an issue addressed in [Chapter 8](#).

- *Constant Conditional Variance*: The conditional variance of the response given the regressors (or, equivalently, the predictors) is constant,

Other

(4.2)

$$\text{Var}(y|x_1, \dots, x_k) = \sigma^2 > 0$$

Failure of this assumption to hold does not invalidate the least-squares estimates of the  $\beta$ s, but failure of the assumption may invalidate coefficient standard errors and their downstream consequences, namely tests and confidence statements. In [Sections 5.1.1–5.1.3](#), we present modifications to standard errors that are less affected than the standard approach by departures from the assumption of constant conditional variance, and in [Section 8.5](#) we introduce diagnostics for nonconstant conditional variance.

<sup>3</sup> Also see the online appendix to the *R Companion* on time-series regression. An alternative, common, and equivalent specification of the model in Equation 4.1 is as

Other

$$y = \beta_0 + \beta_1 x_1 + \dots + \beta_k x_k + \varepsilon$$

where the unobserved variable  $\varepsilon$  is called an *error*. The independence assumption is equivalent to  $\varepsilon_i$  and  $\varepsilon_{i'}$  independent of one another for cases  $i \neq i'$ ; the linearity assumption implies that  $E(\varepsilon|x_1, \dots, x_k) = 0$ ; and the constant

variance assumption implies that  $\text{Var}(\varepsilon|x_1, \dots, x_k) = \sigma^2$ .

Changing assumptions changes the model. For example, it is common to add a *normality assumption*,

Other

$$(\gamma|\mathbf{x}) \sim N(\beta_0 + \beta_1 x_1 + \dots + \beta_k x_k, \sigma^2)$$

or, equivalently,  $(\varepsilon|\mathbf{x}) \sim N(0, \sigma^2)$ , producing the *normal linear model*. The normal linear model provides more structure than is required for fitting linear models by least squares, although it furnishes a strong justification for doing so.

Another common extension to the linear model is to modify the constant variance assumption to

Other

$$\text{Var}(\gamma|\mathbf{x}) = \text{Var}(\varepsilon|\mathbf{x}) = \sigma^2/w$$

for known positive weights  $w$ , producing the *weighted linear model*. There are myriad other changes that might be made to the basic assumptions of the linear model, each possibly requiring a modification in methodology.

The basic R function for fitting linear regression models by *ordinary least squares* (OLS) or *weighted least squares* (WLS) is the lm () function, which is the primary focus of this chapter.

## 4.2 Linear Least-Squares Regression

### 4.2.1 Simple Linear Regression

The Davis data set in the **carData** package contains the measured and self-reported heights and weights of 200 men and women engaged in regular exercise. A few of the data values are missing, and consequently there are only 183 complete cases for the variables that are used in the analysis reported below.

We start by taking a quick look at the data:<sup>4</sup>

```

library("car")
summary(Davis)

   sex          weight         height        repwt
F:112    Min.   : 39.0    Min.   : 57    Min.   : 41.0
M: 88    1st Qu.: 55.0    1st Qu.:164   1st Qu.: 55.0
          Median : 63.0    Median :170   Median : 63.0
          Mean   : 65.8    Mean   :170   Mean   : 65.6
          3rd Qu.: 74.0    3rd Qu.:177   3rd Qu.: 73.5
          Max.   :166.0    Max.   :197   Max.   :124.0
                           NA's   :17

   reht
Min.   :148
1st Qu.:160
Median :168
Mean   :168
3rd Qu.:175
Max.   :200
NA's   :17

```

[4](#) Working in RStudio, we might instead, or in addition, use the `View()` command to see the full data set in the source-editor pane, as described in [Section 1.5](#).

The variables `weight` (measured weight) and `repwt` (reported weight) are in kilograms, and `height` (measured height) and `reht` (reported height) are in centimeters. One of the goals of the researcher who collected these data (Davis, 1990) was to determine whether the self-reports of height and weight are sufficiently accurate to replace the actual measurements, which suggests regressing each measurement on the corresponding self-report.<sup>5</sup>

[5](#) This prediction problem reverses the natural causal direction of the regression, which would be to regress the reports on the measurements.

We focus here on the regression of `weight` on `repwt`. This problem has response  $y = \text{weight}$  and one predictor, `repwt`, from which we obtain the regressor  $x_1 =$

`repwt`. The *simple linear regression model* is a special case of Equation 4.1 with  $k = 1$ . Simple linear regression is fit in R via OLS using the `lm()` function:

```
davis.mod <- lm (weight ~ repwt, data=Davis)
```

The basic form of the `lm()` command is

```
model <- lm (formula=model-formula, data=dataset)
```

The formula argument describes the response and the linear predictor, and is the only required argument. The data argument optionally supplies a data frame that includes the variables to be used in fitting the model.<sup>6</sup> The `lm()` function returns a linear-model object, which we can assign to an R variable, here `davis.mod`. The full set of arguments for `lm()` is described in [Section 4.9](#).

<sup>6</sup> If the data argument isn't specified, `lm()` looks for the variables in the formula in the workspace and then along the search path (see [Section 2.3.1](#)). This is generally inadvisable as there is no guarantee that two variables in the workspace belong to the same data set, and so we will always provide the data argument to statistical-modeling functions like `lm()`.

The general formula syntax was originally proposed by G. N. Wilkinson and Rogers (1973) specifically for use with linear models, and R isn't unique in employing a version of Wilkinson and Rogers's notation.<sup>7</sup> Formulas are used more generally in R, but their application is clearest for regression models with linear predictors, such as the linear regression models discussed in this chapter and the generalized linear models taken up in [Chapter 6](#).

<sup>7</sup> SAS, for example, also uses a version of the Wilkinson and Rogers notation for linear and related models, such as generalized linear models.

A model formula consists of three parts: the *left-hand side*, the  $\sim$  (*tilde*), and the *right-hand side*:

- The left-hand side of the model formula specifies the response variable; it is usually a variable name (`weight`, in the example) but may be an expression

that *evaluates* to the response (e.g., `sqrt(weight)`, `log(income)`, or `income/hours.worked`).

- The tilde is a separator.
- The right-hand side is the most complex part of the formula. It is a special expression, including the names of the predictors, that R evaluates to produce the regressors for the model. The arithmetic operators, `+`, `-`, `*`, `/`, and `^`, have special meaning on the right-hand side of a model formula, but they retain their ordinary meaning on the left-hand side of the formula.

R will use any unadorned numeric predictor on the right-hand side of the model formula as a regressor, as is desired here for simple regression.

The intercept  $\beta_0$  is included in the model without being specified directly. We can put the intercept in explicitly using `weight ~ 1 + repwt`, or suppress the intercept to force the regression through the origin using `weight ~ -1 + repwt` or `weight ~ repwt - 1`. More generally, a minus sign removes a term, here the intercept, from the linear predictor. Using 0 (zero) in a formula also suppresses the intercept: `weight ~ 0 + repwt`.

As subsequent examples illustrate, model formulas can be much more elaborate; formulas are described in detail in [Section 4.9.1](#).

The `lm()` function returns a *linear-model object* of class "lm", which in the example we save in `davis.mod`. We subsequently can call other functions with `davis.mod` as an argument to display aspects of the model or employ the model object to produce useful quantities like predictions. As with any R object, we can print `davis.mod` by typing its name at the R command prompt:

## **davis.mod**

Call:

```
lm(formula = weight ~ repwt, data = Davis)
```

Coefficients:

| (Intercept) | repwt |
|-------------|-------|
| 5.336       | 0.928 |

Printing a linear-model object simply shows the command that produced the model along with the estimated regression coefficients. A more comprehensive summary of the fit is obtained using R's standard `summary()` generic function. The `car` package includes a replacement for this function, called `S()`, that allows the user more control over what gets printed and adds additional functionality that will be described later in this chapter and in the next:

## **S(davis.mod)**

```
Call: lm(formula = weight ~ repwt, data = Davis)
```

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 5.3363   | 3.0369     | 1.76    | 0.081    |
| repwt       | 0.9278   | 0.0453     | 20.48   | <2e-16   |

Residual standard deviation: 8.42 on 181 degrees of freedom  
(17 observations deleted due to missingness)

Multiple R-squared: 0.699

F-statistic: 420 on 1 and 181 DF, p-value: <2e-16

| AIC    | BIC    |
|--------|--------|
| 1303.1 | 1312.7 |

The default output from `S()` is a subset of the information printed by `summary()`.

- A brief header is printed first, repeating the command that created the regression model.
- The part of the output marked Coefficients provides basic information about the estimated regression coefficients. The first column lists the names of the regressors fit by lm(). The intercept, if present, is named (Intercept). The column labeled Estimate provides the least-squares estimates of the regression coefficients.

The column marked Std. Error displays the standard errors of the estimated coefficients. The default standard errors are computed as  $\sigma$  times a function of the regressors, as presented in any regression text, but the S() function also allows you to use other methods for computing standard errors (see [Section 5.1.2](#)).

The column marked t value is the ratio of each estimate to its standard error and is a *Wald test* statistic for the null hypothesis that the corresponding population regression coefficient is equal to zero. If assumptions hold and the errors are normally distributed or the sample size is large enough, then these *t*-statistics computed with the default standard error estimates are distributed under the null hypothesis as *t* random variables with degrees of freedom (*df*) equal to the residual degrees of freedom under the model.

The column marked Pr (>|t|) is the two-sided *p*-value for the null hypothesis assuming that the *t*-distribution is appropriate. For example, the hypothesis that  $\beta_0 = 0$  versus  $\beta_0 \neq 0$ , with the value of  $\beta_1$  unspecified, has a *p*-value of about .08, providing weak evidence against the null hypothesis that  $\beta_0 = 0$ , if the assumptions of the model hold.<sup>8</sup>

- Below the coefficient table is additional summary information, including the residual standard deviation.<sup>9</sup> For the example, . This typical error in prediction is so large that, if correct, it is unlikely that the predictions of actual weight from reported weight would be of any practical value.

The residual *df* are  $n - 2$  for simple regression, here  $183 - 2 = 181$ , and we're alerted to the fact that 17 cases are removed because of missing data.

The Multiple R-squared,  $R^2 \approx 0.70$ , is the square of the correlation between the response and the fitted values and is interpretable as the proportion of variation of the response variable around its mean accounted for by the regression.

The reported  $F$ -statistic provides a likelihood-ratio test of the general null hypothesis that all the population regression coefficients in Equation 4.1, except for the intercept, are equal to zero, versus the alternative that at least one of the  $\beta_j$  is nonzero. If the errors are normal or  $n$  is large enough, then this test statistic has an  $F$ -distribution with the degrees of freedom shown. Because simple regression has only one parameter beyond the intercept, the  $F$ -test is equivalent to the  $t$ -test that  $\beta_1 = 0$ , with  $t_2 = F$ . In other models, such as the GLMs of [Chapter 6](#) or normal nonlinear models,<sup>10</sup> Wald tests and the generalization of the likelihood-ratio  $F$ -test may test the same hypotheses, but they need not provide the same inferences.

- The AIC and BIC values at the bottom of the output are, respectively, the *Akaike Information Criterion* and the *Bayesian Information Criterion*, statistics that are sometimes used for model selection.

[8](#) Typically, the null hypothesis that the population intercept  $\beta_0 = 0$  isn't interesting, but if in Davis's study actual weight can be predicted without bias from reported weight, we would expect  $\beta_0 = 0$ : See below.

[9](#) The output from the `summary()` command calls this the *residual standard error*, but we prefer to reserve the term *standard error* to refer to the estimated sampling standard deviation of a statistic. The quantity is the estimate of the error standard deviation  $\sigma$  in the linear regression model.

[10](#) Nonlinear regression is the subject of an online appendix to the *R Companion*.

The `brief()` function from the **car** package can also be used to obtain a terser summary of the model (try it!).

If individuals were unbiased reporters of their weight, then the regression intercept should be near zero and the slope near 1,  $E(y|x) = x$ . To assess this expectation, we can examine *confidence intervals* for the estimates using the **car** package function `Confint()`, which is a substitute for the standard R `confint()` function.<sup>11</sup>

## **Confint (davis.mod)**

|             | Estimate | 2.5 %    | 97.5 %  |
|-------------|----------|----------|---------|
| (Intercept) | 5.33626  | -0.65604 | 11.3286 |
| repwt       | 0.92784  | 0.83847  | 1.0172  |

[11](#) Unlike `confint ()`, `Confint ()` prints the coefficient estimates along with the confidence limits. Both are generic functions (see [Sections 1.7](#) and [10.9](#)) that are applicable to a wide variety of statistical models.

The values  $\beta_0 = 0$  and  $\beta_1 = 1$  are therefore *marginally* (i.e., individually) consistent with unbiased prediction of weight because these values are included in their respective confidence intervals, although the interval for the intercept is so wide that the coefficient estimates are unlikely to be of much value. The *separate* confidence intervals do not address the hypothesis that *simultaneously*  $\beta_0 = 0$  and  $\beta_1 = 1$  versus the alternative that either or both of the intercept and slope differ from these values.[12](#)

[12](#) We provide the relevant test in [Section 5.3.5](#), where we discuss general linear hypotheses.

We should have started (of course!) by plotting the data, and we now do so belatedly:

```
plot (weight ~ repwt, data=Davis)

abline (0, 1, lty="dashed", lwd=2)

abline (davis.mod, lwd=2)

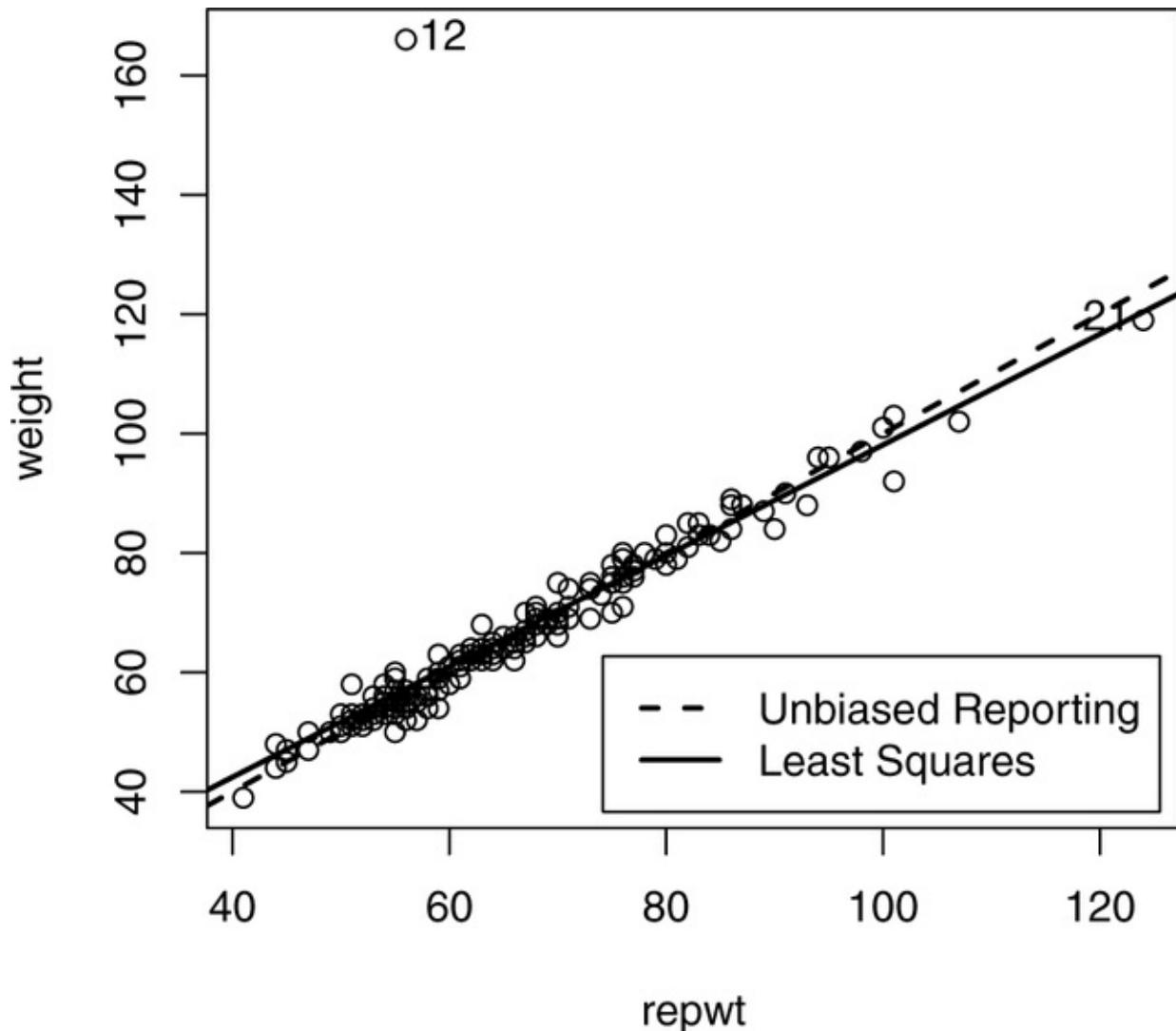
legend ("bottomright", c ("Unbiased Reporting", "Least Squares"), lty=c
("dashed", "solid"), lwd=2, inset=0.02)

with (Davis, showLabels (repwt, weight, n=2, method="mahal"))
```

[1] 12 21

The `plot()` function draws the basic scatterplot, to which we use `abline()` to add the line of unbiased reporting ( $y = x$ , with intercept zero and slope 1, the broken line) and the least-squares line (the solid line). The `legend()` command adds a legend at the lower right of the plot, inset from the corner by 2% of the plot's size. The `showLabels()` function from the `car` package (discussed in [Section 3.5](#)) identifies the two points with the largest Mahalanobis distances from the center of the data. The resulting graph, shown in [Figure 4.1](#), reveals an extreme outlier, Case 12; Case 21 has unusually large values of both measured and reported weight but is in line with the rest of the data.<sup>13</sup> It seems bizarre that an individual who weighs more than 160 kg would report her weight as less than 60 kg, but there is a simple explanation: On data entry, Subject 12's height in centimeters and weight in kilograms were inadvertently exchanged.

**Figure 4.1** Scatterplot of measured weight (weight) by reported weight (repwt) for Davis's data. The solid line is the least-squares linear regression line, and the broken line is the line of unbiased reporting  $y = x$ .



[13](#) Obsessive readers will notice that there are fewer than 183 points visible in [Figure 4.1](#). Both repwt and weight are given to the nearest kilogram, and consequently many points are overplotted; we could reveal the concealed points by jittering, as described in [Section 3.2.1](#).

The proper course of action would be to correct the data, but to extend the example, we instead use the update () function to refit the model removing the 12th case:

```

davis.mod.2 <- update(davis.mod, subset=-12)
S(davis.mod.2)

Call: lm(formula = weight ~ repwt, data = Davis, subset = -12)

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.7338    0.8148   3.36  0.00097
repwt       0.9584    0.0121  78.93 < 2e-16

Residual standard deviation: 2.25 on 180 degrees of freedom
(17 observations deleted due to missingness)
Multiple R-squared: 0.972
F-statistic: 6.23e+03 on 1 and 180 DF, p-value: <2e-16

AIC      BIC
816.27 825.88

```

The `update()` function can be used in many circumstances to create a new model object from an existing one by changing one or more arguments. In this case, setting `subset=-12` refits the model by omitting the 12th case.<sup>[14](#)</sup>

<sup>14</sup> See [Section 4.9.3](#) for more on the `subset` argument.

Extreme outliers such as this one have several deleterious effects on a fitted model, which will be explored more fully in [Chapter 8](#). Outliers can sometimes effectively determine the values of the estimated coefficients, and the  $F$ - and  $t$ -tests are not reliable when outliers are present. We use the `compareCoefs()` function from the `car` package to compare the estimated coefficients and their standard errors from the two fitted regressions, with and without Case 12, and `Confint()` to recompute the confidence intervals for the coefficients:

```
compareCoefs(davis.mod, davis.mod.2)
```

Calls:

```
1: lm(formula = weight ~ repwt, data = Davis)
2: lm(formula = weight ~ repwt, data = Davis, subset = -12)
```

|             | Model 1 | Model 2 |
|-------------|---------|---------|
| (Intercept) | 5.336   | 2.734   |
| SE          | 3.037   | 0.815   |
| repwt       | 0.9278  | 0.9584  |
| SE          | 0.0453  | 0.0121  |

```
Confint(davis.mod.2)
```

|             | Estimate | 2.5 %   | 97.5 %  |
|-------------|----------|---------|---------|
| (Intercept) | 2.73380  | 1.12602 | 4.34158 |
| repwt       | 0.95837  | 0.93441 | 0.98233 |

Both the intercept and the slope change, but not dramatically, when the outlier is removed. Case 12 is at a relatively *low-leverage point*, because its value for repwt is near the center of the distribution of the regressor (see [Section 8.3](#)). In contrast, there is a major change in the residual standard deviation, reduced from an unacceptable 8.4 kg to a possibly acceptable 2.2 kg. As a consequence, the coefficient standard errors are also much smaller when the outlier is removed, and the value of  $R^2$  is greatly increased. Because the coefficient standard errors are now smaller, even though the estimated intercept  $b_0$  is still close to zero and the estimated slope  $b_1$  is close to 1,  $\beta_0 = 0$  and  $\beta_1 = 1$  are both *outside* of the recomputed 95% confidence intervals for the coefficients.<sup>15</sup>

<sup>15</sup> In this book, we indicate an estimated regression coefficient by replacing a Greek letter with the corresponding Roman letter, as in  $b_1$  for  $\beta_1$ .

## 4.2.2 Multiple Linear Regression

*Multiple regression* extends simple regression to more than one predictor. To provide an illustration, we return to the Canadian occupational-prestige data introduced in [Section 3.1.1](#):

```
summary(Prestige)
```

| education     | income        | women         | prestige     |
|---------------|---------------|---------------|--------------|
| Min. : 6.38   | Min. : 611    | Min. : 0.00   | Min. :14.8   |
| 1st Qu.: 8.45 | 1st Qu.: 4106 | 1st Qu.: 3.59 | 1st Qu.:35.2 |
| Median :10.54 | Median : 5930 | Median :13.60 | Median :43.6 |
| Mean :10.74   | Mean : 6798   | Mean :28.98   | Mean :46.8   |
| 3rd Qu.:12.65 | 3rd Qu.: 8187 | 3rd Qu.:52.20 | 3rd Qu.:59.3 |
| Max. :15.97   | Max. :25879   | Max. :97.51   | Max. :87.2   |
| census        | type          |               |              |
| Min. :1113    | bc :44        |               |              |
| 1st Qu.:3120  | prof:31       |               |              |
| Median :5135  | wc :23        |               |              |
| Mean :5402    | NA's: 4       |               |              |
| 3rd Qu.:8312  |               |               |              |
| Max. :9517    |               |               |              |

Just as simple regression should start with a scatterplot of the response versus the predictor, multiple regression should start with an examination of appropriate graphs, such as a scatterplot matrix. In [Section 3.3.2](#), we constructed a scatterplot matrix for the predictors education, income, and women and the response variable prestige ([Figure 3.14](#), page 147); based on this graph, we suggested replacing income by its logarithm. The resulting scatterplot matrix ([Figure 3.17](#), page 159), in which little or no curvature is observed in any of the panels of the plot, suggests that this is a good place to start regression modeling.

We fit the linear model in Equation 4.1 (page 175) for the response variable  $y = \text{prestige}$ , and from the three predictors, we generate  $k = 3$  regressors,  $x_1 = \text{education}$ ,  $x_2 = \log_2(\text{income})$ , and  $x_3 = \text{women}$ . Two of the three predictors are directly represented in the model as regressors, and the third regressor is derived by log transformation of the remaining predictor, income. A fourth regressor, the *constant regressor*  $x_0 = 1$  associated with the intercept, is included automatically in the model unless (as we explained) it is suppressed by including either -1 or +0 in the formula. This one-to-one correspondence of regressors to predictors isn't general for linear models, as we'll see in subsequent examples.

Using the lm () function to fit the multiple-regression model is straightforward:

```



```

The only difference between fitting a simple and multiple linear regression in R is in the right-hand side of the model formula. In a multiple regression, there are several terms, separated by + signs. R finds the three predictors education, income, and women and derives the corresponding regressors education,  $\log_2$ (income), and women. The pluses are read as “and” in interpreting the right-hand side of the formula: “Regress prestige on education *and*  $\log_2(\text{income})$  *and* women.”

The regression coefficients in multiple regression are often called *partial slopes* and can only be interpreted in the context of the other predictors in the model. For example, the estimate  $b_1 \approx 3.7$  suggests that for any fixed values of income and women, the average increment in prestige for an additional year of education is 3.7 prestige units. Interpreting the estimated partial slope of a logged predictor is more complicated. Although the base of logarithms is unimportant for many summaries, base-2 logs can sometimes simplify interpretation of partial slopes, because increasing  $\log_2(x)$  by one unit corresponds to *doubling*  $x$ . Thus, doubling income, holding the other predictors constant, is associated on average with an increase of about 9.3 prestige units. The coefficients of education and

$\log_2(\text{income})$  are both substantial and are both associated with very small  $p$ -values. In contrast, the coefficient of women is small and has a large  $p$ -value.<sup>16</sup>

<sup>16</sup> In interpreting the size of a regression coefficient, we must take into account the units of measurement of the response variable and the predictor. Here, the predictor women is the percentage of occupational incumbents who are women, and thus a hypothetical occupation composed entirely of women would on average have only  $100 \times 0.0469 = 4.69$  more prestige points than an occupation with comparable education and income levels composed entirely of men.

The estimated model is summarized by an equation that relates the *fitted*, or perhaps the *predicted*, value of the response as a function of the predictors,

Other

$$(4.3) \quad \widehat{\text{prestige}} = -110.97 + 3.73 \text{ education} + 9.31 \log_2(\text{income}) + 0.0469 \text{ women}$$

The symbol represents the estimated average value of prestige for a given value of the regressors  $x_1$ ,  $x_2$ , and  $x_3$  (and hence of the predictors education, income, and women).

### 4.2.3 Standardized Regression Coefficients

In some disciplines, it's common to standardize the response variable and the predictors in a multiple linear regression to zero means and unit standard deviations. Then the slope coefficient for a standardized predictor can be interpreted as the average change in the response, in standard-deviation units, for a one-standard-deviation increase in the predictor, holding other predictors constant. The resulting *standardized regression coefficients* are sometimes used to compare the effects of different predictors—a dubious practice, in our view, as we'll explain presently.<sup>17</sup> It is also misleading to compare standardized regression coefficients across groups that differ in variability in the predictors.

<sup>17</sup> Additionally, it is nonsense to standardize dummy regressors or other contrasts representing a factor, to standardize interaction regressors, or to individually standardize polynomial regressors or regression-spline regressors, topics that we take up later in the chapter.

To clarify the limitations of standardized regression coefficients, we introduce the Transact data set in the **carData** package:

***brief(Transact)***

```
261 x 3 data.frame (256 rows omitted)
  t1    t2   time
  [i]  [i]   [i]
1     0 1166  2396
2     0 1656  2348
3     0  899  2403
...
260 370 2644  7930
261 825 4429 13610
```

This data set includes two predictors, t1 and t2, the numbers of transactions of two types performed by the  $n = 261$  branches of a large bank, and the response variable time, the total minutes of labor in each branch.

For brevity, we skip the crucial step of drawing graphs of the data (but invite the reader to do so) and begin instead by regressing time on t1 and t2:

```
S(trans.1 <- lm(time ~ t1 + t2, data=Transact))  
Call: lm(formula = time ~ t1 + t2, data = Transact)
```

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 144.3694 | 170.5441   | 0.85    | 0.4      |
| t1          | 5.4621   | 0.4333     | 12.61   | <2e-16   |
| t2          | 2.0345   | 0.0943     | 21.57   | <2e-16   |

Residual standard deviation: 1140 on 258 degrees of freedom

Multiple R-squared: 0.909

F-statistic: 1.29e+03 on 2 and 258 DF, p-value: <2e-16

| AIC    | BIC    |
|--------|--------|
| 4421.1 | 4435.3 |

The intercept of 144 minutes is the estimated fixed number of minutes that a branch requires for nontransaction business. Because both t1 and t2 count transactions, their coefficients are directly comparable and are measured in minutes per transaction. As it turns out, type-one transactions take much longer on average than type-two transactions: 5.5 minutes versus 2.0 minutes per transaction.

We next standardize the variables in the Transact data set, an operation easily performed by the base-R scale () function:

```
Transact.s <- scale(Transact)
round(var(Transact.s), digits=3)
```

|      | t1    | t2    | time  |
|------|-------|-------|-------|
| t1   | 1.000 | 0.772 | 0.863 |
| t2   | 0.772 | 1.000 | 0.924 |
| time | 0.863 | 0.924 | 1.000 |

```
round(colMeans(Transact.s), digits=10)
```

|  | t1 | t2 | time |
|--|----|----|------|
|  | 0  | 0  | 0    |

We confirm that the covariance matrix of the standardized variables, computed by the var () function, is a correlation matrix, with variances of one down the main diagonal, and that the means of the standardized variables are within rounding error of zero.

Finally, let's refit the regression to the standardized data:

```
trans.s <- update(trans.1, data=as.data.frame(Transact.s))
S(trans.s, brief=TRUE)
```

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 1.18e-16 | 1.87e-02   | 0.0     | 1        |
| t1          | 3.72e-01 | 2.95e-02   | 12.6    | <2e-16   |
| t2          | 6.37e-01 | 2.95e-02   | 21.6    | <2e-16   |

Residual standard deviation: 0.303 on 258 degrees of freedom

Multiple R-squared: 0.909

F-statistic: 1.29e+03 on 2 and 258 DF, p-value: <2e-16

| AIC    | BIC    |
|--------|--------|
| 121.94 | 136.20 |

To refit the model, we *coerce* `Transact.s` to a data frame, as required by `lm()`, because the `scale()` function returns a numeric matrix. The argument `brief=TRUE` to `S()` abbreviates the output.

When the predictors and response are standardized, the intercept is always within rounding error of zero, here (where the asterisk denotes a standardized regression coefficient); we could have suppressed the intercept by entering `-1` on the right-hand side of the model formula, producing the same standardized regression coefficients for `t1` and `t2` (try it!).<sup>18</sup> Scale-invariant statistics, such as the *t*-values for the coefficients, the overall *F*-test, and the *R*<sup>2</sup> for the model, are the same as for the unstandardized regression.

<sup>18</sup> Suppressing the intercept changes the standard errors of the standardized regression coefficients slightly as a consequence of estimating one fewer parameter, increasing the residual *df* by 1. The standard errors for the standardized coefficients are wrong in any event, because they fail to take into account sampling variability in the sample means and standard deviations, which are used to standardize the variables.

In the standardized regression, the coefficient for `t1`, , is *smaller* than the coefficient for `t2`, , while the sizes of the unstandardized regression coefficients are in the opposite order. Why does this happen? Because the variability in type-two transactions across branches is much greater than the variability of type-one transactions:

```
sapply(Transact, sd)
```

| t1     | t2      | time    |
|--------|---------|---------|
| 257.08 | 1180.73 | 3774.05 |

In the current problem, the two predictors have the same units and so we can straightforwardly compare their coefficients; it is therefore apparent that the comparison is distorted by standardizing the variables, as if we used an elastic ruler. Now ask yourself this question: If standardization distorts coefficient comparisons when the predictors have the *same units*, in what sense does standardization allow us to compare coefficients of predictors that are measured in *different units*, which is the dubious application to which standardized coefficients are often put?

## 4.3 Predictor Effect Plots

A major theme of the *R Companion* is the use of graphs to understand data, to help build statistical models, and to present the results of fitting regression models to data. In this section, we introduce *predictor effect plots*, graphs that can be used to visualize the role, or “effect,” of each of the predictors in a fitted regression model.<sup>19</sup> Predictor effects can substitute for, or supplement, summaries based on tables of coefficients, standard errors, and tests, and they avoid most of the complications of interpretation of coefficients.

<sup>19</sup> We use the word *effect* in *effect plot* not necessarily to make a causal claim but rather in the same sense as in the terms *main effect* and *interaction effect*.

We begin by applying predictor effect plots to the multiple linear regression model prestige.mod for prestige fit in the previous section. A complete picture of the regression surface generated by the fitted model would require drawing a four-dimensional graph, with one axis for each of the three predictors, education, income, and women, and a fourth axis for the fitted values given by Equation 4.3 (on page 184). Graphing in four dimensions requires using color, motion, or another “trick” to represent a fourth dimension in our three-dimensional world. Moreover, because the dimensionality of the regression surface grows with the number of predictors, a trick that might work in four dimensions would likely be useless with a larger number of predictors.

Predictor effect plots solve this high-dimensional graphics problem by looking at one or more two-dimensional plots for each of the predictors in the model. To illustrate, concentrate for the moment on the *focal predictor* education. The essential idea is to fix the values of all the other predictors in the model, substituting these fixed values into Equation 4.3. Our default strategy replaces each of the other predictors by its observed average value. Because the observed average income is approximately \$6,798 and the observed average percent women is approximately 28.98%, we substitute these values into Equation 4.3, obtaining

Other

(4.4)

$$\begin{aligned}
 \widehat{\text{prestige}} &= -110.97 + 3.73 \times \text{education} \\
 &\quad + 9.31 \times \log_2(6798) + 0.0469 \times 28.98 \\
 &= 8.98 + 3.73 \times \text{education}
 \end{aligned}$$

Fixing the values of the other predictors in this manner, the effect of education on the response is represented by a straight line with slope equal to the estimated partial regression coefficient of education. The slope would be the same for *any* choice of fixed values of the other predictors. In contrast, the intercept depends on the values of the other predictors, and so the predictor effect plot for education has shape, summarized by the slope, that does not depend on the other predictors, but location (i.e., height), summarized by the intercept, that does depend on the choice of values for the other predictors.

[Figure 4.2](#) shows all three predictor effect plots derived from `prestige.mod` by treating each of the predictors in turn as the focal predictor:

***library ("effects")***

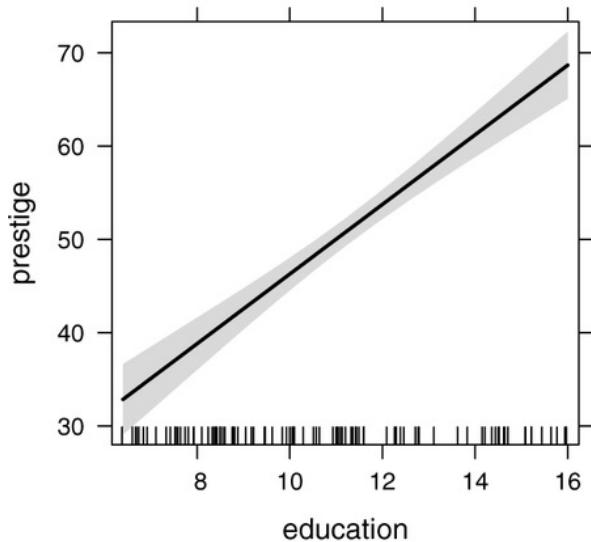
`lattice theme set by effectsTheme ()`

See `?effectsTheme` for details.

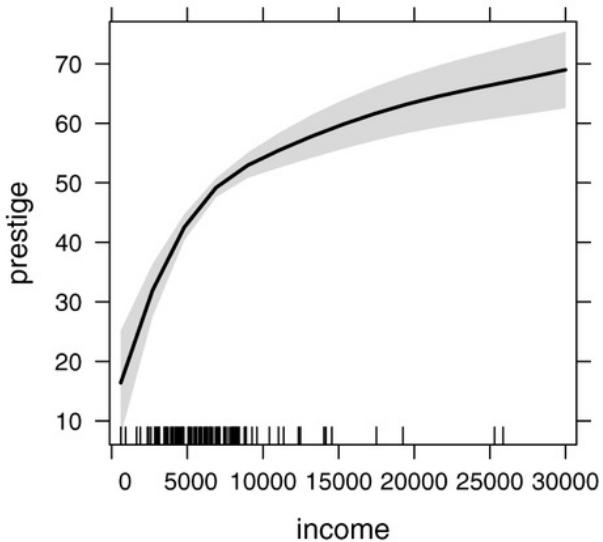
***plot (predictorEffects (prestige.mod))***

**Figure 4.2** Predictor effect plots for the three predictors in the model `prestige.mod`.

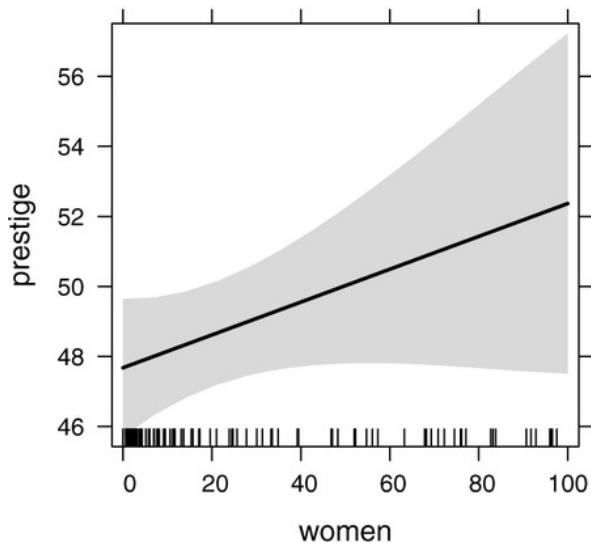
**education predictor effect plot**



**income predictor effect plot**



**women predictor effect plot**



This nested command first uses the `predictorEffects()` function in the **effects** package to compute the object to be graphed. The result returned by this function is then passed to the `plot()` generic function to draw the graph.

Both the `predictorEffects()` function and the `plot()` method for the objects it produces have additional optional arguments, but for many applications, the default values of these arguments produce a reasonable graph, as is the case here. We will illustrate some of these optional arguments in various examples in this chapter and in [Chapter 6](#) on generalized linear models. Additional information is

available from `help ("predictorEffects")` and `help ("plot.eff")` and in a vignette (see [Section 1.3.2](#)) in the **effects** package that provides an extensive gallery of variations on effect plots.

Predictor effect plots are drawn using the **lattice** package. Lattice graphs employ a *theme* that controls many graphical elements like line type, color, and thickness. The **effects** package customizes the standard lattice theme. If you want to change the theme, you should consult `help ("effectsTheme")`, as suggested by the **effects** package startup message.<sup>20</sup>

[20](#) Had we previously loaded the **lattice** package in the current session for some other purpose, then loading the **effects** package would not change the lattice theme, and the package startup message consequently would be slightly different.

The first of the three predictor effect plots for `prestige.mod` is for education, with the line shown on the plot defined by Equation 4.4. The shaded area represents pointwise 95% confidence intervals, called a *pointwise confidence envelope*, about the fitted line, without correction for simultaneous statistical inference.<sup>21</sup> The short vertical lines at the bottom of the graph are a *rug-plot* (sometimes termed a *one-dimensional scatterplot*), showing the values of education in the data. We see that the average value of prestige increases linearly as education increases; for example, occupations with 14 years of education on average have about 7.5 more prestige points than occupations with 12 years of education. This statement holds regardless of the values of the other predictors.

[21](#) Setting the argument `confint=list (type="Scheffe")` to `plot ()` instead displays a wider Scheffé-style confidence envelope (Scheffé, 1959) around the fitted effect, which is a conservative correction for simultaneous inference.

The second panel in [Figure 4.2](#) is the predictor effect plot for income. This plot is curved rather than straight because the horizontal axis is for the predictor income rather than for the regressor  $\log_2$  (income). We see from the rug-plot that most of the occupations have average incomes less than \$10,000 (recall that the data are for 1970) and that the effect of increasing income on prestige is considerably smaller at larger incomes, because the fitted line on the plot flattens out for larger values of income. Whereas the value of the regression coefficient for  $\log$  (income) depends on the base used for logarithms (we used logs to the base 2), the predictor effect plot is the same for any choice of base.

The third panel, for women, has a very large confidence band around the fitted line, suggesting that a horizontal line, equivalent to no effect of women on prestige controlling for the other predictors, is plausible given the variability in the estimated model, thus supporting the *t*-test for the coefficient of women discussed in the preceding section.

## 4.4 Polynomial Regression and Regression Splines

More complex nonlinear functions of a numeric predictor may require more than one parameter and consequently more than one regressor. In this section, we discuss the implementation in R of two such cases: *polynomial regression* and *regression splines*.

### 4.4.1 Polynomial Regression

In polynomial regression, the regressors associated with a predictor  $u_j$  are successive powers of the predictor: , where  $p$  is the *degree* of the polynomial. Thus, the *partial regression function* for the predictor  $u_j$  is . We assume that an intercept is in the model unless it is explicitly excluded.

To illustrate polynomial regression, we'll use the SLID data set from the **carData** package, which contains data for the province of Ontario from the 1994 wave of the Survey of Labour and Income Dynamics, a panel study of the Canadian labor force conducted by Statistics Canada. The data set contains several variables, including wages, the composite hourly wage rate of each respondent, in dollars; education, in years; age, also in years; sex, "Female" or "Male"; and language, "English", "French", or "Other":

**summary(SLID)**

| wages        | education    | age        | sex         |
|--------------|--------------|------------|-------------|
| Min. : 2.3   | Min. : 0.0   | Min. :16   | Female:3880 |
| 1st Qu.: 9.2 | 1st Qu.:10.3 | 1st Qu.:30 | Male : 3545 |

```

Median :14.1   Median :12.1   Median :41
Mean   :15.6   Mean   :12.5   Mean   :44
3rd Qu.:19.8   3rd Qu.:14.5   3rd Qu.:57
Max.   :49.9   Max.   :20.0   Max.   :95
NA's    :3278   NA's    :249

language
English:5716
French : 497
Other   :1091
NA's    : 121

```

### ***nrow(SLID)***

```
[1] 7425
```

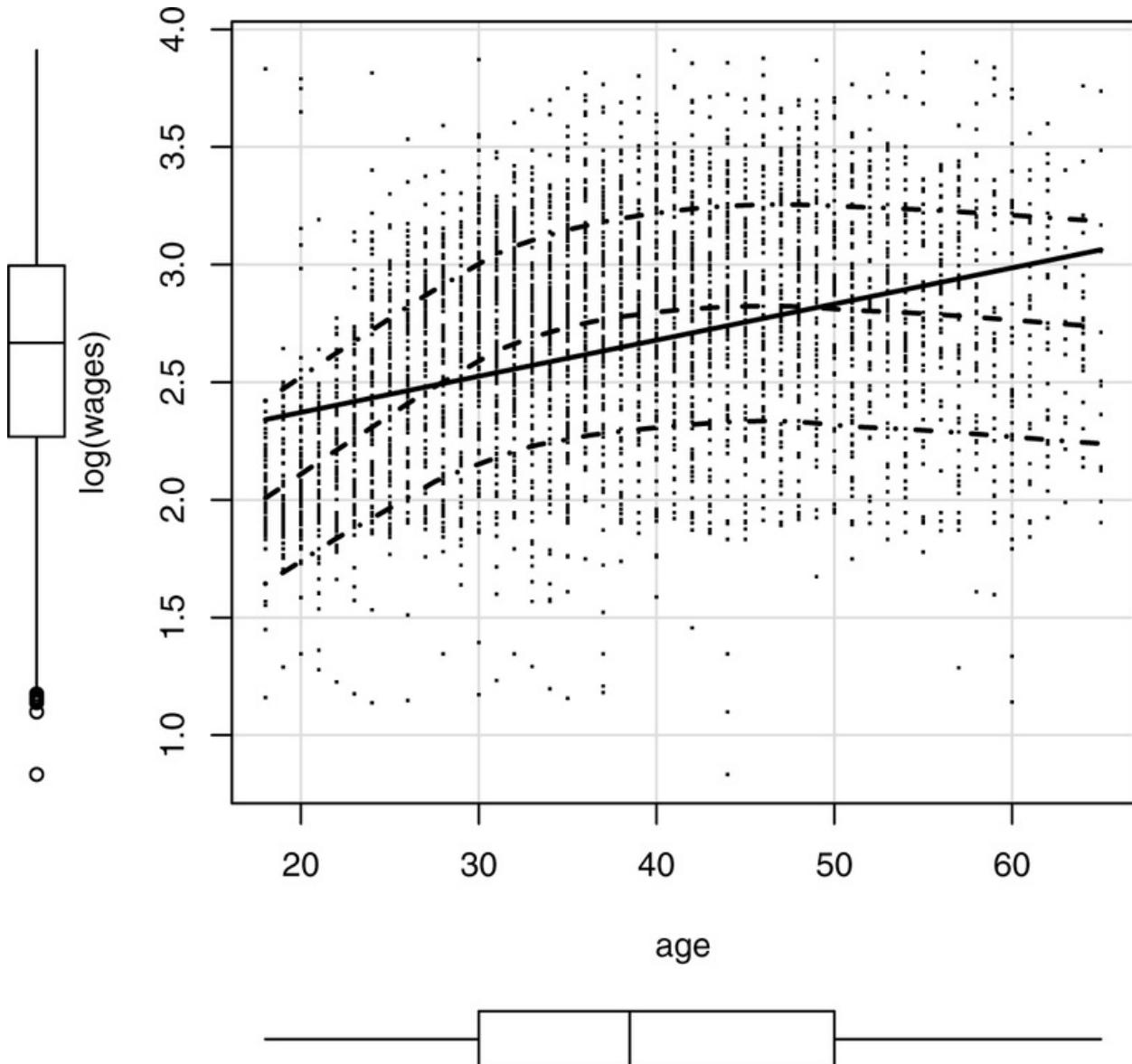
The 3,278 individuals coded NA for wages had no employment income, and a few other values are missing for some respondents. By default, lm () and the graphics functions that we use in this section exclude cases with missing data for any of the variables appearing in a command.<sup>22</sup> Our analysis here is therefore only for respondents with employment income.

<sup>22</sup> For example, the points shown in a scatterplot include only cases for which *both* variables in the graph are observed.

Because initial examination of the data showed that wages are, unsurprisingly, positively skewed, we use log (wages) to plot the data, displaying the relationship between wages and age (in [Figure 4.3](#)):

```
scatterplot (log (wages) ~ age, data=SLID, subset = age >= 18 & age <= 65, pch=".")
```

**Figure 4.3** Scatterplot of log (wages) by age for the SLID data set.



The subset argument restricts the plot to ages between 18 and 65. The argument `pch="."` (for “plotting character”) specifies dots, rather than circles, for the plotted points, which makes the points less prominent in this moderately large data set. A straight line clearly distorts the relationship between  $\log(\text{wages})$  and age in the scatterplot, as the smoother suggests increasing  $\log(\text{wages})$  up to age of about 40, and then steady, or slightly decreasing, wages beyond that point. A simple curve that may match this behavior is a *quadratic polynomial*, that is, a polynomial of degree 2.

The quadratic model can be fit with `lm()` in at least three equivalent ways:

```

brief(mod.quad.1 <- lm(log(wages) ~ poly(age, 2, raw=TRUE),
           data=SLID, subset = age >= 18 & age <= 65))

(Intercept) poly(age, 2, raw = TRUE)1
Estimate      0.6515                  0.09486
Std. Error    0.0697                  0.00375

poly(age, 2, raw = TRUE)2
Estimate      -1.02e-03
Std. Error     4.73e-05

Residual SD = 0.436 on 4010 df, R-squared = 0.222
brief(mod.quad.2 <- update(mod.quad.1, ~ age + I(age^2)))

(Intercept)      age   I(age^2)
Estimate       0.6515  0.09486 -1.02e-03
Std. Error     0.0697  0.00375  4.73e-05

Residual SD = 0.436 on 4010 df, R-squared = 0.222

brief(mod.quad.3 <- update(mod.quad.1, ~ poly(age, 2)))

(Intercept) poly(age, 2)1 poly(age, 2)2
Estimate      2.5388      -3.18      -29.11
Std. Error     0.0122       1.53       1.35

Residual SD = 0.436 on 4010 df, R-squared = 0.222

```

The first, and our preferred, method uses R's `poly()` function to generate the regressors `age` and `age2`, although the names assigned to these regressors are a little more verbose than this. We prefer this approach for two reasons: (1) The coefficients for the regressors `age` and `age2` have reasonably simple interpretations, and (2) functions such as `predictorEffects()` and `Anova()` (see [Sections 4.3](#) and [5.3.4](#)) understand that the two regressors belong to a single *term* in the model—a quadratic term in the predictor `age`. We use the `brief()` function in the `car` package for a compact summary of the fitted model, with more information than is provided by the `print()` method for "lm" objects but less than is provided by `S()`.

The second specification, mod.quad.2, uses the update () function to change the right-hand side of the model formula, entering regressors separately for the linear and quadratic components. We can't include age<sup>2</sup> directly because the <sup>^</sup> operator has special meaning on the right-hand side of a model formula (see [Section 4.9.1](#)), and so we *protect* the expression age<sup>2</sup> within a call to the I () (“Identity” or “Inhibit”) function, which returns its argument unaltered. The resulting model formula is based on *regressors* rather than on *predictors*, but the fitted regression equation is identical to mod.quad.1. Polynomials fit in this manner will not be interpreted correctly by functions that are based on terms in a linear model; for example, Anova () will treat age and I (age<sup>2</sup>) as separate terms in the model rather than as a single term comprising two regressors derived from the same predictor.

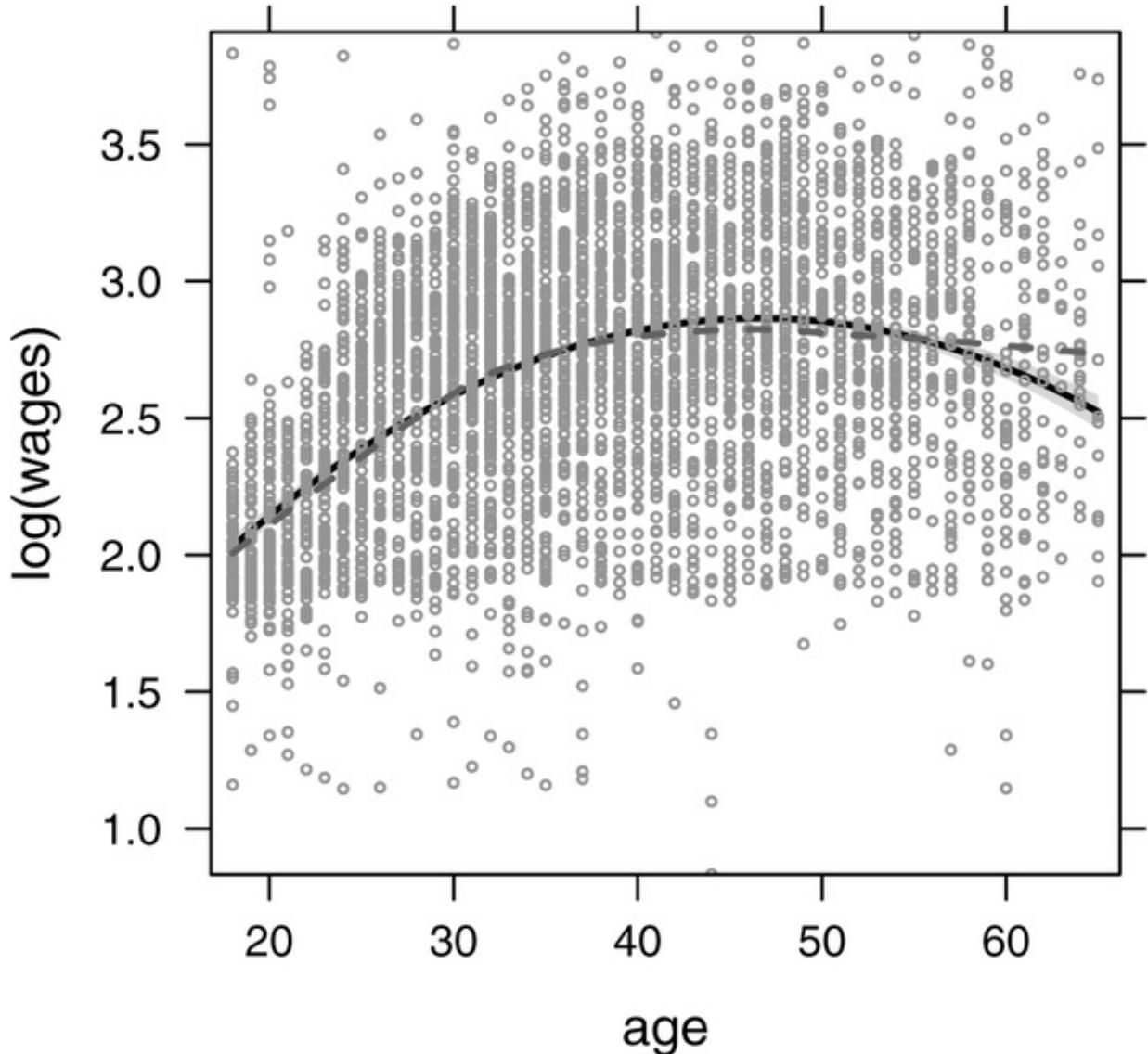
The third choice is to use the poly () function but without the argument raw= TRUE. The resulting regressors are for *orthogonal polynomials*, in which the linear and quadratic regressors are transformed to be uncorrelated and have unit length. The fitted model, mod.quad.3, has different coefficients than mod.quad.1 and mod.quad.2 but produces the same fitted values as the first two models and consequently the same  $R^2$  and residual SD. Both the lm () function and the glm () function for generalized linear models ([Chapter 6](#)) effectively orthogonalize regressors in the process of computation, and so there is no real advantage to employing orthogonal polynomial regressors directly in the model specification. We recommend against using this option because the resulting coefficients are generally more difficult to interpret and because term-aware functions like Anova () and predictorEffects () will produce the same results for raw and orthogonal polynomials.

The predictor effect plot for age in [Figure 4.4](#) is produced by the command

```
plot (predictorEffects (mod.quad.1, residuals=TRUE),
      partial.residuals=list (cex=0.35, col=gray (0.5), lty=2))
```

**Figure 4.4** Predictor effect plot for age in the quadratic regression fit to the SLID data. The solid line represents the fitted quadratic regression model with a gray 95% pointwise confidence envelope around the fit; the broken line is for a loess smooth of the plotted points.

## age predictor effect plot



Because age is the only predictor in the model, there are no other predictors to average over, and so the predictor effect plot merely graphs the fitted regression function. The very narrow, barely visible, confidence band suggests that the relationship is estimated very precisely. The argument `residuals=TRUE` to `predictorEffects()` displays the original data in the plot in problems like this one with only one predictor. The `partial.residuals` argument to the `plot()` function allows us to control the characteristics of the plotted points: We use `cex=0.35` to change the size of the plotted points, making them 35% as large as their default size; change the color of the points to gray (0.5), using the `gray()` function to

generate a gray value halfway between zero (black) and 1 (white); and set `lty=2` to display the smooth as a broken line.<sup>23</sup> Both `predictorEffects()` and its `plot()` method have a variety of optional arguments that you can use to customize predictor effect plots: See `help("predictorEffects")` and `help("plot.eff")`.

<sup>23</sup> See [Section 9.1.2](#) for a general discussion of graphics parameters in R, [Section 9.1.4](#) for an explanation of color selection, and [Section 8.4.2](#) for information about partial residuals.

[Figure 4.4](#) includes two lines: the quadratic curve, with confidence band, corresponding to the fitted model, and a smoother corresponding to a nonparametric regression fitted to the points in the plot. If the quadratic model were correct, then these two lines should be similar. There appears to be some minor disagreement at higher ages, say above age 60, suggesting that the quadratic model may overstate the decline in wages after age 60.

## 4.4.2 Regression Splines\*

Quadratic polynomials are frequently useful in regression modeling, and cubic or third-degree polynomials are occasionally useful, but higher-order polynomials are rarely used. Polynomial regressions are inherently *nonlocal*, with data in one region of the predictor space potentially influencing the fit in other regions. High-order polynomials can therefore take on wild values in regions where data are absent or sparse to produce artificially close fits in regions where data are plentiful.

In contrast to polynomials, *regression splines* produce nonlinear fits that are sensitive to *local* characteristics of the data, making only weak assumptions about the nature of the partial relationship between  $y$  and the numeric predictor  $u^j$ , essentially only an assumption of smoothness. Regression splines are in this respect similar to *nonparametric regression* methods such as loess, but regression splines are fully parametric and therefore can be used as components of a linear model fit by `lm()`. Regression splines and nonparametric regression typically produce very similar results.<sup>24</sup>

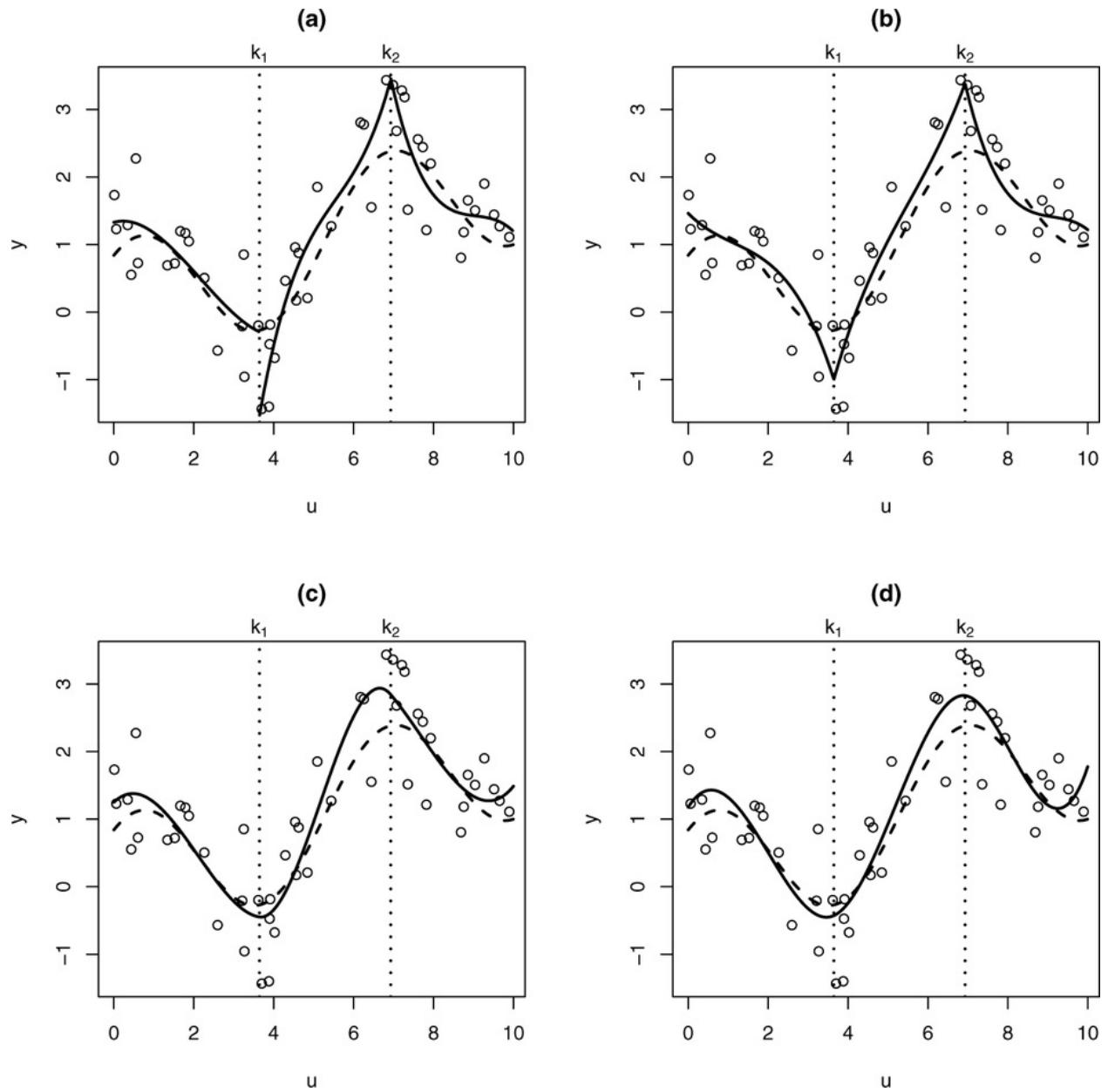
<sup>24</sup> See the appendix to the *R Companion* on nonparametric regression for more information about fitting nonparametric regression models in R.

*Splines* are piecewise polynomial functions, almost always cubic functions, that join at points called *knots*. For the numeric predictor  $u_j$ , the knots are placed along the range of  $u_j$ , with  $p$  knots partitioning the range of  $u_j$  into  $p + 1$  *bins*: values of  $u_j$  to the left of the first knot, values between the first and second knots, ..., values to the right of the  $p$ th knot. The knots are typically placed at regularly spaced quantiles of the distribution of  $u_j$ . For example, if  $p = 1$ , the single knot is at the median of  $u_j$ ; if  $p = 3$ , the knots are placed at the quartiles. There are also two *boundary knots* at or near the extremes of  $u_j$ .

Fitting an unconstrained cubic regression within each bin would require estimating four regression parameters in each bin, for a total of  $4(p + 1)$  parameters. These separate regressions would not be continuous at the knots, leading to undesirable jumps in the fitted curve at the knots. Regression splines are constrained not only to be continuous at the knots but also to have equal first (slope) and second (curvature) derivatives on both sides of each knot. The additional constraints substantially reduce the number of parameters required for the regression spline.

How regression splines work is explained graphically in [Figure 4.5](#), for which  $n = 50$  observations were generated randomly according to the model  $E(y) = \sin(u + 1) + u/5$ , with  $(y|u) \sim N(0, 0.5^2)$ , and with  $u$  values measured in radians and sampled uniformly on the interval  $[0, 10]$ . Knots are placed at the  $1/3$  and  $2/3$  quantiles of  $u$ , producing three bins. The broken line in each panel represents the population regression function that we want to estimate, and the solid line represents the fitted regression. The constraints on the cubic regression in each third of the data increase from panel (a) to panel (d), with panel (d) representing a cubic regression spline.

**Figure 4.5** How regression splines work: (a) piecewise cubics fit independently in each bin, with knots at  $k_1$  and  $k_2$ ; (b) cubic fits constrained to be continuous at the knots; (c) cubic fits that are continuous at the knots with equal slopes at each side; (d) cubic regression spline, which is continuous at the knots with equal slopes and curvature at each side. The “true” population regression curve is indicated by the broken line, and the “data” are artificially generated.



*Source:* Adapted from Fox (2016, Figure 17.5).

*B-splines* implement regression splines by selecting a computationally advantageous set of regressors called the *B-spline basis*. Assuming an intercept in the model, the cubic B-spline basis for the predictor  $x_j$  requires  $p + 4$  regressors rather than the  $4(p + 1)$  parameters for unconstrained fitting. *Natural splines* incorporate the two additional constraints that the regression is linear beyond the boundary knots and thus require two fewer regressors,  $p + 2$ .

Regression-spline terms can be conveniently incorporated in a model fit by lm () using the bs () function for B-splines or the ns () function for natural splines. The bs () and ns () functions are in the **splines** package, which is a standard part of the R distribution; the package isn't loaded automatically at the start of an R session and must therefore be loaded explicitly by the library ("splines") command. Both bs () and ns () have arguments for specifying either degrees of freedom (df), in which case the knots are placed at corresponding quantiles or, alternatively, the location of the knots (knots); bs () also permits the specification of the degree of the spline polynomials (with default degree=3), while ns () always uses a cubic spline.

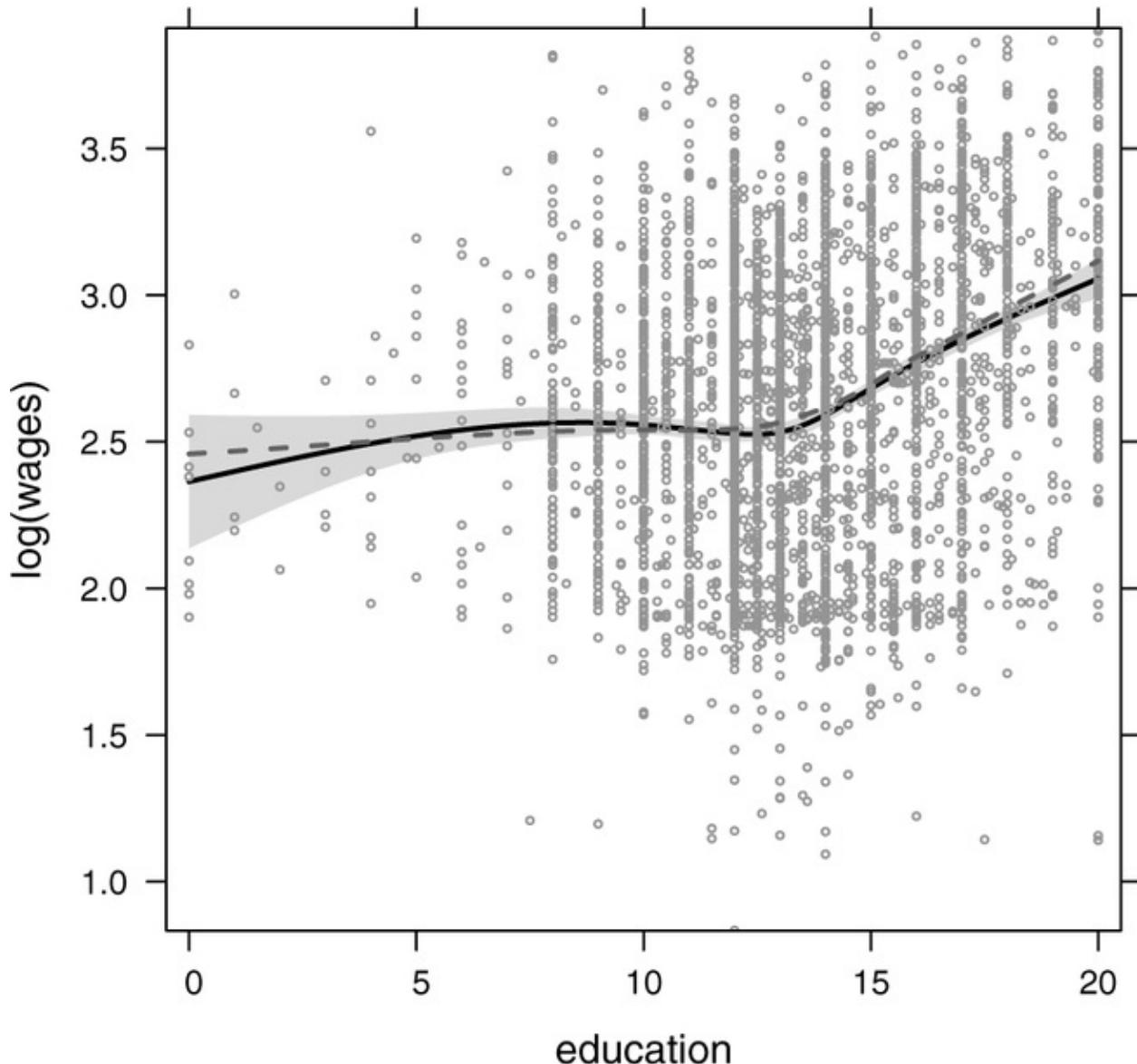
To illustrate, we examine the regression of wages on years of education in the SLID data set, representing the predictor education using a natural spline with five degrees of freedom, which, experience shows, is sufficiently flexible to capture even relatively complex nonlinear relationships:

```
library ("splines")
mod.spline <- update (mod.quad.1, ~ ns (education, df=5))
plot (predictorEffects (mod.spline, residuals=TRUE),
partial.residuals=list (cex=0.35, col=gray (0.5), lty=2))
```

We don't print the estimated regression coefficients because they have no straightforward interpretation. This example has only one predictor, and so the predictor effect plot in [Figure 4.6](#) displays the fitted regression function directly; as before, we include the data in the graph via the argument residuals=TRUE. Both the spline and nonparametric fit suggest relatively little effect of education on log (wages) until education exceeds 12 years, and from that point, log (wages) increases approximately linearly with education.

**Figure 4.6** Predictor effect plot for education in the regression of log (wages) on education, using a 5-*df* natural regression spline for education. The shaded region corresponds to the pointwise 95% confidence region around the solid black line representing the fitted model. The broken line on the plot is a loess smooth fit to the points.

## education predictor effect plot



## 4.5 Factors in Linear Models

The values of qualitative variables are category names rather than measurements. Examples of qualitative variables are gender (male, female, other), treatment in a clinical trial (drug, placebo), country of origin (Canada, Mexico, United States, ...), and job title (accountant, IT guru,...). Qualitative variables can have as few as two categories or a large number of categories. An *ordinal* categorical variable has categories that have a natural ordering, such as

age class (18–25, 26–35, ..., 56–65), the highest educational qualification attained by an individual (less than high school, high school diploma, ..., doctorate), or response on a 5-point scale (strongly disagree, disagree, neutral, agree, strongly agree).

We (and R) call categorical variables *factors* and their categories *levels*. In some statistical packages, such as SAS, factors are called *class variables*. Regardless of what they are called, factors are very common, and statistical software should make some provision for including them in regression models. When you create a factor in R, you can arrange its levels in any order you like, and you can establish arbitrary labels for the levels. You can also directly convert a numeric variable with a few distinct values to a factor, or convert a continuous numeric variable to a factor by binning its values ([Section 2.3.3](#)).<sup>25</sup>

[25](#) It is also possible to represent categorical predictors in R as character data (or, for a dichotomous categorical variable, as logical data) and to use such variables as predictors in linear and other statistical models in much the same manner as R factors. Factors in R have several advantages. First, representing a categorical variable as a factor allows you to order the categories of the variable in a natural manner, not necessarily alphabetically. Second, the values of a factor are stored internally as integers, which can be more efficient than storing the character-string category label for each case if the labels are long and the number of cases is large. Third, factors keep track of the set of levels associated with a categorical variable, which is sometimes important when a level is unexpectedly empty. On the other hand, there's a computational overhead incurred by converting character data into a factor when the data are read.

When you read a data file into R with `read.table()` or `read.csv()`, columns of character data are converted into factors by default (see [Section 2.1.3](#)). Similarly, data sets provided by R packages may include factors. An example is the variable type (type of occupation) in the Prestige data frame:

### **Prestige\$type**

```
[1] prof  
[12] prof  
[23] prof prof prof prof prof bc prof prof wc prof wc  
[34] <NA> wc wc wc wc wc wc wc <NA> bc wc  
[45] wc wc bc bc bc bc bc <NA> bc bc bc  
[56] wc wc bc bc bc bc bc bc bc bc bc  
[67] <NA> bc  
[78] bc  
[89] bc bc bc bc bc bc bc prof bc bc bc  
[100] bc bc bc  
Levels: bc prof wc
```

### **class (Prestige\$type)**

```
[1] "factor"
```

Like all R objects, Prestige\$type has a class, and, naturally, the class of a factor is "factor".

The three levels of the factor type represent blue-collar ("bc"), professional and managerial ("prof"), and white-collar ("wc") occupations. The missing-value symbol, NA, is not counted as a level of the factor.<sup>26</sup> Levels are ordered alphabetically by default, but you can produce any order you wish by calling the factor () function with the argument levels, as in

```
Prestige$type <- factor (Prestige$type, levels=c ("bc", "wc", "prof"))
```

### **levels (Prestige\$type)**

```
[1] "bc" "wc" "prof"
```

<sup>26</sup> In some instances, you may wish to include missing values as a level of a factor. This can be accomplished, for example, for Prestige\$type by the obscure command Prestige\$type1 <- factor (Prestige\$type, exclude=NULL).

The rearranged levels are in a more natural order, roughly corresponding to the status of the three occupational types.

A numeric or character vector can be converted or *coerced* into a factor. Suppose, for example, that x is a numeric vector representing the number of hours allowed for subjects to complete a task:

```
x <- c (2, 2, 4, 4, 8, 8, 8)
```

The factor () function can be used to create a factor from x:

```
(fact.x <- factor (x,  
labels=c (paste (c (2, 4, 8), "hours", sep="-"))))
```

```
[1] 2-hours 2-hours 4-hours 4-hours 4-hours 8-hours  
[8] 8-hours
```

Levels: 2-hours 4-hours 8-hours

The argument labels allows you to assign names to the levels of the factor in place of the default "2", "4", "8"; in this case, we use the paste () function to create the labels "2-hours", "4-hours", and "8-hours". We can also use the *coercion function* as.factor () but with less control over the result:

```
as.factor (x)
```

```
[1] 2 2 4 4 4 8 8 8
```

Levels: 2 4 8

Similarly, a factor can be coerced to either a character vector or a numeric vector. The as.numeric () function replaces each level of the factor by its level number, producing a "numeric" result,

```
as.numeric (fact.x)
```

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

If you want to recover the original numeric vector, you could use

```
c (2, 4, 8)[as.numeric (fact.x)]
```

```
[1] 2 2 4 4 4 8 8 8
```

Finally,

```
as.character (fact.x)
```

```
[1] "2-hours" "2-hours" "4-hours" "4-hours" "4-hours" "8-hours"  
[7] "8-hours" "8-hours"
```

converts the factor to a character vector.

When a factor is included in a model formula, R automatically creates regressors, called *contrasts*, to represent the levels of the factor. There are many ways to represent a factor with contrasts. The default in R is very simple: For a factor with  $q$  distinct levels,  $q - 1$  regressors are created, each of which is *dummy-coded*, with values consisting only of zeros and ones. For example, suppose that we have the factor z with four levels:

```
(z <- factor (rep (c ("a", "b", "c", "d"), c (3, 2, 4, 1))))
```

```
[1] a a a b b c c c c d
```

Levels: a b c d

We can see the dummy variables that are created by calling the `model.matrix()` function, specifying a *one-sided model formula* with z on the right-hand side:

```

model.matrix(~ z)

(Intercept) zb  zc  zd
1             1  0  0  0
2             1  0  0  0
3             1  0  0  0
4             1  1  0  0
5             1  1  0  0
6             1  0  1  0
7             1  0  1  0
8             1  0  1  0
9             1  0  1  0
10            1  0  0  1

attr("assign")
[1] 0 1 1 1

attr("contrasts")
attr("contrasts")$z
[1] "contr.treatment"

```

The first column of the *model matrix*, the matrix of regressors, is a column of ones, representing the intercept. The remaining three columns are the dummy regressors constructed from the factor *z*.<sup>27</sup> The contrasts () function more compactly reveals the contrast coding for the factor *z* but without showing the values of the contrasts for each case:

## ***contrasts(z)***

|   |   |   |
|---|---|---|
| b | c | d |
| a | 0 | 0 |
| b | 1 | 0 |
| c | 0 | 1 |
| d | 0 | 1 |

[27](#) The `model.matrix()` function also attaches the *attributes* "assign" and "contrasts" to the model matrix that it returns; you can safely disregard this information, which is useful for R programs that need to know the structure of the model matrix.

The rows of this matrix correspond to the levels of the factor, so there are four rows. The columns of the matrix correspond to the dummy regressors that are created. Thus, when `z = "a"`, all the dummy regressors are equal to zero; when `z = "b"`, the dummy regressor `zb = 1`; when `z = "c"`, `zc = 1`; and, finally, when `z = "d"`, `zd = 1`. Each level of `z` is therefore uniquely identified by a combination of the dummy regressors, with the first level, `z = "a"`, selected as the *baseline level* —or *reference level* —for the set of dummy regressors. You can change the baseline level using the `relevel()` function; see help ("relevel").

R provides a very general mechanism for changing the contrasts used for factors, but because the results of an analysis generally don't depend fundamentally on the choice of contrasts to represent a factor, most users will have little reason to modify the default contrasts. [Section 4.7](#) provides an extended discussion.

### **4.5.1 A Linear Model With One Factor: One-Way Analysis of Variance**

The simplest linear model with factors, known as *one-way analysis of variance* (abbreviated *one-way ANOVA*), has one factor and no numeric predictors. For example, Baumann and Jones (as reported in Moore & McCabe, 1993) conducted an experiment in which 66 children were assigned at random to one of three experimental groups. The groups represent different methods of teaching reading: a standard method called "Basal" and two new methods called "DTRA"

and "Strat". The researchers conducted two pretests and three posttests of reading comprehension. We focus here on the third posttest. The data for the study are in the data frame Baumann in the **carData** package:

```
summary(Baumann[, c(1, 6)])
```

| group    | post.test.3 |
|----------|-------------|
| Basal:22 | Min. :30    |
| DRTA :22 | 1st Qu.:40  |
| Strat:22 | Median :45  |
|          | Mean :44    |
|          | 3rd Qu.:49  |
|          | Max. :57    |

The researchers were interested in whether the new methods produce better results than the standard method and whether the new methods differ in their effectiveness. The experimental design has 22 subjects in each group:

```
xtabs(~ group, data=Baumann)
```

| group |      |       |
|-------|------|-------|
| Basal | DRTA | Strat |
| 22    | 22   | 22    |

When `xtabs()` ("cross-tabs") is used with a one-sided formula, it counts the number of cases in the data set at each level or combination of levels of the right-hand-side variables, in this case just the factor group.

The `Tapply()` function in the **car** package can be used to compute the means and SDs of the posttest scores for each level of group:

```
Tapply(post.test.3 ~ group, mean, data=Baumann)
```

|      | Basal  | DRTA   | Strat  |
|------|--------|--------|--------|
| mean | 41.045 | 46.727 | 44.273 |

```
Tapply(post.test.3 ~ group, sd, data=Baumann)
```

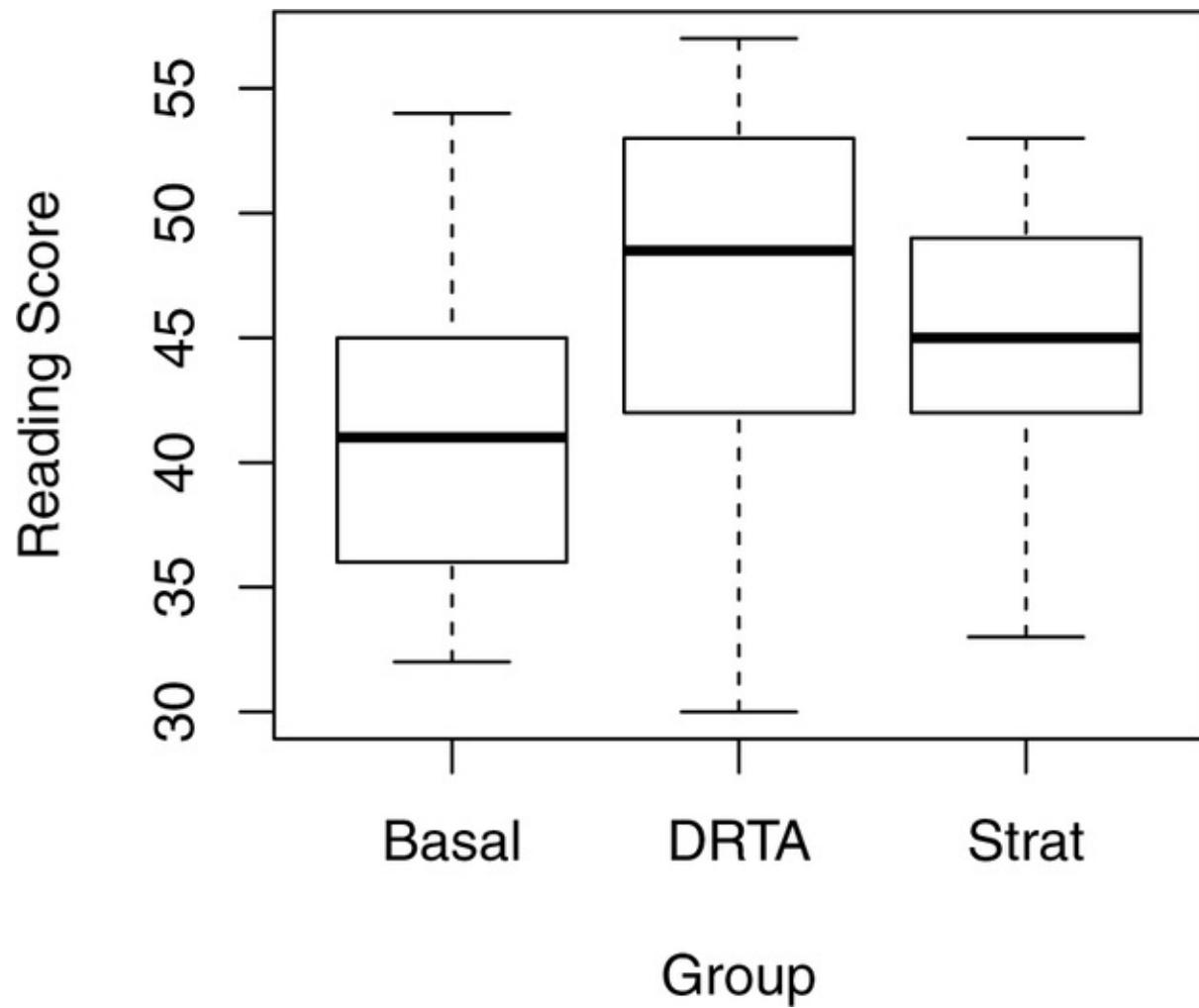
|    | Basal  | DRTA   | Strat  |
|----|--------|--------|--------|
| sd | 5.6356 | 7.3884 | 5.7668 |

Tapply (), which is described in [Section 10.5](#), provides a formula interface to R's standard tapply () function. The first command computes the mean of post. test.3 for every level of group in the Baumann data set. The second command uses sd to compute the standard deviations. The means differ somewhat, while the within-group SDs are similar to each other.

Plotting a numeric variable such as post.test.3 against a factor like group produces parallel boxplots, as in [Figure 4.7](#):

```
plot(post.test.3 ~ group, data=Baumann, xlab="Group", ylab="Reading Score")
```

**Figure 4.7** Posttest reading score by condition, for Baumann and Jones's experiment.



The means and boxplots suggest that there may be systematic differences in level among the groups, particularly between the new methods and the standard one.

The one-way ANOVA model can be fit with `lm ()`:

```
S(baum.mod.1 <- lm(post.test.3 ~ group, data=Baumann))  
Call: lm(formula = post.test.3 ~ group, data = Baumann)
```

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 41.05    | 1.35       | 30.49   | <2e-16   |
| groupDRTA   | 5.68     | 1.90       | 2.98    | 0.004    |
| groupStrat  | 3.23     | 1.90       | 1.70    | 0.095    |

Residual standard deviation: 6.31 on 63 degrees of freedom  
Multiple R-squared: 0.125

F-statistic: 4.48 on 2 and 63 DF, p-value: 0.0152

| AIC    | BIC    |
|--------|--------|
| 435.48 | 444.24 |

We enter the predictor group directly into the formula for the linear model. Because R recognizes that group is a factor, it is replaced by dummy regressors labeled "groupDRTA" and "groupStrat" in the output, prepending the factor name to the level labels. The residual standard deviation shown in the output is the usual pooled estimate of  $\sigma$ , which combines the within-group standard deviations. The coefficients for the two regressors are the estimates of the *differences* in means between the groups shown and the baseline group—the first level of group, "Basal". For example, the estimated mean difference between the "DRTA" and "Basal" groups is 5.68. The *t*-value for the groupDRTA coefficient is therefore the Wald test that the corresponding population mean difference is equal to zero. The intercept is the estimated mean for the baseline "Basal" group. No test is directly available in the summary output to compare the levels "DRTA" and "Strat" or more generally for any pairwise comparison of levels that does not include the baseline level.

To obtain all pairwise comparisons of the means for the three groups, we introduce the `emmeans()` function (an acronym for “estimated marginal means”) in the **emmeans** package (Lenth, 2018):<sup>28</sup>

```

library("emmeans")
emmeans(baum.mod.1, pairwise ~ group)

$emmeans
  group emmean     SE df lower.CL upper.CL
  Basal 41.045 1.3462 63    38.355   43.736
  DRTA  46.727 1.3462 63    44.037   49.417
  Strat 44.273 1.3462 63    41.583   46.963

Confidence level used: 0.95

$contrasts
  contrast      estimate     SE df t.ratio p.value
  Basal - DRTA    -5.6818 1.9038 63   -2.985  0.0111
  Basal - Strat   -3.2273 1.9038 63   -1.695  0.2150
  DRTA - Strat     2.4545 1.9038 63    1.289  0.4064

P value adjustment:
  tukey method for comparing a family of 3 estimates

```

[28](#) The **emmeans** package replaces an earlier package called **lsmeans** (Lenth, 2016). The **lsmeans** package (an acronym for the less sensible term *least-squares means*) is no longer supported, and all its functionality is available in **emmeans**.

The first argument to `emmeans()` is the "lm" object `baum.mod.1`, representing the one-way ANOVA model that we fit to the Baumann data. The `formula` argument for `emmeans()` is idiosyncratic: The left-hand side of the formula specifies pairwise comparisons between group means, and the right-hand side specifies the factor or, more generally, the factors that define grouping.

The `emmeans()` function computes *adjusted means*, displayed in the part of the output labeled `$emmeans`, and all pairwise differences of adjusted means, in the section labeled `$contrasts`. Because this model has only one predictor, the factor `group`, no adjustment is necessary, and the values in the column labeled `emmeans` are simply the group means we computed previously. In more complex

linear models, adjusted and directly computed means are typically different. The next column, labeled SE, displays the standard errors of the (adjusted) means, which, in a oneway ANOVA, are the residual standard deviation divided by the square root of the group sample size. All the SEs are equal because the group sample sizes are the same and the pooled estimate of  $\sigma$  is used for all computations. The last two columns provide 95% confidence intervals for the means based on the Wald statistic. For the current linear model, the 95% confidence interval for each mean is the estimate plus or minus its SE times the 0.975 quantile of the  $t$ -distribution with 63  $df$ .

The second part of the output shows all possible pairwise differences of adjusted means and provides  $p$ -values for  $t$ -tests adjusted for multiple comparisons using, in this instance, Tukey's HSD ("honestly significant difference") method (Bretz, Hothorn, & Westfall, 2011, Section 4.2). The emmeans () function chooses an appropriate multiple-comparison method depending on the problem. Only the difference between Basal and DRTA has a small  $p$ -value, so we tentatively suggest that only this difference is worthy of attention.

One could argue that the interesting comparisons are between Basal and each of the other treatments, producing only two paired comparisons. This set of comparisons can be carried out via the trt.vs.ctrl specification to emmeans (), which compares the baseline level of a factor to all other levels:

```
emmeans (baum.mod.1, trt.vs.ctrl ~ group)
```

```
$emmeans
```

| group | emmean | SE     | df | lower.CI | upper.CI |
|-------|--------|--------|----|----------|----------|
| Basal | 41.045 | 1.3462 | 63 | 38.355   | 43.736   |
| DRTA  | 46.727 | 1.3462 | 63 | 44.037   | 49.417   |
| Strat | 44.273 | 1.3462 | 63 | 41.583   | 46.963   |

Confidence level used: 0.95

```
$contrasts
```

| contrast      | estimate | SE     | df | t.ratio | p.value |
|---------------|----------|--------|----|---------|---------|
| DRTA - Basal  | 5.6818   | 1.9038 | 63 | 2.985   | 0.0079  |
| Strat - Basal | 3.2273   | 1.9038 | 63 | 1.695   | 0.1712  |

P value adjustment: dunnettx method for 2 tests

While the basic statistics are the same as before, the method of multiple comparisons (Dunnet's *t*, Bretz et al., 2011, Section 4.1) is different, and so the *p*-values change slightly. The **emmeans** package includes a vignette that carefully explains the many options available.

## 4.5.2 Additive Models With Numeric Predictors and Factors

Regression analysis with factors and numeric predictors is sometimes called *dummy-variable regression* or *analysis of covariance*, and the numeric predictors may be called *covariates*. To illustrate, we return to the Prestige data set, but this time we use `update()` to remove the numeric predictor `women` from the regression model `prestige.mod` (fit on page 183), retaining the numeric predictors `education` and `income` and adding the factor type to the model:

```



```

The output indicates that four cases were deleted due to missing data, because four of the occupations ("athletes", "newsboys", "babysitters", and "farmers") are coded NA for type. This model has three predictors, education, income, and type, which produce four regressors plus an intercept, effectively one intercept for each level of the factor type and one slope for each of the numeric predictors. The intercepts are  $-81.2$  for "bc",  $-81.2 + (-1.44) = -82.64$  for "wc", and  $-81.2 + 6.75 = -74.45$  for "prof". The slope for education is 3.28 and the slope for  $\log_2(\text{income})$  is 7.27. In this additive model, the covariate slopes are the same for every level of type. We consider models with differing slopes in [Section 4.6.1](#).

The fitted model is summarized graphically in the predictor effect plots in [Figure 4.8](#):

```

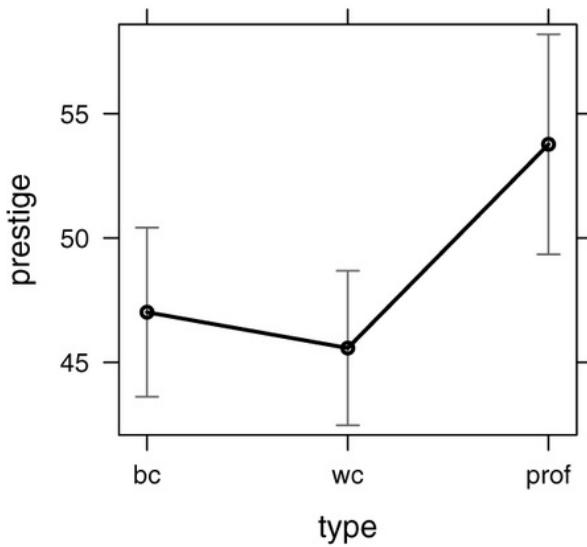
plot(predictorEffects(prestige.mod.1, predictors = ~ type + education +
income))

```

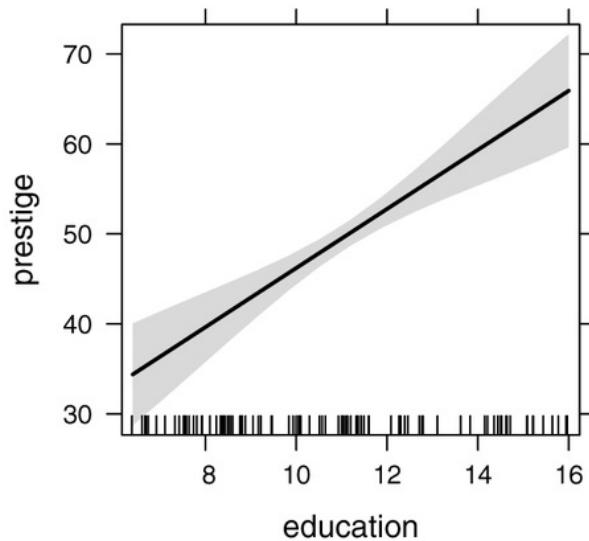
**Figure 4.8** Predictor effect plots for the additive dummy-regression model

prestige.mod.1.

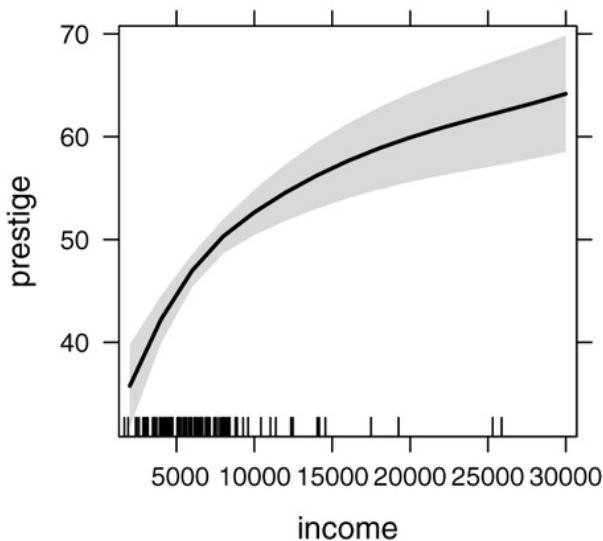
**type predictor effect plot**



**education predictor effect plot**



**income predictor effect plot**



We use the predictors argument to display the panels in the graph in the order in which we intend to discuss them. The plot for type shows the adjusted means for the three groups, defined as the estimated means for each level of type with all other predictors set to their sample mean values.<sup>29</sup> The lines connecting the three adjusted means are a visual aid only, because adjusted means are defined only for the levels of the factor. The vertical error bars are drawn by default to display the individual 95% confidence interval for each of the adjusted means. These error bars describe the variability in the adjusted means.

[29](#) Had the model included additional factors, we would calculate adjusted means for the levels of a particular factor by averaging over the levels of other factors, weighting by sample size. In calculating the predictor effects for the numeric predictors education and income, the levels of type are averaged over in this manner. Although this approach seems complicated, it is equivalent to substituting the sample means of the contrasts for a factor into the fitted linear model. The **effects** package also allows users to average over factors in other ways.

Standard errors for differences in adjusted means, and tests corrected for simultaneous statistical inference, may be computed by the emmeans () function:

```
emmeans(prestige.mod.1, pairwise ~ type)$contrasts
```

| contrast  | estimate | SE     | df | t.ratio | p.value |
|-----------|----------|--------|----|---------|---------|
| bc - wc   | 1.4394   | 2.3780 | 93 | 0.605   | 0.8176  |
| bc - prof | -6.7509  | 3.6185 | 93 | -1.866  | 0.1545  |
| wc - prof | -8.1903  | 2.5882 | 93 | -3.165  | 0.0059  |

P value adjustment:

tukey method for comparing a family of 3 estimates

Only the adjusted means for "wc" and "prof" differ reliably, with a *p*-value of about .006.

Because we fit an additive model with no interactions, the effect of each of the covariates, education and income, is summarized by a single line, displayed in the remaining plots in [Figure 4.8](#). The plot for income is curved because this predictor is represented in the model by the regressor  $\log_2(\text{income})$ . The effect plot for income would be identical for any choice of base for the logarithms.

## 4.6 Linear Models With Interactions

The power of the linear-model formula notation used by lm () and other R statistical-modeling functions is most apparent in more complex models with higher-order terms that generate products of the regressors for different predictors. The predictors then *interact*, in the sense that the partial relationship

of the response to one of the interacting predictors varies with the value of another predictor or, in the case of several interacting predictors, with the values of the others.

We'll first explore modeling interactions between numeric predictors and factors, the simplest case, before turning to models with two or more interacting factors, and finally to interactions between numeric predictors.

### 4.6.1 Interactions Between Numeric Predictors and Factors

We have thus far allowed each level of a factor to have its own intercept, but the slopes for the numeric predictors in the model were constrained to be the same for all levels of the factor. Including interactions in the model accommodates different slopes at each level of a factor, *different slopes for different folks*.

Interactions in an R model formula are specified using the: (*colon*) operator. The interaction between type and education in the Prestige data set, for example, literally multiplies each of the two regressors for type by education to produce two additional interaction regressors, as can be verified by examining a few rows of the corresponding model matrix, selected randomly by the `some()` function in the **car** package:

```
some(model.matrix(~ type + education + education:type,  
  data=Prestige), 8)
```

|                | (Intercept) | typewc | typeprof | education |
|----------------|-------------|--------|----------|-----------|
| social.workers | 1           | 0      | 1        | 14.21     |
| pharmacists    | 1           | 0      | 1        | 15.21     |
| postal.clerks  | 1           | 1      | 0        | 10.07     |
| travel.clerks  | 1           | 1      | 0        | 11.43     |

|                     |   |                  |                    |       |
|---------------------|---|------------------|--------------------|-------|
| textile.labourers   | 1 | 0                | 0                  | 6.74  |
| sheet.metal.workers | 1 | 0                | 0                  | 8.40  |
| electricians        | 1 | 0                | 0                  | 9.93  |
| typesetters         | 1 | 0                | 0                  | 10.00 |
|                     |   | typewc:education | typeprof:education |       |
| social.workers      |   | 0.00             |                    | 14.21 |
| pharmacists         |   | 0.00             |                    | 15.21 |
| postal.clerks       |   | 10.07            |                    | 0.00  |
| travel.clerks       |   | 11.43            |                    | 0.00  |
| textile.labourers   |   | 0.00             |                    | 0.00  |
| sheet.metal.workers |   | 0.00             |                    | 0.00  |
| electricians        |   | 0.00             |                    | 0.00  |
| typesetters         |   | 0.00             |                    | 0.00  |

As before, the first column of the model matrix, a column of ones, is for the intercept. Because typewc, the dummy regressor for the "wc" level of the factor type, is either zero or 1, the interaction regressor typewc:education = typewc × education is zero whenever typewc is zero and is equal to education whenever typewc is 1; similarly, typeprof:education = typeprof × education is zero whenever typeprof is zero and is equal to education whenever typeprof is 1.

Using the update () function, we add interactions between  $\log_2$  (income) and type and between education and type to our previous model for the Prestige data:

```

prestige.mod.2 <- update(prestige.mod.1,
  . ~ . + education:type + log2(income):type)
s(prestige.mod.2)

Call: lm(formula = prestige ~ education + log2(income) + type +
  education:type + log2(income):type, data = Prestige)

```

Coefficients:

|                       | Estimate | Std. Error | t value | Pr(> t ) |
|-----------------------|----------|------------|---------|----------|
| (Intercept)           | -120.046 | 20.158     | -5.96   | 5.1e-08  |
| education             | 2.336    | 0.928      | 2.52    | 0.0136   |
| log2(income)          | 11.078   | 1.806      | 6.13    | 2.3e-08  |
| typewc                | 30.241   | 37.979     | 0.80    | 0.4280   |
| typeprof              | 85.160   | 31.181     | 2.73    | 0.0076   |
| education:typewc      | 3.640    | 1.759      | 2.07    | 0.0414   |
| education:typeprof    | 0.697    | 1.290      | 0.54    | 0.5900   |
| log2(income):typewc   | -5.653   | 3.052      | -1.85   | 0.0673   |
| log2(income):typeprof | -6.536   | 2.617      | -2.50   | 0.0143   |

Residual standard deviation: 6.41 on 89 degrees of freedom  
 (4 observations deleted due to missingness)

Multiple R-squared: 0.871

F-statistic: 75.1 on 8 and 89 DF, p-value: <2e-16

|        |        |
|--------|--------|
| AIC    | BIC    |
| 652.77 | 678.62 |

The regression model still has only the three predictors, education, income, and type, but it now comprises nine regressors, including the intercept. There are effectively one intercept and two slopes for each level of the factor type. In this model:

- For the baseline level "bc", the intercept is -120.05, the slope for education is 2.34, and the slope for  $\log_2(\text{income})$  is 11.08.
- For level "wc", the intercept is  $-120.05 + 30.24 = -89.81$ , the slope for education is  $2.34 + 3.64 = 5.98$ , and the slope for  $\log_2(\text{income})$  is  $11.08 + (-5.65) = 5.43$ .
- For level "prof", the intercept is  $-120.05 + 85.16 = -34.89$ , the slope for education is  $2.34 + 0.70 = 3.04$ , and the slope for  $\log_2(\text{income})$  is  $11.08 + (-6.54) = 4.54$ .

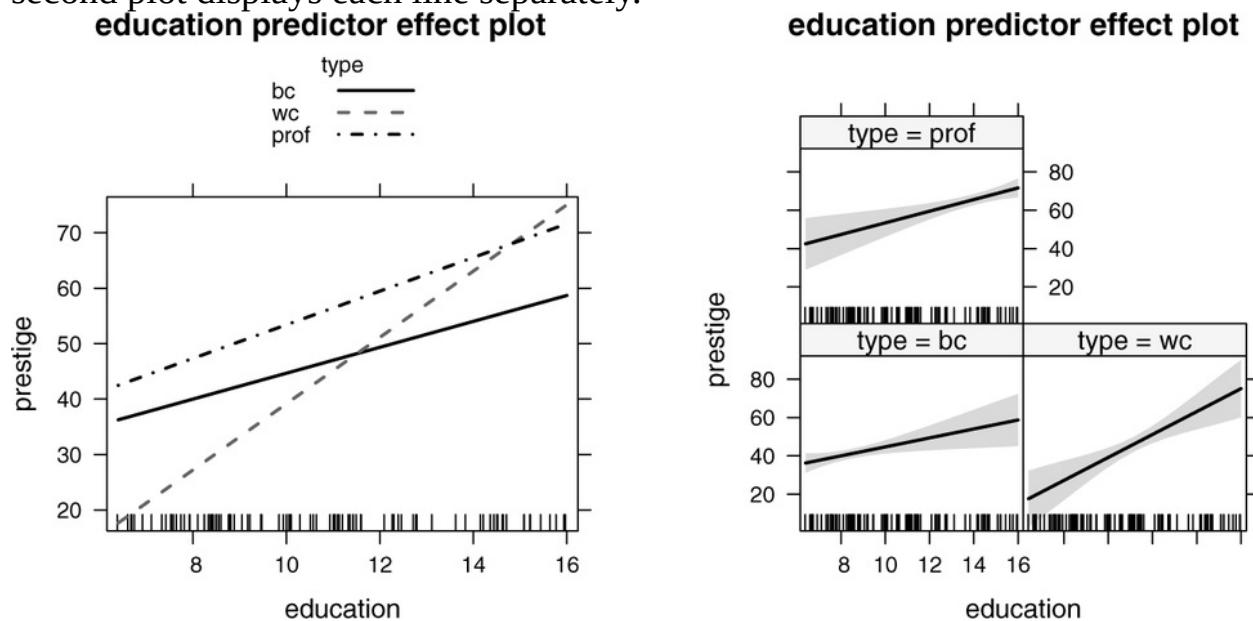
Calculating estimates of interesting quantities like within-level slopes from the coefficients of a model with interactions is generally straightforward but tedious. Further complicating the interpretation of parameters is their dependence on the choice of the contrasts that are used to represent factors, so readers of your work would need to know how you represented factors in the model to know how to interpret parameter estimates. We encourage the use of predictor effect plots to visualize and summarize fitted regression models because these plots do not depend on parametrization of factors and also are oriented toward displaying partial regression mean functions and adjusted means, which are usually the most interesting quantities.

[Figure 4.9](#) shows the predictor effect plot for education in the model `prestige.mod.2`, drawn in two ways:

```
plot (predictorEffects (prestige.mod.2, ~ education), lines=list  
(multiline=TRUE))
```

```
plot (predictorEffects (prestige.mod.2, ~ education))
```

**Figure 4.9** Alternative predictor effect plots for education in the model `prestige.mod.2`. The first plot overlays the lines for each level of type, while the second plot displays each line separately.



In both cases, the fitted values of prestige are shown as a joint function of

education and type, setting income to its sample mean. The value to which income is fixed affects only the labeling of the vertical axis (i.e., the height of the graph) and not its shape (the configuration of fitted values).

The first predictor effect plot in [Figure 4.9](#) displays the estimated slopes for education at the three levels of type in a single panel. Each line is drawn with a different line type, as shown in the key at the top of the graph.<sup>30</sup> The education slope is largest for "wc" and smallest for "bc" occupations; except at the highest level of education, where there are no "wc" or "bc" occupations, "prof" occupations have higher average prestige, controlling for income, than the other occupational levels.

[30](#) If color were available, as it is on a computer screen, the three lines would be differentiated by color.

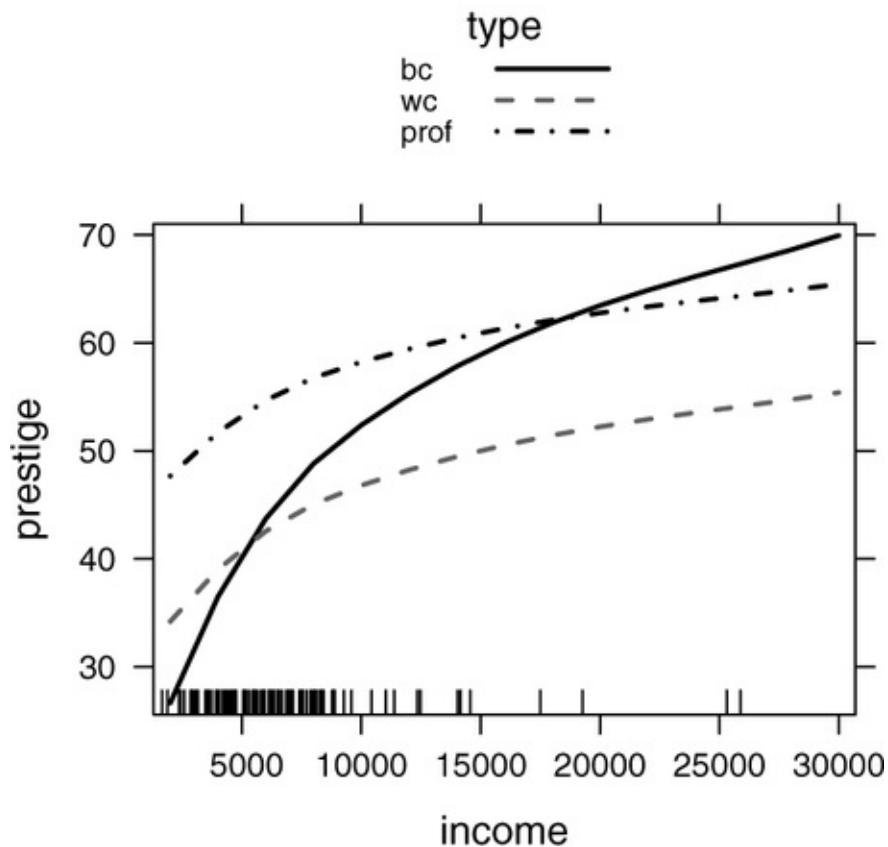
The second graph, at the right of [Figure 4.9](#), displays the same information, but each level of type appears in a separate panel, showing pointwise confidence regions around the fitted lines, which are not included by default in the multiline graph. The confidence regions are wider for lower education "prof" jobs, for higher education "bc" jobs, and at both extremes for "wc" jobs—that is, in regions of education where data are sparse or absent. We often draw plots both with multiline=TRUE to compare the levels of a factor, and with the default multiline=FALSE, to examine uncertainty in the estimated effect. You can add confidence bands to a multiline effect plot with the argument confint=list (style="bands") to plot (), but the resulting graph is often too visually busy to be of much use (reader: try it for the current example).

[Figure 4.10](#) shows the multiline predictor effect plot for income:

```
plot (predictorEffects (prestige.mod.2, ~ income), lines=list  
      (multiline=TRUE))
```

**Figure 4.10** Predictor effect plot for income in the model prestige.mod.2.

## income predictor effect plot



The lines in the plot are curved because the regressor representing income appears in the model in log scale. Prestige increases fastest with income for "bc", while the curves for "wc" and "prof" are nearly parallel, suggesting that the effect of income is similar for these two levels. Except at the far right, where there are no "bc" or "wc" occupations, average prestige controlling for education is higher for "prof" occupations than for the other two occupational types.

The predictor effect plot for type is more complex than the plots for the other two predictors because type interacts with *both* education and income, and so adjusted means for levels of type need to be computed for each combination of education and income:

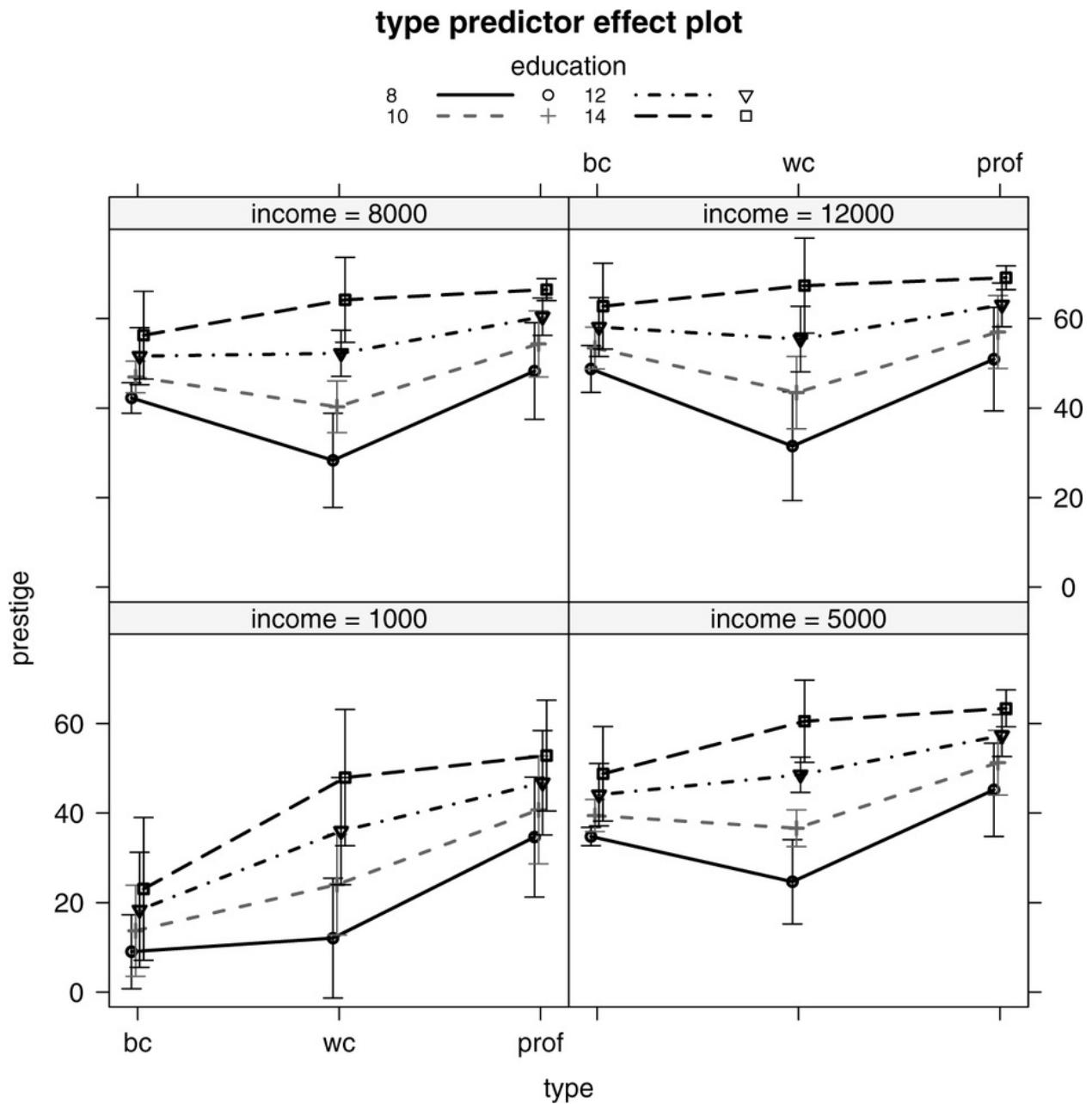
```
plot (predictorEffects (prestige.mod.2, ~ type,  
xlevels=list (income=c (1000, 5000, 8000, 12000),  
education=c (8, 10, 12, 14))),
```

```
lines=list (multiline=TRUE), confint=list (style="bars"))
```

The `xlevels` argument to `predictorEffects()` sets each of the predictors `income` and `education` to four different values. The default is to use five approximately equally spaced values for each predictor, but in this instance, the results are hard to decode with five levels. We use the `xlevels` argument to choose the values of `education` for conditioning at 8, 10, 12, and 14 years, and for `income` at \$1,000, \$5,000, \$8,000, and \$12,000. Once again, we use the argument `multiline=TRUE`.

The four lines in each panel of [Figure 4.11](#) are for the four specified values of the predictor `education`. Each panel of the plot is for one of the four values of `income`, as recorded in the strip at the top of the panel. An unusual convention of lattice graphics is that the panels are ordered from the bottom row to the top row, so the smallest income level is in the lower-left panel and the largest in the upper-right panel.<sup>31</sup> The plotted values are the fitted means for the three levels of occupational type, with `education` and `income` set to 16 combinations of values. Finally, we use the `confint` argument to add error bars around the fitted means. The plotted points are automatically staggered slightly so that the error bars won't overplot each other. Some of the combinations of values of the predictors—for example, "bc" or "wc" occupations with 14 years of education or income above \$8,000—don't occur in the data and consequently are associated with very large confidence intervals for their fitted means; after all, `prestige.mod.2` is fit to only 98 cases.

**Figure 4.11** Predictor effect plot for type in the model `prestige.mod.2`.



[31](#) The **lattice** package, described more generally in [Section 9.3.1](#), is used to draw effect plots. Some aspects of the plot can be controlled by the lattice argument to plot(); see help ("plot.eff").

If type and education didn't interact fixing the value of income, all the lines in each panel of [Figure 4.11](#) would be parallel. That is not the case, where we see that in general, the estimated level of prestige varies more by education for "wc" occupations than for "bc" or "prof" occupations. Occupations of type "prof" have the highest average prestige for all specified combinations of levels of income

and education.

Interactions between factors and numeric regressors extend to multiple-*df* terms for numeric predictors, such as regression splines and polynomial regressors. For example, returning to the SLID data (from [Section 4.4](#)), we can specify a model with nonlinear interactions between sex and each of education and age:

```
mod.slid.int <- lm (log (wages) ~  
  sex*(ns (education, df=5) + poly (age, degree=2)), data=SLID,  
  subset = age >= 18 & age <= 65)  
  
plot (predictorEffects (mod.slid.int, predictors = ~ education + age,  
  transformation=list (link=log, inverse=exp)),  
  lines=list (multiline=TRUE), confint=list (style="bands"),  
  axes=list (y=list (lab="Hourly Wage Rate")))
```

Using \* in this manner on the right-hand side of the model formula adds main effects and interactions for sex and education and for sex and age to the model; we explain this compact notation immediately below. Thus, the potentially non-linear flexible relationship of log (wages) to education, and the quadratic relationship of log (wages) to age, can each be different for women and men, as is apparent in the predictor effect plots for education and age in [Figure 4.12](#). In constructing these plots, we use the optional transformation argument to predictorEffects () to undo the log transformation of wages, the lines argument to plot () to obtain multiline graphs, the confint argument to draw confidence bands, and the axes argument to customize the y-axis label. The predictor effect plot for education sets age to its average value in the data, and the plot for age sets education to its average value. We invite the reader to draw and examine the predictor effect plot for sex:

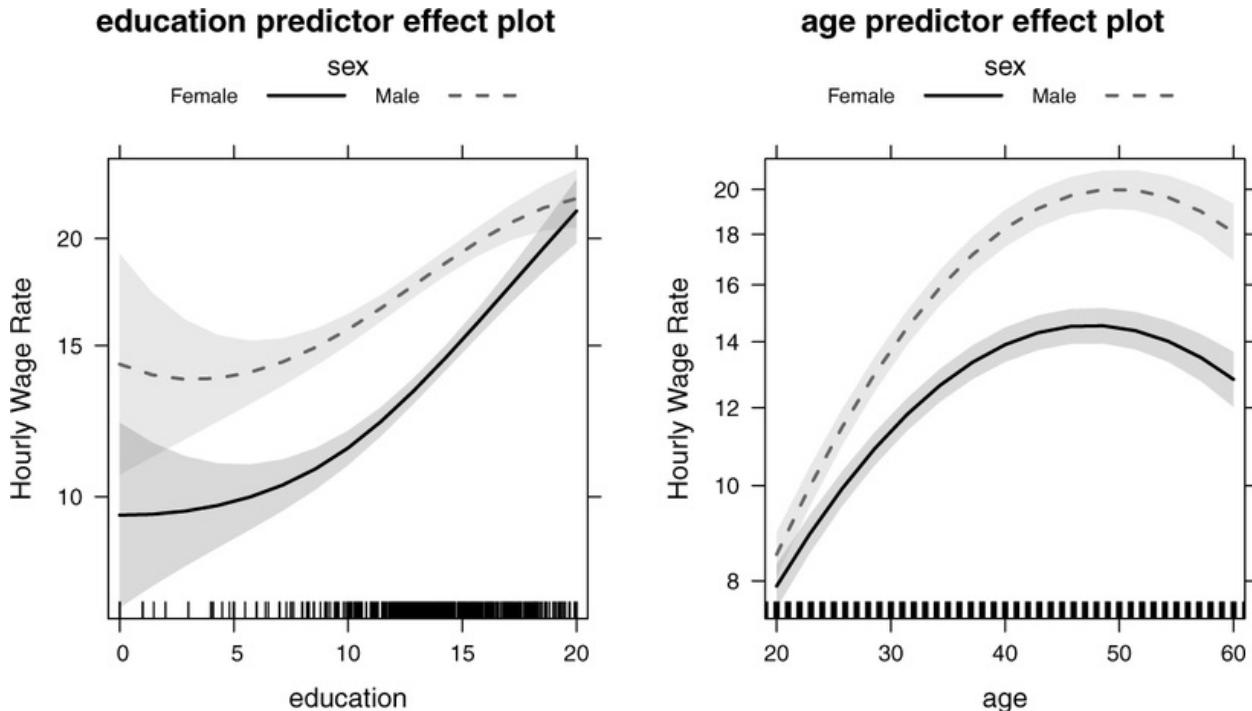
```
plot (predictorEffects (mod.slid.int, predictors= ~ sex, transformation=list  
  (link=log, inverse=exp))),
```

```

  lines=list (multiline=TRUE), confint=list (style="bars"),
  axes=list (y=list (lab="Hourly Wage Rate")))

```

**Figure 4.12** Predictor effect plots for education and age in the model `slid.mod.int`.



## 4.6.2 Shortcuts for Writing Linear-Model Formulas

In this section, we describe some convenient shortcuts that simplify linear-model formulas in R.<sup>32</sup> Recall that the plus sign (+) adds a term to a formula, the minus sign (-) removes a term, and the colon (:) creates an interaction.

1. The term  $a*b$ , which uses the \* *crossing operator*, is equivalent to  $a + b + a:b$ .
2. The term  $a*(b + c)$  is equivalent to  $a*b + a*c$ , which expands to  $a + b + c + a:b + a:c$ .
3. The term  $(a + b + c)^2$  is equivalent to  $a + b + c + a:b + a:c + b:c$ .
4. The term  $a*b*c$  generates the three-predictor interaction and all of its lower-order relatives and so is equivalent to  $a + b + c + a:b + a:c + b:c + a:b:c$ .

[32](#) Also see [Section 4.9.1](#).

For example, the formula for the model `prestige.mod.2` can be written in any of the following equivalent ways:

```
prestige ~ log2(income) + education + type  
        + log2(income):type + education:type  
prestige ~ log2(income)*type + education*type  
prestige ~ (log2(income) + education)*type
```

We generally recommend using `*` to specify models with interactions because it is easier to make mistakes when the `:` operator is used directly. For example, as the reader can verify, [33](#) the model formula

`prestige ~ education:type + log2(income):type`

is *not* equivalent to

`prestige ~ education?type + log2(income)?type`

[33](#) The model `prestige ~ education?type + log2(income)?type` fits a common intercept for the three occupational groups and different education and `log2(income)` slopes for the groups, not a generally sensible specification; see [Section 4.9.1](#) for a more general discussion of the structure of linear-model formulas.

Model formulas in R are most straightforward when they conform to the *principle of marginality* (Nelder, 1977): *lower-order relatives* of a higher-order term, such as `A:B` or `A:B:C` (where A, B, and C could be factors, numeric predictors, or terms such as polynomials and splines), should be in the model if the higher-order term is in the model. The lower-order terms are said to be *marginal* to their higher-order relative. So, for example, a model that includes the three-way interaction `A:B:C` should also include the two-way interactions `A:B`, `A:C`, and `B:C`, and the main effects A, B, and C, all of which are marginal

to the three-way term. The interpretation of R model formulas that apparently violate marginality is more complicated (see [Section 4.9.1](#)). Terms generated by the `*` operator obey the principle of marginality.

### 4.6.3 Multiple Factors

In this section, we consider problems with only factors as predictors, using data drawn from the General Social Survey (GSS), conducted by the National Opinion Research Center of the University of Chicago (NORC, 2018). Our first example is based on the 2016 GSS, with the response variable `vocab`, the number of words out of 10 that each respondent correctly defined on a vocabulary test included in the survey. As predictors, we use the factors `nativeBorn`, with levels "yes" for U.S.-born respondents and "no" for others, and `ageGroup`, which assigns each respondent to one of the age categories "18-29", "30-39", "40-49", "50-59", "60+". The data for the example are included in the `GSSvocab` data set in the **carData** package, and we begin by obtaining a subset of the data that contains only these variables for the 2016 round of the GSS:

```
GSS16 <- na.omit (subset (GSSvocab, year=="2016", select=c ("vocab",  
"ageGroup", "nativeBorn")))
```

We quote "2016" in the call to the `subset ()` function because `year` is a factor in the data set, and its levels are therefore character values. We call the `na.omit ()` function to remove respondents with missing data for any of the three variables.

The two factors divide the data into 10 *cells*, defined by the combinations of levels of `nativeBorn` and `ageGroup`, with the following cell counts:

```
xtabs (~ nativeBorn + ageGroup, data=GSS16)
```

|            |     | ageGroup |       |       |       |     |
|------------|-----|----------|-------|-------|-------|-----|
|            |     | 18-29    | 30-39 | 40-49 | 50-59 | 60+ |
| nativeBorn | no  | 37       | 52    | 46    | 45    | 52  |
|            | yes | 283      | 295   | 216   | 316   | 516 |

All 10 within-cell counts are greater than zero, with more native-born than non-native-born respondents in each age group. The sample sizes are unequal, as is typical in the analysis of observational data like these. Many experiments, in contrast, are designed to have equal sample sizes in each cell, sometimes termed *balanced data*, which simplifies interpretation of results because regressors representing different factors are then uncorrelated. The regression models we employ to analyze problems with multiple factors can be applied with equal or unequal cell counts.

We call the `Tapply()` function to compute cell means and within-cell standard deviations:

```
Tapply(vocab ~ nativeBorn + ageGroup, mean, data=GSS16)
```

| ageGroup   |        |        |        |        |        |
|------------|--------|--------|--------|--------|--------|
| nativeBorn | 18-29  | 30-39  | 40-49  | 50-59  | 60+    |
| no         | 4.4595 | 5.3846 | 5.5217 | 5.4889 | 5.2308 |
| yes        | 5.8092 | 6.0034 | 6.2407 | 6.1614 | 6.2926 |

```
Tapply(vocab ~ nativeBorn + ageGroup, sd, data=GSS16)
```

| ageGroup   |        |        |        |        |        |
|------------|--------|--------|--------|--------|--------|
| nativeBorn | 18-29  | 30-39  | 40-49  | 50-59  | 60+    |
| no         | 2.3874 | 2.5061 | 2.2185 | 2.0629 | 2.5019 |
| yes        | 1.5797 | 1.7947 | 1.6586 | 1.8316 | 2.0014 |

All of the within-cell means fall in the range from about 4.5 to 6.3 words correct. Nonnative-born respondents have a smaller mean for each age group, by less than one word. With one exception, standard deviations are generally less than 2.0 words in the native-born cells and somewhat larger than 2.0 in the nonnative-born cells. The models that we fit assume that the population SD is the same in each cell, and for this presentation we will imagine that this assumption is tenable; we return to the issue in [Section 5.1.2.](#)<sup>34</sup>

<sup>34</sup> Although the standard deviations appear to be systematically smaller in the native-born cells, the differences are not great. Only large differences in within-cell standard deviations—indicated, for example, by sample standard deviations that differ by a factor of two or more—invalidate the analysis-of-variance tests

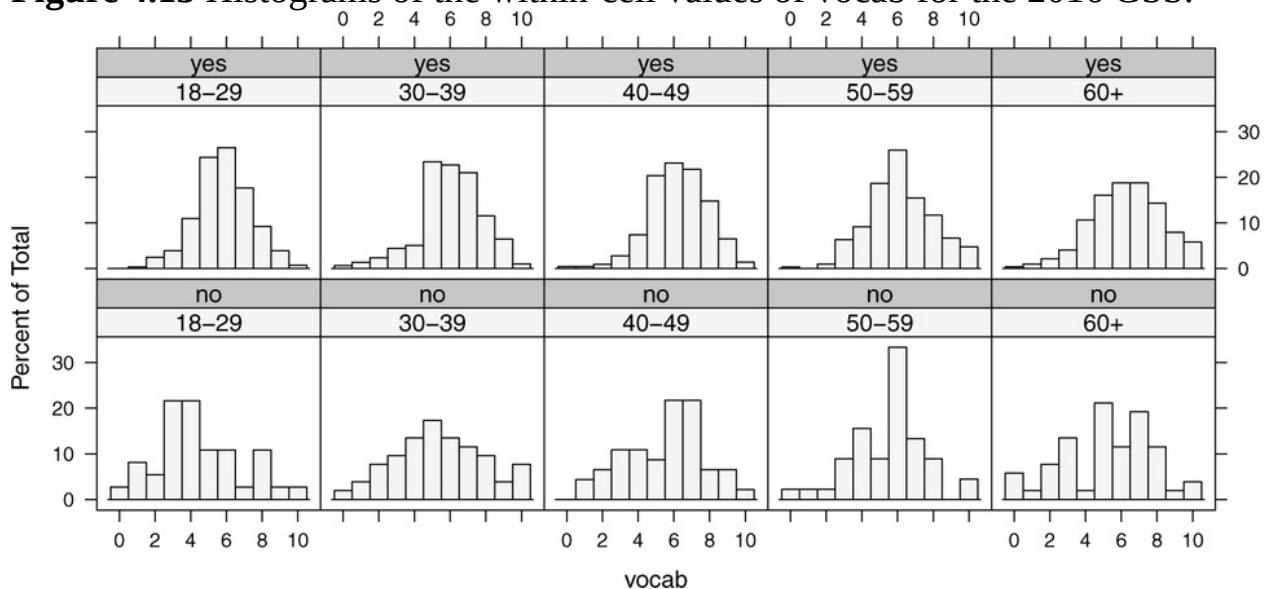
performed for multifactor linear models. We discuss nonconstant error variance in a more general context in [Section 8.5](#).

[Figure 4.13](#) provides a visual summary of the data using within-cell histograms:

**library (*lattice*)**

**histogram (~ vocab | ageGroup + nativeBorn, data=GSS16, breaks=seq (-.5, 10.5, length=12))**

**Figure 4.13** Histograms of the within-cell values of vocab for the 2016 GSS.



We use the `histogram()` function in the **lattice** package to draw the graph (see [Section 9.3.1](#)). The `breaks` argument specifies 11 bins, centered at the values 0, 1, ..., 10; because the response is discrete, with integer values, each bar in a histogram is the percent of the respondents with `vocab` equal to a particular value. Respondents with all possible values of `vocab` exist in every cell. The histograms are generally unimodal, and in some cases they are skewed. Linear regression models easily cope with behavior like this with large enough within-cell sample sizes because estimates are based on the cell means, and these are approximately normally distributed even when the values of `vocab` within a cell are not normally distributed.<sup>35</sup>

<sup>35</sup> If the response is strongly skewed, which isn't the case here, the mean may not be a good summary of the center of the data. In [Sections 3.4](#) and [8.4](#), we

discuss how to transform data to approximate symmetry.

The largest model we can fit to these data includes an intercept, main effects for each of the factors, and the interaction between them:

```
mod.vocab.1 <- lm (vocab ~ nativeBorn*ageGroup, data=GSS16)
```

A model like this, with two or more interacting factors and no numeric predictors, is sometimes termed a *multiway ANOVA model* —in this case, a *two-way ANOVA*.

```
data.frame(coef.name=paste("b", 0:9, sep=""),
estimate=coef(mod.vocab.1))
```

|                             | coef.name | estimate |
|-----------------------------|-----------|----------|
| (Intercept)                 | b0        | 4.45946  |
| nativeBornyes               | b1        | 1.34973  |
| ageGroup30-39               | b2        | 0.92516  |
| ageGroup40-49               | b3        | 1.06228  |
| ageGroup50-59               | b4        | 1.02943  |
| ageGroup60+                 | b5        | 0.77131  |
| nativeBornyes:ageGroup30-39 | b6        | -0.73095 |
| nativeBornyes:ageGroup40-49 | b7        | -0.63073 |
| nativeBornyes:ageGroup50-59 | b8        | -0.67722 |
| nativeBornyes:ageGroup60+   | b9        | -0.28786 |

The `data.frame()` function allows us to create a compact table of the coefficient estimates. The first column gives the names of the coefficients that are automatically created by `lm()`. The second column names the coefficients according to the convention used in this book, as  $b_0, \dots, b_9$ . The last column displays the values of the  $b_j$ . Because there are 10 coefficients to fit the 10 cell means, the fitted values, computed according to the following table, match the

observed cell means exactly:

| ageGroup | nativeBorn  |                         |
|----------|-------------|-------------------------|
|          | no          | yes                     |
| 18-29    | $b_0$       | $b_0 + b_1$             |
| 30-39    | $b_0 + b_2$ | $b_0 + b_1 + b_2 + b_6$ |
| 40-49    | $b_0 + b_3$ | $b_0 + b_1 + b_3 + b_7$ |
| 50-59    | $b_0 + b_4$ | $b_0 + b_1 + b_4 + b_8$ |
| 60+      | $b_0 + b_5$ | $b_0 + b_1 + b_5 + b_9$ |

We can use the `Effect()` function in the **effects** package to verify that the fitted values in each cell are the same as the cell means by comparing them to the table of cell means we previously computed directly (on page 215):

```
Effect(c("nativeBorn", "ageGroup"), mod.vocab.1)
```

| nativeBorn*ageGroup effect |        |        |        |        |        |
|----------------------------|--------|--------|--------|--------|--------|
| ageGroup                   |        |        |        |        |        |
| nativeBorn                 | 18-29  | 30-39  | 40-49  | 50-59  | 60+    |
| no                         | 4.4595 | 5.3846 | 5.5217 | 5.4889 | 5.2308 |
| yes                        | 5.8092 | 6.0034 | 6.2407 | 6.1614 | 6.2926 |

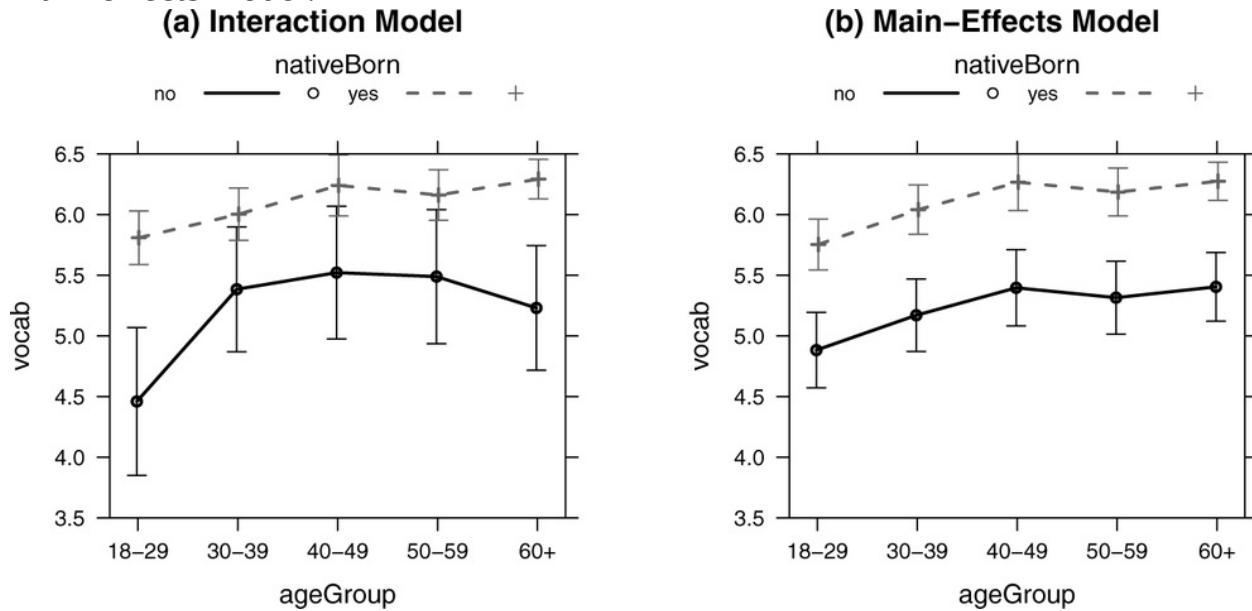
In problems with only factors and no numeric predictors, the analysis usually centers on examination of the estimated cell means under various models and on tests summarized in an ANOVA table, not on the parameter estimates. If alternative contrasts were used to define the regressors for the factors, the formulas relating parameter estimates to cell means would change, but the estimated cell means and the Type II ANOVA table, the default choice for the `Anova()` function in the **car** package, would stay the same. Concentrating on cell means renders choice of contrasts moot.<sup>36</sup>

[36](#) We discuss ANOVA tables, including Type II tests, extensively in [Section 5.3.4](#) and explain how to use alternative contrast codings in [Section 4.7.1](#).

The effect plot for model mod.vocab.1, which includes the nativeBorn  $\times$  ageGroup, interaction is shown in [Figure 4.14 \(a\)](#):

```
plot(Effect(c("nativeBorn", "ageGroup"), mod.vocab.1),
  main= "(a) Interaction Model", confint=list(style="bars"),
  lines=list(multiline=TRUE), ylim=c(3.5, 6.5))
```

**Figure 4.14** Mean vocab by nativeBorn and ageGroup for the 2016 General Social Survey. The error bars show nominal 95% confidence intervals for each cell mean assuming constant error variance. (a) Model with interactions; (b) main-effects model.



The lines connecting the points on the plot are for reference only, as the model is defined only for the 10 combinations of the factor levels. As we previously noted, native-born respondents have larger sample mean vocab for all age groups, but the differences between the two levels of nativeBorn are not the same for all ages, with larger differences for the youngest and oldest groups. The error bars are computed based on a pooled estimate of the within-cell standard deviation, which is the residual SD for the model. The half-length of an error bar

in a model with as many parameters as cells and a large sample size is the pooled SD divided by the square root of the number of observations in the cell times 1.96 for nominal 95% confidence intervals.

The emmeans () function in the **emmeans** package can be used to obtain tests comparing levels of nativeBorn for each age group:

```
emmeans(mod.vocab.1, pairwise ~ nativeBorn | ageGroup) $contrasts  
  
ageGroup = 18-29:  
  contrast estimate      SE   df t.ratio p.value  
  no - yes -1.34973 0.33075 1848  -4.081  <.0001  
  
ageGroup = 30-39:  
  contrast estimate      SE   df t.ratio p.value  
  no - yes -0.61877 0.28456 1848  -2.175  0.0298  
ageGroup = 40-49:  
  contrast estimate      SE   df t.ratio p.value  
  no - yes -0.71900 0.30723 1848  -2.340  0.0194  
  
ageGroup = 50-59:  
  contrast estimate      SE   df t.ratio p.value  
  no - yes -0.67250 0.30146 1848  -2.231  0.0258  
  
ageGroup = 60+:  
  contrast estimate      SE   df t.ratio p.value  
  no - yes -1.06187 0.27528 1848  -3.857  0.0001
```

The multiple-comparison corrected *p*-values are very small for the youngest and oldest age groups and marginal for the intermediate age groups.

A first-order, main-effects, or additive model is obtained by deleting the interaction from the full model:

```

mod.vocab.2 <- update(mod.vocab.1, . ~ . - nativeBorn:ageGroup)

plot(Effect(c("nativeBorn", "ageGroup"), mod.vocab.2),
     main = "(b) Main-Effects Model", lines = list(multiline = TRUE),
     confint = list(style = "bars"), ylim = c(3.5, 6.5))
```

The main-effects model forces the profiles in [Figure 4.14 \(b\)](#) to be parallel, so the fitted cell means are no longer exactly equal to the observed cell means. To help compare the two graphs, we use the ylim argument to ensure that both have the same range on the vertical axis. The error bars are generally shorter for the main-effects model because more data are used to estimate each of the adjusted means than are used in the interaction model, and, as it turns out, the residual SDs for the two models are similar.

If the interactions displayed in [Figure 4.14 \(a\)](#) are only due to chance, then the data can be usefully summarized by considering the main effects of the two factors separately:

**emmeans (mod.vocab.2, pairwise ~ nativeBorn) \$contrasts**

| contrast | estimate | SE      | df   | t.ratio | p.value |
|----------|----------|---------|------|---------|---------|
| no - yes | -0.87074 | 0.13329 | 1852 | -6.533  | <.0001  |

Results are averaged over the levels of: ageGroup

**emmeans (mod.vocab.2, pairwise ~ ageGroup) \$contrasts**

| contrast      | estimate   | SE      | df   | t.ratio | p.value |
|---------------|------------|---------|------|---------|---------|
| 18-29 - 30-39 | -0.2873441 | 0.14671 | 1852 | -1.959  | 0.2869  |
| 18-29 - 40-49 | -0.5135776 | 0.15784 | 1852 | -3.254  | 0.0102  |
| 18-29 - 50-59 | -0.4322990 | 0.14527 | 1852 | -2.976  | 0.0247  |
| 18-29 - 60+   | -0.5213338 | 0.13228 | 1852 | -3.941  | 0.0008  |
| 30-39 - 40-49 | -0.2262335 | 0.15489 | 1852 | -1.461  | 0.5883  |
| 30-39 - 50-59 | -0.1449549 | 0.14228 | 1852 | -1.019  | 0.8468  |
| 30-39 - 60+   | -0.2339898 | 0.12915 | 1852 | -1.812  | 0.3669  |
| 40-49 - 50-59 | 0.0812786  | 0.15371 | 1852 | 0.529   | 0.9844  |
| 40-49 - 60+   | -0.0077563 | 0.14174 | 1852 | -0.055  | 1.0000  |
| 50-59 - 60+   | -0.0890348 | 0.12743 | 1852 | -0.699  | 0.9568  |

Results are averaged over the levels of: nativeBorn

P value adjustment:

tukey method for comparing a family of 5 estimates

Controlling for age, the native born on average have slightly, but reliably, higher vocabulary scores than the nonnative born. The youngest age group has lower vocab scores on average than all age groups older than 30–39, with no other reliable differences.

The full GSSvocab data set contains vocab scores and other variables from 20 rounds of the GSS conducted between 1978 and 2016, including data on the additional factors gender, with levels "female" and "male", and educGroup, the education level of the respondent, with levels "<12 yrs", "12 yrs" for high school graduates, "13-15 yrs" for some college, "16 yrs" for college graduates, and ">16 yrs" for postgraduate work. The larger data set thus has five potential factors, year with 20 levels, ageGroup with five levels, educGroup with five levels, gender with two levels, and nativeBorn with two levels, for a total of  $20 \times 5 \times 5$

$\times 2 \times 2 = 2,000$  cells. To estimate the full model with all main effects, two-factor interactions, ..., and the five-factor interaction, would require fitting 2,000 parameters to the 28,867 respondents in the data set.<sup>37</sup>

<sup>37</sup> Moreover, of the 2,000 cells in the cross-classification of the five factors, 164 have no data, implying that fewer than 2,000 parameters can be estimated; see [Section 4.8](#).

Regression models for analyzing problems of this kind typically include only main effects and perhaps a few two-factor or three-factor interactions and thus have relatively few parameters. How to choose the interactions to investigate will be partly addressed in [Chapter 5](#). In the current example, we include only the two-way nativeBorn by ageGroup interaction, described previously in the two-factor model for 2016 alone:

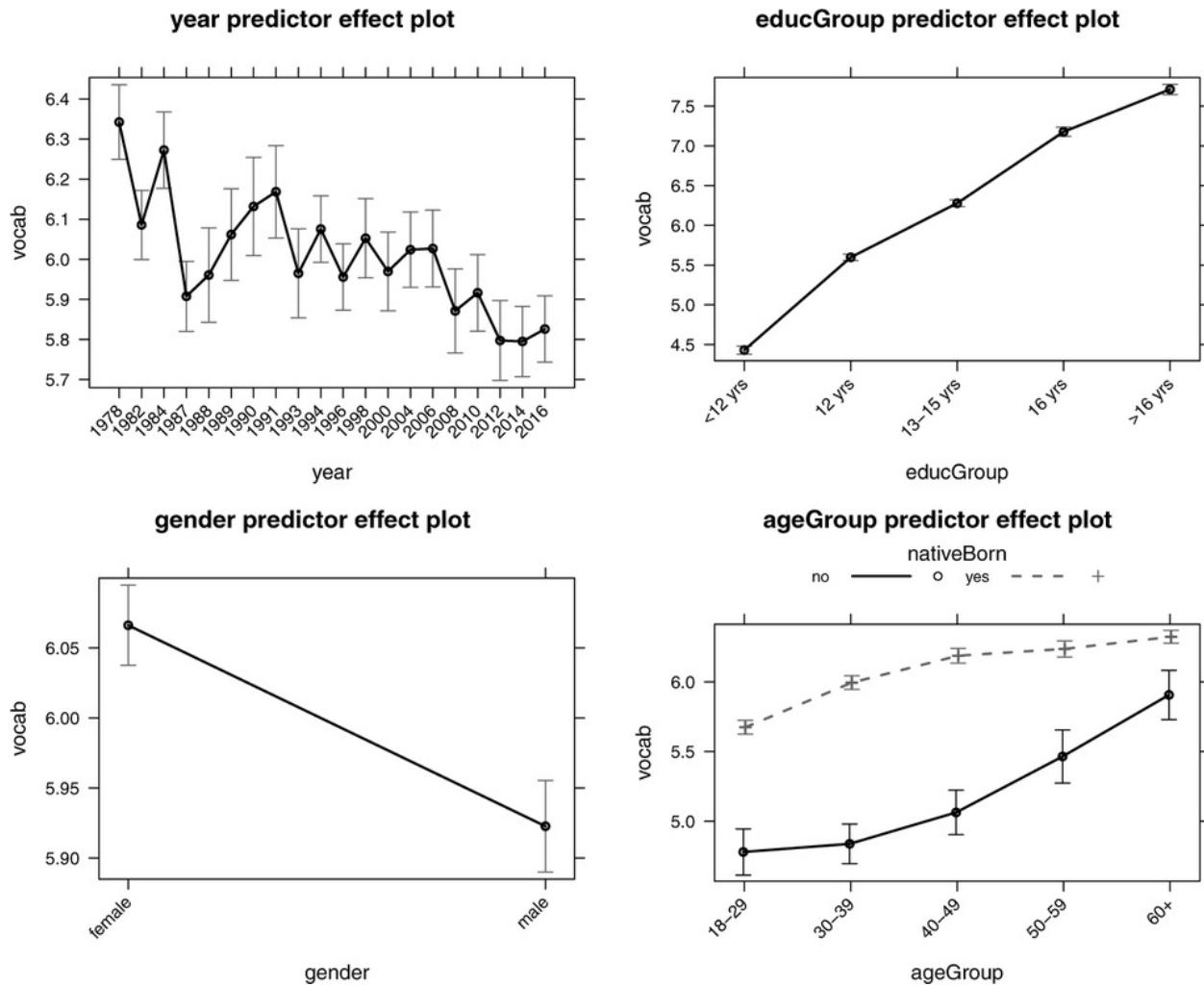
```
mod.vocab.3 <- lm (vocab ~ year + educGroup + gender +  
nativeBorn*ageGroup, data=GSSvocab)
```

Omitting any interactions with year, for example, implies that the effects of all the other predictors are the same in each year, although the average vocabulary score may still vary from year to year as a consequence of the year main effect. This is a strong assumption, and one would in general want to explore whether or not it is reasonable by examining models that include interactions with year. Similar comments apply to other missing interactions.

The model mod.vocab.3 fits 34 regression coefficients, a large number compared to most of the examples in this book, but small compared to the 2,000 cell means being modeled. We summarize the fitted model with the predictor effect plots shown in [Figure 4.15](#):

```
plot (predictorEffects (mod.vocab.3, ~ . - nativeBorn), axes=list (x=list  
(rotate=45)), lines=list (multiline=TRUE), confint=list (style="bars"))
```

**Figure 4.15** Predictor effect plots for the model mod.vocab.3 fit to all 20 years of the GSS.



The formula argument to `predictorEffects()` suppresses the predictor effect plot for `nativeBorn`, which would provide another, and as it turns out essentially redundant, view of the `nativeBorn`  $\times$  `ageGroup` interaction.

The plotted values are all *adjusted means* where, for example, the points for year are adjusted for the remaining effects. These are not the same as the marginal means for year, *ignoring* rather than *controlling for* the other predictors, because of both the unequal number of respondents per cell and the terms omitted from the model. The error bars again reflect variation in the adjusted means.

Assuming that the fitted model is appropriate for the data, the adjusted mean vocabulary score generally decreases modestly over the nearly 40 years covered by the GSS, from about 6.3 words to about 5.8 words. The effect of education is clear, with very large differences among educational groups and very small error estimates: The lowest-education group has an adjusted mean of about 4.5 words,

while the highest has an adjusted mean above 7.5 words. There is a small gender difference, with females about 0.15 words higher on average than males. The interaction between nativeBorn and ageGroup is more apparent, and somewhat different, using the additional predictors and all the data, rather than just 2016: The means for nativeborn respondents change less with ageGroup than the means for the nonnative born, and the gap between the native and nonnative born decreases at higher age levels. In comparing the several predictor effect plots in [Figure 4.15](#), it's important to notice that the vertical axes of the plots are scaled differently; if our primary purpose were comparisons among the graphs, then we could specify a common vertical axis by adding `y=list(lim=c(4, 8))` to the `axes` argument. (Reader: Try it to see why we didn't do it!).

#### 4.6.4 Interactions Between Numeric Predictors\*

Interactions between numeric predictors are inherently more complex than interactions between numeric predictors and factors, or interactions between factors. The simplest such model is the *linear-by-linear interaction model*; for the response variable  $y$  and numeric predictors  $x_1$  and  $x_2$ , the model takes the form

Other

$$(4.5) \quad E(y) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2$$

The regression coefficients in Equation 4.5 have the following interpretations:

- The intercept  $\beta_0$ , as usual, is the expected value of  $y$  when both  $x_1$  and  $x_2$  are zero. The intercept usually isn't of particular interest and is not directly interpretable when there are no data near  $x_1 = x_2 = 0$ .
- $\beta_1$  is the change in  $y$  associated with a one-unit increment in  $x_1$  when  $x_2 = 0$ , and  $\beta_2$  has a similar interpretation. These coefficients may or may not be directly interpretable, depending on whether the other  $x$  has values near zero.
- To see the interpretation of  $\beta_{12}$ , we differentiate  $E(y)$  with respect to each of  $x_1$  and  $x_2$ :

Other

$$\frac{\partial E(y)}{\partial x_1} = \beta_1 + \beta_{12}x_2$$

$$\frac{\partial E(y)}{\partial x_2} = \beta_2 + \beta_{12}x_1$$

Thus, for fixed values of  $x_2$ , the regression of  $y$  on  $x_1$  is linear, but with slope changing linearly with the value of  $x_2$  — and similarly for the relationship between  $y$  and  $x_2$  fixing  $x_1$ . This is why Equation 4.5 is called the linear-by-linear interaction model.

The linear-by-linear interaction model is represented in R by the simple formula  $y \sim x1*x2$ .<sup>38</sup>

<sup>38</sup> We'll see an application of a model of this form to a logistic regression in [Section 6.3.2](#).

There is an infinite variety of more complex models specifying interactions between numeric variables. One such model for two predictors is the *full quadratic model*

Other

$$(4.6) \quad E(y) = \beta_0 + \beta_{10}x_1 + \beta_{11}x_1^2 + \beta_{20}x_2 + \beta_{21}x_2^2 + \beta_{12}x_1x_2$$

The partial relationships in this model are quadratic at fixed values of the other predictor; for example, fixing :

Other

$$E(y|x_2 = x_2^*) = (\beta_0 + \beta_{20}x_2^* + \beta_{21}x_2^{*2}) + (\beta_{10} + \beta_{12}x_2^*)x_1 + \beta_{11}x_1^2$$

Differentiating Equation 4.6 with respect to each predictor, and rearranging, we see how the partial slope for each predictor varies both with its value and with the value of the other predictor:

Other

$$\frac{\partial E(y)}{\partial x_1} = \beta_{10} + (2\beta_{11} + \beta_{12}x_2)x_1$$

$$\frac{\partial E(y)}{\partial x_2} = \beta_{20} + (2\beta_{21} + \beta_{12}x_1)x_2$$

The full-quadratic model is fit in R using `poly()` with two predictors as arguments on the right-hand side of the model, as in

```
y ~ poly(x1, x2, degree=2, raw=TRUE)
```

We illustrate with the SLID data, regressing `log(wages)` on age and education. The full quadratic model does a reasonable job of summarizing the data:

```
mod.slid.fullquad <- lm (log (wages) ~  

poly (age, education, degree=2, raw=TRUE),  

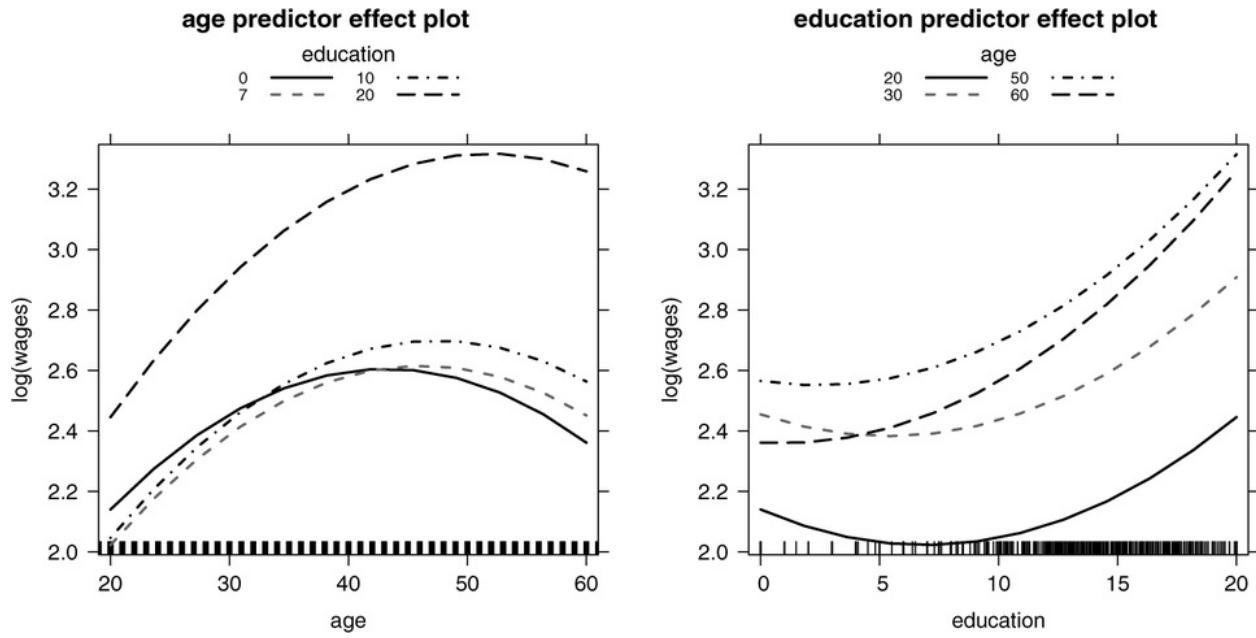
data=SLID, subset = age >= 18 & age <= 65)
```

Simply printing the estimated regression coefficients is uninformative in this problem. The predictor effect plots shown in [Figure 4.16](#) provide a much more understandable summary of the fitted model:

```
plot (predictorEffects (mod.slid.fullquad, xlevels=list (education=4,  

age=4)), multiline=TRUE)
```

**Figure 4.16** Predictor effect plots for the full quadratic model regressing  $\log_2(\text{wages})$  on age and education in the SLID data set.



We see that average wages increase with age up to about age 45, at which point wages begin to decline. This pattern holds for the lowest three education levels; at the highest education level, wages are generally much higher than at the three lower levels, and the age at which average wages attains its maximum level is somewhat higher—around age 50. The interaction between age and education is also summarized by the predictor effect plot for education, but the story told by this view of the model is less compelling.

## 4.7 More on Factors

### 4.7.1 Dummy Coding

As we have seen, a factor with  $q$  levels generates  $q-1$  regressors. For example, a factor with two levels requires only one regressor, usually coded zero for the first level and 1 for the second. This *dummy regressor* then represents the factor in the model. With more than two levels, the default method of defining dummy regressors in R is a generalization, in which the  $j$ th dummy regressor, for  $j = 1, \dots, q - 1$ , is equal to 1 for cases with the factor at level  $(j + 1)$  and zero otherwise, as was illustrated in [Section 4.5](#) for a factor named `z`. R uses the function `contr.treatment()` to create dummy regressors.

### 4.7.2 Other Factor Codings

The default method of coding factors is easy to use and intuitive; indeed, for most purposes, the method of coding a factor is irrelevant to the analysis. Nevertheless, R allows us to change how factors are coded.

How R codes regressors for factors is controlled by the "contrasts" option:

```
getOption("contrasts")
```

|                                |                           |
|--------------------------------|---------------------------|
| unordered                      | ordered                   |
| <code>"contr.treatment"</code> | <code>"contr.poly"</code> |

Two values are provided by the "contrasts" option, one for unordered factors and the other for ordered factors, which, as their name implies, have levels that are ordered from smallest to largest. Each entry corresponds to the name of a function that converts a factor into an appropriate set of contrasts; thus, the default functions for defining contrasts are `contr.treatment()` for unordered factors and `contr.poly()` for ordered factors.

We can see how the contrasts for a factor are coded by using the `contrasts()` function:

```
with(Prestige, contrasts(type))
```

|          |
|----------|
| wc prof  |
| bc 0 0   |
| wc 1 0   |
| prof 0 1 |

The result shows the dummy-coding scheme discussed in [Section 4.7.1](#). The first level of type is coded zero for both dummy regressors, the second level is coded 1 for the first and zero for the second regressor, and the third level is coded 1 for the second and zero for the first regressor.

It is possible to change the reference level for a factor using the `base` argument to `contr.treatment()`, as in

```

Prestige$type1 <- Prestige$type
contrasts(Prestige$type1) <-
  contr.treatment(levels(Prestige$type1), base=3)
contrasts(Prestige$type1)

      bc  wc
bc      1   0
wc      0   1
prof    0   0

```

We can also use the relevel () function to change the baseline level:

```
Prestige$type2 <- relevel(Prestige$type, ref="prof")
```

This command creates a new factor with "prof" as the baseline level but has the possibly undesirable side effect of reordering the levels of the factor, in this case making "prof" the *first* level:

```
contrasts(Prestige$type2)
```

|      | bc | wc |
|------|----|----|
| prof | 0  | 0  |
| bc   | 1  | 0  |
| wc   | 0  | 1  |

The function contr.SAS () is similar to contr.treatment (), except that it uses the *last* level as the baseline rather than the *first*. It is included in R to make many results match those from the SAS program:

```
contrasts(Prestige$type) <- "contr.SAS"  
contrasts(Prestige$type)
```

|      | bc | wc |
|------|----|----|
| bc   | 1  | 0  |
| wc   | 0  | 1  |
| prof | 0  | 0  |

R includes two additional functions for coding unordered factors, `contr.helmert()` and `contr.sum()`. The first of these uses *Helmert matrices* (which also can be used for ordered factors):

```
contrasts(Prestige$type) <- "contr.helmert"  
contrasts(Prestige$type)
```

|      | [,1] | [,2] |
|------|------|------|
| bc   | -1   | -1   |
| wc   | 1    | -1   |
| prof | 0    | 2    |

Each of the Helmert regressors compares one level beyond the first to the average of the preceding levels. If the number of cases at each level is equal, then the Helmert regressors are orthogonal (uncorrelated) in the model matrix, but they are not orthogonal in general. Helmert coding is currently rarely used, although it was the default method for coding factors in some older computer programs.

Yet another method of constructing contrasts for unordered factors, called *deviation coding*, is implemented in R by the `contr.sum()` function:

```

contrasts(Prestige$type) <- "contr.sum"
contrasts(Prestige$type)

[,1] [,2]
bc      1   0
wc      0   1
prof   -1  -1

```

Deviation coding derives from so-called *sigma* or *sum-to-zero* constraints on the parameters associated with the levels of a factor. For example, the coefficient for the last level of type, "prof", is implicitly constrained equal to the negative of the sum of the coefficients for the other levels, and the redundant last coefficient is omitted from the model. Each coefficient compares the corresponding level of the factor to the average of all the levels. This coding is consistent with the treatment of factors in ANOVA models in many statistical textbooks and produces orthogonal regressors for different terms in the model in “balanced” data (i.e., data with equal cell counts).<sup>39</sup>

<sup>39</sup> Although in unbalanced data the regressors for different factors computed by contr.sum () and contr.helmert () are correlated in the model matrix, they may nevertheless be used to compute correct Type III tests, because the sets of regressors are orthogonal in the row basis of the model matrix (see [Section 5.3.4](#)), which has one row for each cell.

The **car** package includes the functions contr.Treatment () and contr. Sum (). These behave similarly to the standard contr.treatment () and contr. sum () functions but produce more descriptive contrast names:

```

contrasts(Prestige$type) <- "contr.Treatment"
contrasts(Prestige$type)

[T.wc] [T.prof]
bc      0      0
wc      1      0
prof    0      1

contrasts(Prestige$type) <- "contr.Sum"
contrasts(Prestige$type)

[S.bc] [S.wc]
bc      1      0
wc      0      1
prof   -1     -1
```

An alternative to changing the contrasts of individual factors is to reset the global contrasts option; for example:

```
options(contrasts=c("contr.Sum", "contr.poly"))
```

These contrast-generating functions would then be used by default for all subsequent model fits. In setting the contrasts option, we must specify contrasts for both unordered and ordered factors. Contrasts can also be set as an argument to the lm () function (see [Section 4.9.8](#)).

### 4.7.3 Ordered Factors and Orthogonal-Polynomial Contrasts

R has a built-in facility for working with factors that have ordered levels. The default contrast-generating function for ordered factors is contr.poly (), which creates *orthogonal-polynomial contrasts*. Suppose, for example, that we treat the variable type (type of occupation) in the Duncan data set as an ordered factor:

```

Duncan$type.ord <- ordered(Duncan$type,
  levels=c("bc", "wc", "prof"))
Duncan$type.ord
[1] prof prof prof prof prof prof prof prof wc   prof prof prof
[13] prof prof prof wc    prof prof prof prof wc   wc   wc   wc
[25] bc   bc
[37] bc   bc
Levels: bc < wc < prof

round(contrasts(Duncan$type.ord), 3)
      .L     .Q
[1,] -0.707 0.408
[2,]  0.000 -0.816
[3,]  0.707 0.408

```

The ordered () function creates an ordered factor, just as factor () creates an unordered factor. The column labeled .L in the contrast matrix represents a linear contrast and that labeled .Q, a quadratic contrast. The two contrasts are orthogonal when there are equal numbers of cases at the different levels of the factor but not in general.<sup>40</sup> This approach is most appropriate when the categories are, in some sense, equally spaced, but in certain fields, it is traditional to use orthogonal-polynomial contrasts more generally for ordinal predictors.

<sup>40</sup> Because the contrasts are orthogonal in the row basis of the model matrix they will produce correct Type III tests, even for unbalanced data in two-and higher-way ANOVA (see [Section 5.3.4](#)).

Applied to a one-way ANOVA model for prestige in the Duncan data, we get the following result:

```
S(lm(prestige ~ type.ord, data=Duncan))
```

```
Call: lm(formula = prestige ~ type.ord, data = Duncan)
```

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 46.62    | 2.75       | 16.95   | < 2e-16  |
| type.ord.L  | 40.79    | 3.61       | 11.31   | 2.5e-14  |
| type.ord.Q  | 12.20    | 5.69       | 2.14    | 0.038    |

Residual standard deviation: 15.9 on 42 degrees of freedom  
Multiple R-squared: 0.757  
F-statistic: 65.6 on 2 and 42 DF, p-value: 1.21e-13  
AIC BIC  
381.48 388.71

```
Tapply(prestige ~ type.ord, mean, data=Duncan)
```

|  | bc     | wc     | prof   |
|--|--------|--------|--------|
|  | 22.762 | 36.667 | 80.444 |

The coefficients for type.ord represent linear and quadratic trends across the levels of the factor. Looking at the *t*-statistics and the associated *p*-values, there is very strong evidence for the linear component of the effect of type of occupation (*p* ≈ 0) and weaker evidence for the quadratic component (*p* = .038).

When there is a discrete numeric (as opposed to qualitative or ordinal) predictor, we suggest using the poly () function to generate *orthogonal-polynomial regressors* (discussed more generally in [Section 4.4.1](#)). To construct a simple example, suppose that we have a numeric variable representing the dosage of a drug in a clinical trial:

```
dosage <- c(1, 4, 8, 1, 4, 8, 1, 4)
```

Because dosage has only three distinct values—1, 4, and 8—we could treat it as a factor or as an ordered factor, coding two regressors to represent it in a linear model. It wouldn't be appropriate to use contr.poly () here to generate the regressors because the levels of dosage aren't equally spaced; moreover, because the data are unbalanced, the regressors created by contr.poly () would be

correlated. The `poly()` function (discussed in [Section 4.4.1](#)) will generate orthogonal-polynomial regressors for dosage:<sup>41</sup>

```
X <- poly(dosage, degree=2)
round(X, 3)
```

|       | 1      | 2      |
|-------|--------|--------|
| [1, ] | -0.375 | 0.261  |
| [2, ] | 0.016  | -0.456 |
| [3, ] | 0.538  | 0.293  |
| [4, ] | -0.375 | 0.261  |
| [5, ] | 0.016  | -0.456 |
| [6, ] | 0.538  | 0.293  |
| [7, ] | -0.375 | 0.261  |
| [8, ] | 0.016  | -0.456 |

```
attr(, "coefs")
attr(, "coefs")$alpha
[1] 3.8750 5.0486
```

```
attr(, "coefs")$norm2
[1] 1.000 8.000 58.875 269.656
```

```
attr(, "degree")
[1] 1 2
attr(, "class")
[1] "poly"    "matrix"

zapsmall(cor(X))

 1 2
1 1 0
2 0 1
```

[41](#) As we noted previously for the `model.matrix()` function, you can safely ignore the attributes attached to the object returned by `poly()`.

The second argument to `poly()` specifies the degree of the polynomial, here `degree=2`, a quadratic, one degree fewer than the number of distinct values of dosage. The first column of `X` represents a linear trend in dosage, while the second column represents a quadratic trend net of the linear trend. We used the `zapsmall()` function to round very small numbers to zero, showing that the correlation between the linear and quadratic regressors, computed by the `cor()` function, is zero within rounding error.<sup>[42](#)</sup>

[42](#) As a general matter, one can't rely on computer arithmetic involving nonintegers to be exact, and so here the correlation between the linear and quadratic components is computed to be a very small number rather than precisely zero.

The `poly()` function can also be used to generate raw, as opposed to orthogonal, polynomials, as explained in [Section 4.4.1](#):

```

(X2 <- poly(dosage, degree=2, raw=TRUE) )

      1   2
[1,] 1   1
[2,] 4 16
[3,] 8 64
[4,] 1   1
[5,] 4 16
[6,] 8 64
[7,] 1   1
[8,] 4 16
attr(,"degree")
[1] 1 2
attr(,"class")
[1] "poly"    "matrix"

round(cor(X2), 3)

      1       2
1 1.000 0.972
2 0.972 1.000

```

The first column is just dosage and the second the square of dosage. Used in a linear model, the two specifications—orthogonal and raw polynomials—are equivalent in the sense that they produce the same fit to the data.

Before proceeding, we return the contrasts option to its default value:

```
options (contrasts=c ("contr.treatment", "contr.poly"))
```

#### 4.7.4 User-Specified Contrasts\*

In some instances, there are  $q - 1$  specific comparisons among the  $q$  levels of a factor that are of interest, and these comparisons need not correspond to the contrasts constructed by any of the standard contrast-generating functions. The Baumann data set (introduced on page 201) is from a randomized experiment in which children were taught reading by one of three methods, a standard method, "Basal", and two new methods, "DTRA" and "Strat". It is natural here to test for differences between the standard method and the new methods, and between the two new methods. These tests can be formulated in R with user-defined contrasts.

Here is the general idea: Suppose that the vector  $\mu$  represents the population factor-level means in a one-way ANOVA or the raveled factor-combination means for a two-way or higher classification. If there are  $q$  means, then there are  $q - 1$  *df* for the differences among them. Let the *contrast matrix*  $\mathbf{C}$  be a  $q \times (q - 1)$  matrix of rank  $q - 1$ , each of the columns of which sums to zero. Then,

Other

$$\mu = [1, \mathbf{C}] \begin{bmatrix} \alpha \\ \gamma \end{bmatrix}$$

is a linear model for the factor-level means. In this equation,  $1$  is a  $q \times 1$  vector of ones,  $\alpha$  is the *general mean* of the response variable  $y$ , and  $\gamma$  contains  $q - 1$  parameters for the  $q - 1$  contrasts specified by the columns of  $\mathbf{C}$ . The trick is to formulate  $\mathbf{C}$  so that the  $(q - 1) \times 1$  parameter vector  $\gamma$  represents interesting contrasts among the level means. Because  $[1, \mathbf{C}]$  is nonsingular (*Reader:* why?), we can solve for the parameters as a linear transformation of the means:

Other

$$\begin{bmatrix} \alpha \\ \gamma \end{bmatrix} = [1, \mathbf{C}]^{-1} \mu$$

A very simple way to proceed (though not the only way) is to make the columns of  $\mathbf{C}$  mutually orthogonal. Then, the rows of  $[1, \mathbf{C}]^{-1}$  will be proportional to the corresponding columns of  $[1, \mathbf{C}]$ , and we can directly code the contrasts of

interest among the means in the columns of  $\mathbf{C}$ .

None of this requires that the factor have equal numbers of cases at its several levels, but if these counts are equal, as in the Baumann data set, then not only are the columns of  $\mathbf{C}$  orthogonal, but the columns of the model matrix  $\mathbf{X}$  constructed from  $\mathbf{C}$  are orthogonal as well. Under these circumstances, we can partition the regression sum of squares for the model into 1-*df* components due to each contrast.

For the Baumann data the two contrasts of interest are (1) "Basal" versus the average of "DRTA" and "Strat" and (2) "DRTA" versus "Strat".<sup>43</sup>

```
C <- matrix(c(1, -0.5, -0.5, 0, 1, -1), 3, 2)
colnames(C) <- c(":Basal vs. DRTA & Strat", ":DRTA vs. Strat")
rownames(C) <- c("Basal", "DRTA", "Strat")
C
```

|       | :Basal vs. DRTA & Strat | :DRTA vs. Strat |
|-------|-------------------------|-----------------|
| Basal | 1.0                     | 0               |
| DRTA  | -0.5                    | 1               |
| Strat | -0.5                    | -1              |

[43](#) Setting the row names of the contrast matrix isn't necessary here, but we do it for clarity.

The columns of  $\mathbf{C}$  correspond to these two comparisons, which we set as the contrasts for the factor group and refit the model:

```

contrasts(Baumann$group) <- C
S(lm(post.test.3 ~ group, data=Baumann))

Call: lm(formula = post.test.3 ~ group, data = Baumann)

```

Coefficients:

|                              | Estimate | Std. Error | t value |
|------------------------------|----------|------------|---------|
| (Intercept)                  | 44.015   | 0.777      | 56.63   |
| group:Basal vs. DRTA & Strat | -2.970   | 1.099      | -2.70   |
| group:DRTA vs. Strat         | 1.227    | 0.952      | 1.29    |
|                              | Pr(> t ) |            |         |
| (Intercept)                  | <2e-16   |            |         |
| group:Basal vs. DRTA & Strat | 0.0088   |            |         |
| group:DRTA vs. Strat         | 0.2020   |            |         |

Residual standard deviation: 6.31 on 63 degrees of freedom

Multiple R-squared: 0.125

F-statistic: 4.48 on 2 and 63 DF, p-value: 0.0152

| AIC    | BIC    |
|--------|--------|
| 435.48 | 444.24 |

The  $t$ -statistics for the contrasts test the two hypotheses of interest, and so we have strong evidence that the new methods are superior to the old ( $p = .0088$ ), but little evidence of a difference in efficacy between the two new methods ( $p = .20$ ). The overall  $F$ -test and other similar summaries are unaffected by the choice of contrasts as long as we use  $q - 1$  of them and the  $\mathbf{C}$  matrix has full column rank  $q - 1$ .

User-specified contrasts may also be used for factors in more complex linear models, including multifactor models with interactions.

## 4.7.5 Suppressing the Intercept in a Model With Factors\*

The rule that a factor with  $q$  levels requires  $q - 1$  regressors generally does not hold if the model formula excludes the intercept:

```

S(prestige.mod.4 <- update(prestige.mod.1, . ~ . - 1))

Call: lm(formula = prestige ~ education + log2(income) + type -
1, data = Prestige)

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
education      3.284     0.608    5.40  5.1e-07
log2(income)   7.269     1.190    6.11  2.3e-08
typebc        -81.202    13.743   -5.91  5.6e-08
typewc         -82.641    13.788   -5.99  3.9e-08
typeprof      -74.451    15.118   -4.92  3.7e-06
Residual standard deviation: 6.64 on 93 degrees of freedom
(4 observations deleted due to missingness)

Multiple R-squared: 0.983
F-statistic: 1.11e+03 on 5 and 93 DF, p-value: <2e-16

AIC      BIC
655.93 671.44

```

The primary difference between this fit and prestige.mod.1 (page 205) is that there is now a separate intercept for each level of type. Summaries such as  $df$ ,  $\sigma$ , and the estimates of the other coefficients and their standard errors are all identical.<sup>44</sup> The coefficients of the now three dummy regressors for type are the intercepts for the three groups, and in some instances, this can provide a simpler description of the model. The usefulness of tests for this parametrization is questionable.<sup>45</sup> We suggest that you generally avoid leaving off the intercept in a linear model unless you have a specific reason for doing so and then are very careful to interpret the coefficients and the results of statistical tests correctly.

<sup>44</sup> The  $R^2$ 's for the two models differ. When the intercept is suppressed, `lm()` calculates  $R^2$  based on variation around zero rather than around the mean of the response, producing a statistic that does not generally have a sensible interpretation.

<sup>45</sup> For example, the Type II  $F$ -test for type produced by `Anova(prestige.mod.4)` has 3 rather than 2  $df$ , and tests the probably uninteresting hypothesis that all three intercepts are equal to *zero* rather than the more interesting hypothesis that they are all equal to *each other*. The `Anova()` function in the `car` package is

discussed in [Section 5.3.4](#).

## 4.8 Too Many Regressors\*

In some instances, we may try to fit models with regressors that are exact linear combinations of each other. This can happen by accident, for example, by including scores on subtests (e.g.,  $x_1$ ,  $x_2$ , and  $x_3$ ) and the total score of all the subtests ( $x_4 = x_1 + x_2 + x_3$ ) as regressors in the same linear model. If you fit a model like  $y \sim x_1 + x_2 + x_3 + x_4$  using `lm()` or many other R statistical-modeling functions, the function will add the regressors in the order in which they appear in the formula, discover that  $x_4$  is a linear combination of the previous regressors, and effectively drop it from the formula. Using the formula  $y \sim x_4 + x_3 + x_2 + x_1$  produces the same fitted values but different regressors and regression coefficients because  $x_1$  is now dropped, as it is specified last.

To take another example, suppose that you have two variables,  $x_5$  and  $x_6$ , that are rounded heights of children, with  $x_5$  in centimeters and  $x_6$  in inches. If you mistakenly fit the model

***lm (y ~ x5 + x6 + others...)***

nonsensical coefficients for both  $x_5$  and  $x_6$  are likely to be produced because although  $x_5$  and  $x_6$  are in principle identical except for units of measurement, rounding makes them slightly less than perfectly correlated. Errors of this sort are not automatically detected by `lm()`.

Too many regressors may also occur when fitting models with factors and interactions. To illustrate, we use the nutrition data set in the **emmeans** package, which we loaded previously in the chapter:

### **summary(nutrition)**

| age  | group         | race        | gain          |
|------|---------------|-------------|---------------|
| 1: 7 | FoodStamps:60 | Black :21   | Min. :-20.00  |
| 2:23 | NoAid :47     | Hispanic: 3 | 1st Qu.: 0.00 |
| 3:64 |               | White :83   | Median : 3.00 |
| 4:13 |               |             | Mean : 2.29   |
|      |               |             | 3rd Qu.: 6.00 |
|      |               |             | Max. : 15.00  |

These data, taken from Milliken and Johnson (2004), are the results of an observational study of nutrition education. Low-income mothers who enrolled in a nutrition knowledge program were classified by age group, with four levels, and race, with levels "Black", "Hispanic", and "White".<sup>46</sup> The response is gain, the change in score on a nutrition test, after the program minus before the program. We begin by looking at the number of mothers in each of the  $4 \times 3 = 12$  cells:

### ***xtabs(~ race + age, data=nutrition)***

|          | age |    |    |    |
|----------|-----|----|----|----|
| race     | 1   | 2  | 3  | 4  |
| Black    | 2   | 7  | 10 | 2  |
| Hispanic | 0   | 0  | 3  | 0  |
| White    | 5   | 16 | 51 | 11 |

<sup>46</sup> To simplify the presentation here, a third factor group, whether or not the mother's family gets Food Stamps, is not included.

Three of the four cells for "Hispanic" mothers are empty. We have seen that least-squares regression copes well with unequal sample sizes in the cells, but empty cells are another matter entirely.

Suppose that we fit a linear regression model with main effects and the two-factor interaction:

```
mod.nut.1 <- lm (gain ~ race*age, data=nutrition)
```

This model has one parameter for the intercept,  $3 + 2 = 5$  parameters for the main effects, and  $3 \times 2 = 6$  parameters for the two-factor interaction, for a total of 12 parameters. Thus, the model fits 12 parameters to only nine cell means, and consequently three of the parameters cannot be estimated:

```
S(mod.nut.1, brief=TRUE)
```

Coefficients: (3 not defined because of singularities)

|                   | Estimate | Std. Error | t value | Pr(> t ) |
|-------------------|----------|------------|---------|----------|
| (Intercept)       | 4.000    | 4.195      | 0.95    | 0.34     |
| raceHispanic      | 0.867    | 3.906      | 0.22    | 0.82     |
| raceWhite         | -5.200   | 4.964      | -1.05   | 0.30     |
| age2              | -4.714   | 4.757      | -0.99   | 0.32     |
| age3              | -3.200   | 4.596      | -0.70   | 0.49     |
| age4              | -5.500   | 5.933      | -0.93   | 0.36     |
| raceHispanic:age2 | NA       | NA         | NA      | NA       |
| raceWhite:age2    | 8.039    | 5.645      | 1.42    | 0.16     |
| raceHispanic:age3 | NA       | NA         | NA      | NA       |
| raceWhite:age3    | 7.616    | 5.371      | 1.42    | 0.16     |
| raceHispanic:age4 | NA       | NA         | NA      | NA       |
| raceWhite:age4    | 10.336   | 6.741      | 1.53    | 0.13     |

Residual standard deviation: 5.93 on 98 degrees of freedom

Multiple R-squared: 0.0666

F-statistic: 0.874 on 8 and 98 DF, p-value: 0.541

| AIC    | BIC    |
|--------|--------|
| 695.29 | 722.02 |

The lm () function fits the intercept first, then all the main effects, and finally the regressors for the two-factor interaction. All the interactions with level "Hispanic" of race are not estimated, indicated by the NAs in the coefficient table. The term *singularities* is used to describe perfectly collinear regressors, and the corresponding coefficients, which are not estimated, are said to be *aliased*.

The Effect () function in the **effects** package can be used to print the estimated cell means:

```
Effect(c("race", "age"), mod.nut.1)
```

```
race*age effect
          age
race      1       2       3       4
Black     4.0 -0.71429 0.8000 -1.5000
Hispanic   NA        NA 1.6667      NA
White    -1.2  2.12500 3.2157  3.6364
```

Adjusted means for the cells with no data are not estimable, as indicated by the NAs.

Without collecting additional data, there are at least three reasonable, but not completely satisfactory, approaches to analyzing these data. First, the level "Hispanic" could be dropped from the analysis, limiting the study to the two other races. Second, data could be restricted to the third age group, ignoring the data for other ages. Third, we can fit a model without the interactions:

```
(mod.nut.2 <- update(mod.nut.1, ~ . - race:age))
```

Call:

```
lm(formula = gain ~ race + age, data = nutrition)
```

Coefficients:

| (Intercept) | raceHispanic | raceWhite | age2  |
|-------------|--------------|-----------|-------|
| -1.256      | 0.651        | 2.158     | 1.015 |

```

      age3          age4
      2.271        2.276

Effect(c("race", "age"), mod.nut.2)

race*age effect
      age
race           1       2       3       4
Black     -1.25575 -0.24038 1.0154 1.0201
Hispanic -0.60448  0.41088 1.6667 1.6714
White      0.90230  1.91767 3.1735 3.1782

```

The additive model fits only six parameters, for the intercept and the main effects, to the nine cells with data, and hence all of the parameters and all of the adjusted cell means are estimable. This approach is sensible only if the interaction between the two factors is actually zero or negligibly small.

## 4.9 The Arguments of the lm () Function

The lm () function has several additional useful arguments beyond those that we have encountered, and some of the arguments that we discussed previously have uses that were not mentioned. The args () function reports the arguments to lm ():

**args ("lm")**

```
function (formula, data, subset, weights, na.action, method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset, ...)  
NULL
```

We will describe each of these arguments in turn.<sup>47</sup>

<sup>47</sup> Also see help ("lm").

### 4.9.1 formula

A formula for `lm()` consists of a left-hand side, specifying the response variable, and a right-hand side, specifying the terms in the model; the two sides of the formula are separated by a tilde (`~`). We read the formula `a ~ b` as “`a` is modeled as `b`,” or “`a` is regressed on `b`.”

The left-hand side of the formula can be any valid R expression that evaluates to a numeric vector of the appropriate length. The arithmetic operators, `+`, `-`, `*`, `/`, and `^`, have their usual meanings on the left side of the formula, and we can call whatever functions are appropriate to our purpose. For example, with reference to the Guyer data set (introduced in [Chapter 2](#)), we could replace the number of cooperative responses by the percentage of cooperative responses,

**`lm (100*cooperation/120 ~ condition*sex, data=Guyer)`**

or (using the `logit()` function in the `car` package) by the log-odds of cooperation,

**`lm (logit (cooperation/120) ~ condition*sex, data=Guyer)`**

The right-hand side of the model formula may include factors and expressions that evaluate to numeric vectors and matrices. Character and logical variables are treated like factors: The unique values of a character variable are equivalent to factor levels, although character values are always ordered alphabetically; likewise, the values of a logical variable are also equivalent to factor levels and are ordered `FALSE`, `TRUE`.

Because the arithmetic operators have special meaning on the right-hand side of model formulas, arithmetic expressions that use these operators have to be *protected* inside function calls: For example, the `+` in the term `log (salary + interest + dividends)` has its usual arithmetic meaning on the right-hand side of a model formula, even though `salary + interest + dividends` does not when it is unprotected.

The *identity or inhibit function* `I()`, which returns its argument unaltered, may be used to protect arithmetic expressions in model formulas. For example, to

regress prestige on the sum of education and income in Duncan's data set, thus implicitly forcing the coefficients of these two predictors to be equal, we may write,

```
(lm(prestige ~ I(income + education), data=Duncan))
```

Call:

```
lm(formula = prestige ~ I(income + education), data = Duncan)
```

Coefficients:

|  | (Intercept) | I(income + education) |
|--|-------------|-----------------------|
|  | -6.063      | 0.569                 |

estimating a single coefficient for the sum.

We have already described many of the special operators that appear on the right side of linear-model formulas. In [Table 4.1](#), A and B represent elements in a linear model: numeric vectors, matrices, factors (or logical or character variables), or expressions (such as  $a + b$  or  $a * b$ ) composed from these.<sup>48</sup>

**Table 4.1**

| Expression | Interpretation                      | Example             |
|------------|-------------------------------------|---------------------|
| A + B      | include both A and B                | income + education  |
| A - B      | exclude B from A                    | a*b*c - a:b:c       |
| A:B        | all interactions of A and B         | type:education      |
| A*B        | A crossed with B, i.e., A + B + A:B | type*education      |
| B %in% A   | B nested within A                   | education %in% type |
| A/B        | A + B %in% A                        | type/education      |
| A^k        | all effects crossed up to order k   | (a + b + c)^2       |

<sup>48</sup> Cf. the table of operators in formulas given in Chambers (1992, p. 29).

The last three operators in [Table 4.1](#), %in%, /, and ^, are new to us:

- %in% is actually a synonym for: but is typically used differently, to create nested terms rather than for interactions.
- / is a shorthand for nesting, in the same sense as \* is a shorthand for

crossing. As the table indicates, A/B is equivalent to A + B %in% A and hence to A + B:A. To see how this works, consider the following contrived example:<sup>49</sup>

```
(f <- factor(rep(LETTERS[1:3], each=3)))
```

```
[1] A A A B B B C C C  
Levels: A B C
```

```
(x <- rep(1:3, 3))
```

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

```
model.matrix(~ f/x)
```

|   | (Intercept) | fB | fC | fA:x | fB:x | fC:x |
|---|-------------|----|----|------|------|------|
| 1 | 1           | 0  | 0  | 1    | 0    | 0    |
| 2 | 1           | 0  | 0  | 2    | 0    | 0    |
| 3 | 1           | 0  | 0  | 3    | 0    | 0    |
| 4 | 1           | 1  | 0  | 0    | 1    | 0    |
| 5 | 1           | 1  | 0  | 0    | 2    | 0    |
| 6 | 1           | 1  | 0  | 0    | 3    | 0    |
| 7 | 1           | 0  | 1  | 0    | 0    | 1    |
| 8 | 1           | 0  | 1  | 0    | 0    | 2    |
| 9 | 1           | 0  | 1  | 0    | 0    | 3    |

```
attr(),"assign")
```

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

```
attr(),"contrasts")
```

```
attr(),"contrasts")$f
```

```
[1] "contr.treatment"
```

```
model.matrix(~ x:f)
```

|   | (Intercept) | x:fA | x:fB | x:fC |
|---|-------------|------|------|------|
| 1 | 1           | 1    | 0    | 0    |
| 2 | 1           | 2    | 0    | 0    |

```

3           1   3   0   0
4           1   0   1   0
5           1   0   2   0
6           1   0   3   0
7           1   0   0   1
8           1   0   0   2
9           1   0   0   3

attr("assign")
[1] 0 1 1 1
attr("contrasts")
attr("contrasts")$f
[1] "contr.treatment"

```

[49](#) Recall that the attributes attached to the model matrix can be ignored.

Thus,  $f/x$  fits an intercept, two dummy regressors for  $a$ , with  $f = "A"$  as the baseline level, and a separate slope for  $x$  within each of the levels of  $f$ . In contrast,  $x:f$ , or equivalently  $x \%in% f$  (or, indeed,  $f:x$  or  $f \%in% x$ ), fits a common intercept and a separate slope for  $x$  within each level of  $f$ .

- The  $\wedge$  operator builds crossed effects up to the order given in the exponent. Thus, the example in the table,  $(a + b + c)^{\wedge}2$ , expands to all main effects and pairwise interactions among  $a$ ,  $b$ , and  $c$ : that is,  $a + b + c + a:b + a:c + b:c$ . This is equivalent to another example in the table,  $a*b*c - a:b:c$ .

The intercept, represented by 1 in model formulas, is included in the model unless it is explicitly excluded by specifying -1 or 0 in the formula.

## 4.9.2 data

The data argument ordinarily supplies a data frame containing the variables that appear in the model. If the data argument is omitted, then data are retrieved from the global environment, and so objects will be found in the normal manner along the search path, such as in an attached data frame. Explicitly setting the data

argument is a sound general practice (as explained in [Section 2.3.1](#)). An additional advantage of using the `data` argument is that if a data frame has row names, these then conveniently become the case names of quantities derived from the fitted model, such as residuals and fitted values, and can be used, for example, to label points in plots.

### 4.9.3 subset

The `subset` argument is used to fit a model to a subset of cases. Several forms are possible:

- A logical vector, as in

```
lm (weight ~ repwt, data=Davis, subset = sex == "F") # fit only to women
```

where `sex == "F"` returns a vector of TRUEs and FALSEs.

- A numeric vector of case indices to include

```
lm (weight ~ repwt, data=Davis, subset=c (1:99, 141)) # use only  
cases 1 to 99 and 141
```

- A numeric vector with negative entries, indicating the cases to be *omitted* from the fitted model

```
lm (prestige ~ income + education, data=Duncan, subset = -c (6, 16))  
# exclude cases 6 and 16
```

- A character vector containing the row names of the cases to be included, an option for which we cannot provide a compelling example.

### 4.9.4 weights

If we loosen the constant-variance assumption in Equation 4.2 (page 175) to

Other

$$\text{Var}(y|x_1, \dots, x_k) = \sigma^2/w$$

for known weights  $w > 0$ , then we have a weighted-least-squares problem. The `weights` argument takes a numeric vector of nonnegative values of length equal to the number of cases and produces a weighted-least-squares fit.

### 4.9.5 `na.action`

The "na.action" option, set by options (`na.action="function-name"`), is initially set to "na.omit", employing the `na.omit()` function to delete cases with missing values for any of the variables appearing in the model, thus producing a *complete-case analysis*.

The function `na.exclude()` is similar to `na.omit()`, but it saves information about the deleted cases. This information may be used by functions that perform computations on linear-model objects. When quantities such as residuals are calculated, `na.exclude()` causes entries corresponding to cases with missing data to be NA, rather than simply absent from the result. Filling out results with NAs can be advantageous because it preserves the number of cases in the data set; for example, when plotting residuals against a predictor, we need do nothing special to ensure that both variables have the same number of entries. We made sure that the functions in the `car` package are compatible with `na.exclude()`, and we encourage its use in preference to `na.omit()`.

A third `na.action` option is `na.fail()`, which stops all calculations and reports an error if any missing data are encountered. Although we will not pursue the possibility here, you can handle missing data in other ways by writing your own missing-data function. There are also other strategies for dealing with missing data in R.<sup>50</sup>

<sup>50</sup> For example, an online appendix to the *R Companion* discusses multiple imputation of missing data.

Because we usually fit more than one model to the data, it is generally advantageous to handle missing data *outside* of `lm()`, to ensure that all models are fit to the same subset of valid cases (see [Section 2.3.2](#)). To do otherwise is to invite inconsistency. This is true, incidentally, not only in R but also in other statistical software.

## 4.9.6 method, model, x, y, qr\*

These are technical arguments, relating to how the computations are performed and what information is stored in the returned linear-model object; see `?lm`.

## 4.9.7 singular.ok\*

R builds a full-rank model matrix by removing redundant dummy regressors. Under some circumstances, such as perfect collinearity, or when there is an empty cell in an ANOVA, the model matrix may be of deficient rank, and not all the coefficients in the linear model will be estimable. If `singular.ok` is `TRUE`, which is the default, then R will fit the model anyway, setting the aliased coefficients of the redundant regressors to NA.<sup>51</sup>

[51](#) Which regressors are identified as redundant is generally partly arbitrary; see [Section 4.8](#).

## 4.9.8 contrasts

This argument allows us to specify contrasts for factors in a linear model, in the form of a list with named elements; for example:

```
mod.vocab.1 <- lm (vocab ~ nativeBorn*ageGroup, data=GSS16,  
contrasts=list (nativeBorn=contr.Sum, ageGroup=contr.poly))
```

The elements of the list may be contrast-generating functions (as in the example above); the quoted names of contrast-generating functions, for example,

```
mod.vocab.1 <- lm (vocab ~ nativeBorn*ageGroup, data=GSS16,  
contrasts=list (nativeBorn="contr.Sum", ageGroup="contr.poly"))
```

or matrices whose columns define the contrasts, as in

```
C <- matrix (c (1, -0.5, -0.5, 0, 1, -1), 3, 2)
```

```
colnames (C) <- c ("Basal vs. DRTA & Strat", "DRTA vs. Strat")
```

```
lm (post.test.3 ~ group, data=Baumann, contrasts=list (group=C))
```

revisiting the Baumann one-way ANOVA (cf. page 230).

### 4.9.9 offset

An *offset* is a term added to the right-hand side of a model with no associated parameter to be estimated—it implicitly has a fixed coefficient of 1. In a linear model, specifying a variable as an offset is equivalent to *subtracting* the variable from the response—for example, `lm (y ~ x, offset=z)` is equivalent to `lm (y - z ~ x)`. An alternative, also producing the same fit to the data, is to use the `offset()` function directly on the right-hand side of the model formula: `lm (y ~ x + offset (z))`. Offsets are more useful in generalized linear models (discussed in [Chapter 6](#)) than in linear models.

## 4.10 Complementary Reading and References

- The statistical background for the material in this chapter is covered in detail in Fox (2016, Part II) and in Weisberg (2014, chaps. 2–6).
- The original descriptions of model formulas and the implementation of linear models in S (the precursor to R) appeared, respectively, in Chambers and Hastie (1992a) and Chambers (1992) (two chapters in *Statistical Models in S*, edited by Chambers & Hastie, 1992b). This material is still worth reading.

# 5 Coefficient Standard Errors, Confidence Intervals, and Hypothesis Tests

John Fox & Sanford Weisberg

In earlier chapters of the *R Companion*, we assumed general familiarity with standard errors, confidence intervals, and hypotheses tests, all of which are introduced in most basic statistics courses. In this chapter, we develop these three related topics in greater depth. Variances are parameters that describe variation both in measured variables and in statistics computed from observed data. The variance of a statistic is called its *sampling variance*, and an estimate of the square root of the sampling variance, the estimated standard deviation of a statistic, is called its *standard error*. The standard error of a statistic is therefore to be distinguished from the sample standard deviation of a variable. *Confidence statements* express uncertainty about point estimates of parameters by providing plausible ranges of values for the parameters, either individually (*confidence intervals*) or in combination (*confidence regions*). Hypothesis tests frame the summary of an analysis in terms of decisions or *p*-values.

The **car** package includes functions that either provide important functionality missing from the basic R distribution, or in some cases, that are meant to replace standard R functions. This chapter concentrates on applications to linear models, but the ideas developed in the chapter are far more general, and many of the functions described here apply as well to the generalized linear models discussed in [Chapter 6](#) and the mixed-effects models taken up in [Chapter 7](#), as well as to other kinds of regression models, some of which are described in the online appendices to the *R Companion*.

- [Section 5.1](#) shows how to compute coefficient standard errors, both conventionally and in several nonstandard ways, including standard errors that take nonconstant error variance into account, bootstrapped standard errors computed nonparametrically by resampling from the observed data, and standard errors calculated by the delta method for nonlinear functions of regression coefficients.
- [Section 5.2](#) shows how to construct various kinds of confidence intervals and confidence regions for regression coefficients.
- Finally, [Section 5.3](#) explains how to formulate a variety of hypothesis tests for regression coefficients, including a discussion of analysis-of-variance tables and general linear hypotheses.

Collectively, the methods described in this chapter provide powerful and general tools for statistical inference in regression analysis.

## 5.1 Coefficient Standard Errors

### 5.1.1 Conventional Standard Errors of Least-Squares Regression Coefficients

For an example in this section, we return to the Transact data set introduced in [Section 4.2.3](#), loading the **car** package, which also loads the **carData** package in which the Transact data reside:

```
library("car")
summary(Transact)
```

|          | t1   | t2           | time          |
|----------|------|--------------|---------------|
| Min. :   | 0    | Min. : 148   | Min. : 487    |
| 1st Qu.: | 85   | 1st Qu.:1516 | 1st Qu.: 3618 |
| Median : | 214  | Median :2192 | Median : 5583 |
| Mean :   | 281  | Mean :2422   | Mean : 6607   |
| 3rd Qu.: | 437  | 3rd Qu.:3175 | 3rd Qu.: 8712 |
| Max. :   | 1450 | Max. :5791   | Max. :20741   |

The Transact data set includes two predictors, t1 and t2, the numbers of transactions of two types in  $n = 261$  bank branches, along with the response variable time, the minutes of labor performed in each branch. The regressors are simply the predictors, and so the regression model takes the form  $\text{time} \sim t1 + t2$ , with three regression coefficients, an intercept and slopes for t1 and t2:

```
S(trans.1 <- lm(time ~ t1 + t2, Transact), brief=TRUE)
```

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 144.3694 | 170.5441   | 0.85    | 0.4      |
| t1          | 5.4621   | 0.4333     | 12.61   | <2e-16   |
| t2          | 2.0345   | 0.0943     | 21.57   | <2e-16   |

Residual standard deviation: 1140 on 258 degrees of freedom  
Multiple R-squared: 0.909

```
F-statistic: 1.29e+03 on 2 and 258 DF, p-value: <2e-16
```

```
  AIC      BIC
```

```
4421.1 4435.3
```

As a reminder, the `S()` function in the `car` package is a replacement for the standard R `summary()` function for linear and some other statistical models. The default printed output produced by `S()` is shorter than the default for `summary()` and is further abbreviated by the argument `brief=TRUE` (see `help("S")` for details). One subtle difference in the output is that  $s = 1140$  is labeled the “residual standard deviation” by `S()`, to reflect our preferred usage, while it is potentially confusingly labeled the “residual standard error” by `summary()`. The estimated coefficient vector is returned by the standard R `coef()` function,

```
coef(trans.1)
```

| (Intercept) | t1     | t2     |
|-------------|--------|--------|
| 144.3694    | 5.4621 | 2.0345 |

and the full estimated coefficient covariance matrix is returned by the `vcov()` function:

```
(V <- vcov(trans.1))
```

|             | (Intercept) | t1        | t2          |
|-------------|-------------|-----------|-------------|
| (Intercept) | 29085.291   | 23.581695 | -12.6832940 |
| t1          | 23.582      | 0.187721  | -0.0315363  |
| t2          | -12.683     | -0.031536 | 0.0088994   |

The square roots of the diagonal elements of this matrix are the coefficient standard errors as reported by `S()` in the column labeled Std. Error:

```
sqrt(diag(V))
```

| (Intercept) | t1       | t2       |
|-------------|----------|----------|
| 170.544103  | 0.433268 | 0.094337 |

The pairwise correlations among the estimated coefficients are given by

```
round(cov2cor(V), 3)
```

|             | (Intercept) | t1     | t2     |
|-------------|-------------|--------|--------|
| (Intercept) | 1.000       | 0.319  | -0.788 |
| t1          | 0.319       | 1.000  | -0.772 |
| t2          | -0.788      | -0.772 | 1.000  |

At the risk of stating the obvious, the entries in this correlation matrix are estimated *coefficient* sampling correlations, not correlations among the *regressors*. We can compute the correlation between the variables t1 and t2 as follows:

```
round (with (Transact, cor (t1, t2)), 3)
```

```
[1] 0.772
```

The sampling correlations of the coefficients of t1 and t2 with the intercept, incidentally, are not a cause for alarm, but simply reflect the fact that the values of t1 and t2 (especially the latter) are generally far from zero. The *negative* correlation between the coefficients of t1 and t2 reflects the *positive* correlation between these variables, and in the simple case of just two regressors beyond the regressor for the intercept, .

## 5.1.2 Robust Regression Coefficient Standard Errors

When the assumption of constant error variance  $\sigma^2$  is violated, or if observations are correlated, ordinary-least-squares (OLS) estimates of coefficients remain consistent and unbiased, as long as the mean function in the regression model is correctly specified, but coefficient standard errors, and hence confidence intervals and hypothesis tests, can be seriously biased. In some instances, improved estimates of the coefficient standard errors can be obtained that are consistent even when the error variance isn't constant or when observations are correlated. In this section, we address nonconstant error variance.<sup>1</sup>

[1](#) We briefly take up standard errors for regression models fit to dependent data by OLS regression in [Chapter 7](#) on mixed-effects models and in the online appendix to the *R Companion* on time-series regression.

## Sandwich Standard Errors: Theory\*

To motivate the methodology, we use matrix notation. Write the linear model as

Other

$$\eta(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$$

where  $\mathbf{y}$  is the response vector,  $\mathbf{X}$  is the model matrix, and  $\varepsilon$  is the vector of regression errors. The usual least-squares estimate of  $\beta$  is

Other

$$E(y|\mathbf{x}) = \eta(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$$

where we introduce the additional notation  $\mathbf{C} = (\mathbf{X}' \mathbf{X})^{-1} \mathbf{X}'$  for the matrix of constants, depending on  $\mathbf{X}$  but not on  $\mathbf{y}$ , that transforms  $\mathbf{y}$  to  $\mathbf{b}$ . The covariance matrix of  $\mathbf{b}$  is then

Other

$$(5.1) \quad g[\mu(\mathbf{x})] = \eta(\mathbf{x})$$

The only unknown in this equation is  $\text{Var}(\mathbf{y})$ , the covariance matrix of  $\mathbf{y}$ . The usual OLS assumption is that  $\text{Var}(\mathbf{y}) = \text{Var}(\varepsilon) = \sigma^2 \mathbf{I}_n$ , where  $\mathbf{I}_n$  is the order- $n$  identity matrix, implying that all the errors have the same variance  $\sigma^2$  and are uncorrelated. We will instead assume more generally that  $\text{Var}(\varepsilon) = \Sigma$ , an arbitrary covariance matrix. Substituting  $\Sigma$  into Equation 5.1, we get

Other

$$(5.2) \quad \text{Var}(y|\mathbf{x}) = \phi \times V[\mu(\mathbf{x})]$$

When  $\Sigma = \sigma^2 \mathbf{I}_n$ , Equation 5.2 reduces to the standard OLS formula,  $\text{Var}(\mathbf{b}) = \sigma^2 (\mathbf{X}'\mathbf{X})^{-1}$ , but in general it is different. Equation 5.2 is not computable as it stands because it depends on the unknown value of  $\Sigma$ . The idea is to estimate  $\Sigma$  by some function of the residuals  $e_i$  from the OLS fit and then to use the estimate to compute coefficient standard errors. The resulting coefficient covariance matrix is called a *sandwich estimator*, with the two outer identical terms  $(\mathbf{X}'\mathbf{X})^{-1}$  in Equation 5.2 representing the “bread” and the inner term  $\mathbf{X}' \Sigma \mathbf{X}$  the “contents” of the sandwich.

Suppose that the observations are independently sampled, implying that  $\Sigma$  is a diagonal matrix but with potentially unequal positive diagonal elements. Each of the errors could consequently have a different variance, a problem generally called *nonconstant error variance* or, if you prefer long Greek-derived words, *heteroscedasticity*. White (1980) proposed using the diagonal matrix to estimate  $\Sigma$ . Work by Long and Ervin (2000) suggests that it is advantageous to use a slightly modified version of the White sandwich estimator, which they term HC3, based on ; here,  $h_i$  is the  $i$ th diagonal entry of the *hat-matrix*  $\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1} \mathbf{X}'$  (see [Section 8.3.2](#)).

## Sandwich Standard Errors: An Illustrative Application

We fit a regression model by OLS to the Transact data in [Section 5.1.1](#). OLS estimates are unbiased even if the variances of the errors are misspecified, but Cunningham and Heathcote (1989) suggested for this example that larger branches with more transactions are likely to have larger-magnitude errors. Under these circumstances sandwich estimates of coefficient standard errors should prove more accurate than the usual standard errors computed for OLS regression.<sup>2</sup>

<sup>2</sup> The preceding starred section explains the derivation of the term *sandwich*.

The hccm () function in the **car** package computes a sandwich estimate of the coefficient covariance matrix, assuming that the errors are independent and by default employing the HC3 modification (recommended by Long & Ervin, 2000) to White’s estimator (White, 1980):<sup>3</sup>

### ***hccm(trans.1)***

|             | (Intercept) | t1       | t2         |
|-------------|-------------|----------|------------|
| (Intercept) | 41273.514   | 98.35363 | -30.216838 |
| t1          | 98.354      | 0.53116  | -0.105559  |
| t2          | -30.217     | -0.10556 | 0.026717   |

[3](#) `hccm()` is an acronym for *heteroscedasticity-corrected covariance matrix*. The **sandwich** package (Zeileis, 2004) provides a similar function, `vcovHC()`, along with sandwich coefficient covariance estimators that apply in other circumstances, such as to dependent data (see [Section 7.2.7](#) for an application to clustered data). In addition, the **sandwich** package includes general infrastructure for sandwich coefficient covariance estimators.

The sandwich standard errors can be used directly in the model summary via the `vcov`. argument to `S()`:

### ***S(trans.1, vcov.=hccm)***

```
Call: lm(formula = time ~ t1 + t2, data = Transact)
Standard errors computed by hccm
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 144.369    203.159    0.71   0.48
t1           5.462      0.729    7.49  1.1e-12
t2           2.035      0.163   12.45 < 2e-16

Residual standard deviation: 1140 on 258 degrees of freedom
Multiple R-squared:  0.909
F-statistic: 876 on 2 and 258 DF,  p-value: <2e-16

AIC      BIC
4421.1  4435.3
```

The sandwich standard errors for the `Transact` regression are considerably larger than the coefficient standard errors computed by the conventional approach. The latter, therefore, underestimate the true variability of the regression coefficients.

The `vcov.` argument to `S()` allows the user to specify a function to compute the coefficient covariance matrix, as in the example where we set `vcov.=hccm`. Consequently, standard errors computed by `hccm()` are used for both the *t*-and *F*-values reported in the `S()` output. The `vcov.` argument can also be set to a directly supplied coefficient covariance, and so `vcov.=hccm(trans=1)` would produce exactly the same result. The default value is `vcov.=vcov`, which uses the standard R `vcov()` function to compute the covariance matrix of the coefficients.

Standard errors computed by a sandwich estimator are sometimes called *robust standard errors* because they provide reasonable results when the assumption of constant error variance does not hold and because the loss of efficiency in the standard errors is small when the error variance is in fact constant. Confidence intervals and hypothesis tests based on sandwich standard errors are approximate, and their validity depends on large-sample theory, so they may prove inaccurate in small samples.

Consult help ("`hccm`"), and the references given therein, for alternative sandwich estimators of the coefficient covariance matrix in the presence of nonconstant error variance. Sandwich estimators for nonconstant error variance are generally inappropriate for generalized linear models, which don't typically assume constant conditional variance and for which misspecification of the variance function can lead to biased estimates of coefficients (D. Freedman, 2006).

### 5.1.3 Using the Bootstrap to Compute Standard Errors

R is well suited for the bootstrap and other computationally intensive methods. The **boot** package (Canty & Ripley, 2017), which is associated with Davison and Hinkley (1997) and is part of the standard R distribution, and the **bootstrap** package, associated with Efron and Tibshirani (1993), provide comprehensive facilities in R for bootstrapping. Several other R packages are available for using the bootstrap in specialized situations. In this section, we describe the `Boot()` function in the **car** package, which serves as a relatively simple interface to the `boot()` function in the **boot** package.<sup>4</sup>

<sup>4</sup> A more complete description of the bootstrap and of the `Boot()` function is provided in an online appendix to the *R Companion*. In the second edition of the *R Companion*, we suggested the use of the `bootCase()` function in **car** for the

bootstrap. We now recommend using `Boot()`, which can be applied more generally to many different regression problems and uses the very high-quality algorithms in `boot()` to do the computing.

The essential idea of the *case-resampling bootstrap* is as follows:

1. Refit the regression with modified data, obtained by *sampling n cases with replacement from the n cases of the original data set*. Because sampling is done with replacement, some of the cases in the original data will be sampled several times and some not at all. Compute and save summary statistics of interest from this bootstrap sample.
2. Repeat Step 1 a large number of times, say  $R = 999$ .

The standard deviation of the  $R$  bootstrapped values of each statistic estimates its standard error. Because the bootstrap is a simulated random sampling procedure, its results are subject to sampling variation; using a large value of  $R$  makes sampling variation in the bootstrap estimate of the standard error small.

The `Boot()` function implements the case-resampling bootstrap. `Boot()` has several arguments, only the first of which is required:

#### **args ("Boot")**

```
function (object, f = coef, labels = names (f (object)), R = 999, method = c  
("case", "residual"), ncores = 1, ...) NULL
```

**object** A regression-model object produced, for example, by `lm()`.

**f** A function that returns a vector of statistics when it is applied to a model object; the values of these statistics are saved for each bootstrap replication. The default is `f=coef`, which returns the vector of regression coefficients for each bootstrap sample. Setting `f = function (m) c (coef (m), residSD= summary (m)$sigma)` would return both the coefficient estimates and the estimated residual SD.

**labels** Names for the elements of the vector of statistics returned by `f()`. The default usually works.

**R** The number of bootstrap replications, defaulting to 999.

**method** There are two primary forms of the nonparametric bootstrap, case resampling (`method="case"`), the default, which is described above and which we use in this section, and residual resampling (`method="residual"`), which is discussed in the online appendix to the *R Companion* on bootstrapping. There is also a parametric version of the bootstrap, where observations (e.g., errors in regression) are drawn from a known distribution, like the normal distribution, but it isn't implemented in the `Boot()` function.

**ncores** This argument is used to take advantage of parallel computations on computers with several processors or multiple cores and can be helpful in large problems, as described in the online appendix on bootstrapping.

We apply `Boot` to the `trans.1` model fit to the `Transact` data (on page 244):

```
set.seed(23435) # for reproducibility
```

```
betahat.boot <- Boot(trans.1)
```

Loading required namespace: `boot`

We resample from the data with the default  $R = 999$  bootstrap replications and on each replication save the vector of estimated regression coefficients. By setting the seed for R's random-number generator to a known value, we make it possible to reproduce our results exactly.<sup>5</sup> The estimates of the coefficients for the 999 replications are stored in `betahat.boot$t`. Here is a partial listing of the values of the bootstrapped regression coefficients:

```

brief(betahat.boot$t)

999 x 3 matrix (994 rows omitted)
      (Intercept)      t1      t2
[1,]      324.36 5.2856 2.0088
[2,]      72.89 5.7029 2.0970
[3,]      187.50 4.5655 2.1680
. . .
[998,]     111.97 4.7156 2.1725
[999,]     313.47 5.6376 1.9371

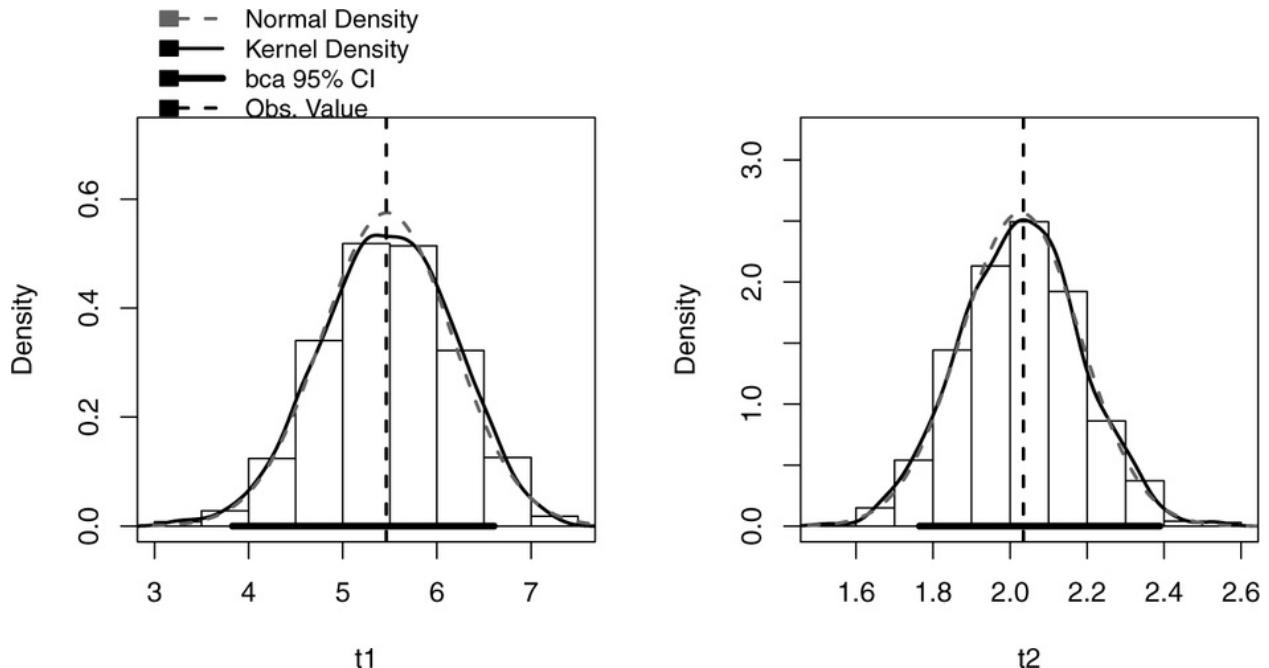
```

<sup>5</sup> See [Section 10.7](#) on conducting random simulations.

The object `betahat.boot` is of class "boot", and so all the functions in the **boot** and **car** packages for manipulating "boot" objects can be used. For example, the bootstrap replications of the regression coefficients can be viewed marginally (i.e., individually) in histograms, as shown in [Figure 5.1](#), because the generic `hist()` function has a method in the **car** package for "boot" objects:

```
hist(betahat.boot, parm=c ("t1", "t2"))
```

**Figure 5.1** Histograms of  $R = 999$  bootstrap estimates for the coefficients of  $t_1$  and  $t_2$  in the Transact regression.



To save space, we show only the histograms for the bootstrapped slope coefficients of  $t1$  and  $t2$ . Superimposed on each of the histograms is a normal density curve with mean and standard deviation matching those of the bootstrap samples, along with a nonparametric kernel density estimate. The regression coefficient estimate from the original data is marked by a dashed vertical line, and a confidence interval for the population regression coefficient based on the bootstrap samples is shown as a thick horizontal line at the bottom of the graph. In both graphs, the distribution of bootstrapped coefficients closely resembles a sample from a normal distribution, as is very frequently the case when the bootstrap is applied to regression models.

The **car** package also includes "boot" methods for the `confint()` and `Confint()` functions, to produce bootstrap confidence intervals, and a `vcov()` method for computing a bootstrap estimate of the covariance matrix of the regression coefficients (or of other bootstrapped statistics). There is also a `summary()` method for "boot" objects.<sup>6</sup>

```
summary(betahat.boot)
```

```
Number of bootstrap replications R = 999
                                original bootBias bootSE bootMed
(Intercept)      144.37  13.27146 196.055  155.31
t1                 5.46   0.00849   0.694    5.48
t2                 2.03  -0.00707   0.155    2.03
```

**6** See `help("summary.boot")` for a description of these functions in the **car** package and `help(package="boot")` for other functions for "boot" objects. After loading the **car** and **boot** packages, you can also enter the command methods (`class="boot"`).

The column `original` shows the OLS estimates of the regression coefficients; `bootBias` is the difference between the OLS coefficients and the average of the bootstrap replications and estimates the bias in the OLS coefficients; `bootSE` reports the bootstrap standard errors for the OLS estimates—simply the standard deviation of the  $R = 999$  bootstrap replications of each coefficient; and `bootMed` is the median of the 999 replications of each coefficient.

The `S()` function allows you to use bootstrapped standard errors in summarizing an "lm" model; for example:

```
S(trans.1, vcov.=vcov(betahat.boot))
```

```
Call: lm(formula = time ~ t1 + t2, data = Transact)
Standard errors computed by vcov(betahat.boot)
```

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 144.369  | 196.055    | 0.74    | 0.46     |
| t1          | 5.462    | 0.694      | 7.87    | 9.6e-14  |
| t2          | 2.035    | 0.155      | 13.09   | < 2e-16  |

Residual standard deviation: 1140 on 258 degrees of freedom

Multiple R-squared: 0.909

F-statistic: 930 on 2 and 258 DF, p-value: <2e-16

AIC BIC  
4421.1 4435.3

The bootstrapped standard errors of the regression coefficients are considerably larger than the standard errors computed in the usual manner for OLS estimates, as we would expect due to nonconstant error variance in the Transact data. The bootstrapped standard errors are similar to the robust standard errors computed for the model in [Section 5.1.2](#).

## 5.1.4 The Delta Method for Standard Errors of Nonlinear Functions\*

In [Section 4.2.1](#), we used the Davis data set to regress measured weight on reported weight, both in kg, for 183 men and women engaged in regular exercise, fitting the regression without Case 12, for whom measured weight was incorrectly recorded:

```
davis.mod <- lm(weight ~ repwt, data=Davis, subset=-12)
brief(davis.mod)
```

|            |             |        |
|------------|-------------|--------|
|            | (Intercept) | repwt  |
| Estimate   | 2.734       | 0.9584 |
| Std. Error | 0.815       | 0.0121 |

Residual SD = 2.25 on 180 df, R-squared = 0.972

With Case 12 removed, the fitted regression line has a positive intercept  $b_0 =$

2.73 and estimated slope  $b_1 = 0.96$ .

The fitted regression line and the line of unbiased prediction of weight from reported weight, with intercept zero and slope 1, cross at some point. For values of weight to the left of the crossing point, people on average overreport their weights, and for values of weight to the right of the crossing point, they underreport their weights. We can estimate the crossing point by solving for weight in the equation

Other

$$\log L_0 = \sum \log p[y_i; \hat{\mu}(\mathbf{x}_i), \phi]$$

The estimated crossover weight is therefore a *nonlinear* function of the regression coefficients, and so standard linear-model methods cannot be used to find the standard error of this quantity.

The *delta method* can be used to approximate the standard error of a nonlinear function of the coefficients. More generally, given an approximately normally distributed estimator  $\mathbf{b}$  of a vector of parameters  $\beta$  with estimated covariance matrix  $\mathbf{V}$ , an estimator of any function  $g(\beta)$  is  $g(\mathbf{b})$ , and its approximate variance is

Other

$$(5.3) \quad \log L_1 = \sum \log p(y_i; \mathbf{y}_i, \phi)$$

where  $\dot{g}(\mathbf{b})$  is the vector of partial derivatives of  $g(\beta)$  with respect to the elements of  $\beta$  (i.e.,  $\partial g(\beta)/\partial \beta$ ), evaluated at the estimates  $\mathbf{b}$ .

The `deltaMethod()` function in the `car` package performs this computation. For the example:

```
deltaMethod(davis.mod, "(Intercept)/(1 - repwt)")

              Estimate      SE 2.5 % 97.5 %
(Intercept)/(1 - repwt)    65.676 4.0134 57.81 73.542
```

The first argument to `deltaMethod()` is the regression model object `davis.mod`. The second argument is a quoted expression that when evaluated gives  $g(\mathbf{b})$ , which is specified using the names of the coefficients in the fitted-model object. The standard R function `D()` is used internally by `deltaMethod()` to differentiate this expression symbolically. The `deltaMethod()` function returns  $g(\mathbf{b})$  and its standard error, computed as the square root of Equation 5.3. The last two printed values are the limits of a 95% Wald confidence interval, computed assuming that  $g(\mathbf{b})$  is approximately normally distributed. By default, `deltaMethod()` uses the `vcov()` function to calculate the estimated covariance matrix of  $\mathbf{b}$ , but you can change this by supplying the `vcov.` argument, as in the `S()` function, for example, to employ a sandwich estimator of the coefficient covariance matrix.<sup>[7](#)</sup>

<sup>[7](#)</sup> See help ("`deltaMethod`") for how to use `deltaMethod()` with other classes of regression models.

As a second example, we return to the regression model that we fit to the Transact data in [Section 5.1.1](#). Imagine that we are interested in estimating the *ratio* of the regression coefficients  $t_1$  and  $t_2$  for the two kinds of transactions and in computing the standard error of the ratio. Using the delta method:

```
deltaMethod(trans.1, "t1/t2")
```

|           | Estimate | SE      | 2.5 %  | 97.5 % |
|-----------|----------|---------|--------|--------|
| $t_1/t_2$ | 2.6847   | 0.31899 | 2.0595 | 3.3099 |

Let's compare the standard error produced by the delta method for this example to the bootstrapped standard error, conveniently obtained by reusing the bootstrap samples that we computed earlier for the Transact regression in [Section 5.1.3](#):

```
boots <- cbind(betahat.boot$t,
  "t1/t2"= betahat.boot$t[, "t1"]/betahat.boot$t[, "t2"])
nrow(boots)

[1] 999
```

```
head(boots)
```

|       | (Intercept) | t1     | t2     | t1/t2  |
|-------|-------------|--------|--------|--------|
| [1, ] | 324.361     | 5.2856 | 2.0088 | 2.6313 |
| [2, ] | 72.890      | 5.7029 | 2.0970 | 2.7195 |
| [3, ] | 187.498     | 4.5655 | 2.1680 | 2.1058 |
| [4, ] | 51.116      | 5.6813 | 2.0429 | 2.7810 |
| [5, ] | 206.675     | 4.5571 | 2.0897 | 2.1807 |
| [6, ] | -10.487     | 5.1407 | 2.1269 | 2.4170 |

```
sd(boots[, "t1/t2"]) # bootstrap SE
```

```
[1] 0.54156
```

The delta method is optimistic for this example, producing an estimated standard error,  $SE_{\text{delta}} = 0.319$ , that is much smaller than the bootstrap standard error,  $SE_{\text{boot}} = 0.541$ , and therefore the delta method confidence interval is too short.

The assumption of constant error variance is doubtful for the Transact regression. Consequently, the estimated variances of the regression coefficients are too small, and so the estimated variance of their ratio is also too small. Using `hccm()` with `deltaMethod()` to estimate the coefficient covariance matrix,

```
deltaMethod(trans.1, "t1/t2", vcov.=hccm)
```

|       | Estimate | SE      | 2.5 %  | 97.5 % |
|-------|----------|---------|--------|--------|
| t1/t2 | 2.6847   | 0.55836 | 1.5903 | 3.779  |

produces a delta method standard error that is very close to the bootstrap value.

Although the `deltaMethod()` function is designed to compute a standard error for a nonlinear function of regression coefficients, it can be applied to any function of statistics for which the covariance matrix is either known or for which an estimate is available. Consequently, `deltaMethod()` can also be used to calculate the standard error of a linear combination of coefficients; for example,

```

deltaMethod(trans.1, "t1 + t2")

      Estimate       SE  2.5 % 97.5 %
t1 + t2    7.4966 0.36544 6.7804 8.2129

```

returns the estimate and standard error of the sum of the regression coefficients for t1 and t2.

## 5.2 Confidence Intervals

### 5.2.1 Wald Confidence Intervals

Wald confidence intervals, named for Abraham Wald (1902–1950), use only a statistic, its standard error (or if the statistic is multivariate, its estimated covariance matrix), and percentiles of an assumed sampling distribution to get confidence intervals. The most familiar instance in the linear regression setting is the confidence interval for a regression coefficient  $\beta_j$  of the form

Other

$$D(\mathbf{y}; \hat{\boldsymbol{\mu}}) = 2(\log L_1 - \log L_0)$$

where  $t_d$  is a percentile of the  $t$ -distribution with  $d$  df. For example, the 0.975 quantile is used for a 95% confidence interval, while the 0.90 quantile is used for an 80% confidence interval. These quantiles are easily computed in R using qt(), although many functions compute Wald intervals as a matter of course.

The Wald procedure is justified when the regression errors are normally distributed and the model is correctly specified, because  $b_j$  is then normally distributed,  $[\text{SE}(b_j)]^2$  has a multiple of a chi-square distribution, and these two statistics are independent. Without normality, the Wald interval is still justified in large-enough samples.

The confint() function in base R and the substitute Confint() function in the **car** package both compute Wald confidence intervals for linear models. Confint() includes a vcov. argument, facilitating the use of a coefficient covariance matrix different from the standard one, either specified directly or in the form of a

function to be applied to an "lm" model object. Exact distributions are generally not available when confidence intervals are computed with an alternative coefficient covariance matrix estimator, and thus only the large-sample justification applies.

For the regression model we fit to the Transact data, where there is evidence of nonconstant error variance, we can use a robust estimator of the coefficient covariance matrix, computed by the hccm () function (and discussed in [Section 5.1.2](#)):

### ***Confint(trans.1, vcov.=hccm)***

Standard errors computed by hccm

|             | Estimate | 2.5 %     | 97.5 %   |
|-------------|----------|-----------|----------|
| (Intercept) | 144.3694 | -255.6912 | 544.4301 |
| t1          | 5.4621   | 4.0269    | 6.8972   |
| t2          | 2.0345   | 1.7127    | 2.3564   |

### ***confint(trans.1)***

|             | 2.5 %     | 97.5 %   |
|-------------|-----------|----------|
| (Intercept) | -191.4662 | 480.2051 |
| t1          | 4.6089    | 6.3152   |
| t2          | 1.8488    | 2.2203   |

Unlike confint (), Confint () reports the estimated coefficients along with the confidence limits. In this problem, the confidence intervals produced by confint (), which wrongly assume constant error variance, are unrealistically narrow in comparison to the intervals produced by Confint () using a robust estimator of the coefficient covariance matrix.

## **5.2.2 Bootstrap Confidence Intervals**

One of the primary applications of the bootstrap is to confidence intervals. The bootstrap is attractive in this context because it requires fewer distributional assumptions than other methods. If `b1` is an object produced by the `Boot()` function, then setting the argument `vcov.=vcov(b1)` in a call to `Confint()` uses the sample covariance matrix of the bootstrap replicates to estimate coefficient standard errors, otherwise computing Wald confidence intervals in the standard manner. In particular, for linear models, quantiles of the  $t$ -distribution are used for confidence intervals when the quantiles of the normal distribution are more appropriate. The intervals produced are therefore somewhat too wide, an issue we consider unimportant because the justification of bootstrap confidence intervals is in any event asymptotic.

For the `Transact` data, we obtain bootstrap confidence intervals that are very similar to those produced in the preceding section with a robust estimate of the coefficient covariance matrix. Reusing the bootstrap samples in `betahat.boot` that we computed in [Section 5.1.3](#), we find

```
Confint(trans.1, vcov.=vcov(betahat.boot))
```

|             | Estimate | 2.5 %     | 97.5 %   |
|-------------|----------|-----------|----------|
| (Intercept) | 144.3694 | -241.7028 | 530.4417 |
| t1          | 5.4621   | 4.0960    | 6.8282   |
| t2          | 2.0345   | 1.7285    | 2.3406   |

Although the Wald-based bootstrap methodology implemented in the **car** package is adequate for most problems, the `boot.ci()` function in the **boot** package includes several other methods for computing confidence intervals that are occasionally more accurate. The "norm" method in `boot.ci()` is similar to the method employed by `Confint()` using the bootstrap coefficient standard errors, except that `boot.ci()` uses normal rather than  $t$  quantiles and adjusts the confidence limits for the bootstrap estimate of bias; for linear and generalized linear models, the bias is likely to be very small. More elaborate methods implemented in `boot.ci()` are based on percentiles of the bootstrap samples: The "perc" method uses simple percentiles, and the "bca" method uses percentiles adjusted for bias.<sup>8</sup> See Davison and Hinkley (1997, Section 5.2) or Fox (2016,

Section 21.2) for an explanation of the methodology.

[8](#) Applied directly to a "boot" object such as `betahat.boot`, `Confint()` by default computes  $bc_a$  confidence intervals: Try it!

### 5.2.3 Confidence Regions and Data Ellipses\*

The `Confint()` function computes *marginal* confidence intervals separately for each regression coefficient. In this section, we discuss *simultaneous confidence regions* for regression coefficients. We concentrate on confidence regions for two coefficients at a time so that we can draw two-dimensional pictures, and we work with Duncan's occupational-prestige data introduced in [Chapter 1](#), treating prestige as the response and income and education as predictors:

```
duncan.mod <- lm(prestige ~ income + education, data=Duncan)
(ci <- Confint(duncan.mod))

      Estimate      2.5 %    97.5 %
(Intercept) -6.06466 -14.68579 2.55646
income        0.59873   0.35723 0.84023
education     0.54583   0.34755 0.74412
```

Generalization of these intervals to more than one dimension, assuming normality, produces *joint confidence regions*, called *confidence ellipses* in two dimensions and *confidence ellipsoids* in more than two dimensions.

The `confidenceEllipse()` function in the `car` package can be used to draw a confidence ellipse for the coefficients of the two predictors in Duncan's regression, as shown in [Figure 5.2 \(a\)](#):

```
confidenceEllipse(duncan.mod, levels=c(0.85, 0.95), main="(a)")

abline(v=ci[2, 2:3], lty=2, lwd=2) # marginal CI for education

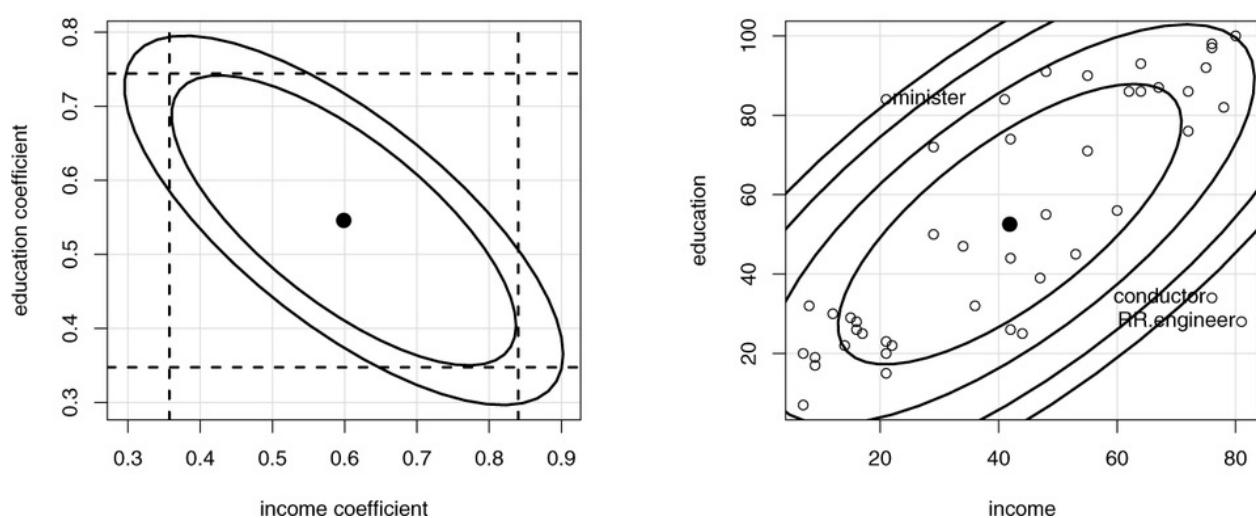
abline(h=ci[3, 2:3], lty=2, lwd=2) # marginal CI for income
```

**Figure 5.2 (a)** 95% (outer) and 85% (inner) joint confidence ellipses for the

coefficients of income and education in Duncan's regression of prestige on these predictors, augmented by the marginal 95% confidence intervals (broken lines). The black dot represents the estimated regression coefficients,  $(b_1, b_2)$ . (b) Scatterplot and 50%, 75%, 90%, and 95% data ellipses for income and education. In (b), three cases are identified as unusual: minister, conductor, and RR.engineer. The black dot represents the means of the two variables, .

(a)

(b)



The outer ellipse in this graph is a 95% joint confidence region for the population regression coefficients  $\beta_1$  and  $\beta_2$ : With repeated sampling, 95% of such ellipses will simultaneously include  $\beta_1$  and  $\beta_2$ , if the fitted model is correct and normality holds. The orientation of the ellipse reflects the negative correlation between the estimates. Contrast the 95% confidence ellipse with the marginal 95% confidence intervals, also shown on the plot. Some points within the marginal intervals—with larger values for both of the coefficients, for example—are implausible according to the joint region. Similarly, the joint region includes values of the coefficient for income, for example, that are excluded from the marginal interval. The inner ellipse, generated with a confidence level of 85% and termed the *confidence interval generating ellipse*, has perpendicular shadows on the parameter axes that correspond to the marginal 95% confidence intervals for the coefficients.

When variables have a bivariate-normal distribution, *data ellipses* represent estimated probability contours, containing expected fractions of the data. The `dataEllipse()` function in the **car** package draws data ellipses for a pair of variables. We illustrate with income and education in Duncan's occupational-prestige data:

*with (Duncan,*

*dataEllipse (income, education,*

*levels=c (0.5, 0.75, 0.9, 0.95),*

*id=list (method="mahal", n=3, labels=rownames (Duncan)),*

*main="(b)"*

#x2009; )

We use the id argument to label the three points farthest in Mahalanobis distance from the center of the data using the row labels of the Duncan data frame. The result appears in [Figure 5.2 \(b\)](#). The contours of the data ellipses are set to enclose 50%, 75%, 90%, and 95% of bivariate-normal data. The ellipses also represent contours of constant Mahalanobis distance from the point of means of the two variables.

The 95% ellipses in the two panels of [Figure 5.2](#) differ in shape by only a 90° rotation, because the data ellipse is based on the sample covariance matrix of the predictors, while the confidence ellipse is based on the sample covariance matrix of the slope coefficients, which is proportional to the inverse of the sample covariance matrix of the predictors.

For more on data ellipses, confidence ellipses, and the role of ellipses and ellipsoids more generally in statistics, see Friendly et al. (2013).

## 5.3 Testing Hypotheses About Regression Coefficients

The accuracy of  $p$ -values for standard hypothesis tests about linear regression models depends either on the normality of the errors or on sufficiently large sample size. Under these circumstances, test statistics generally have  $t$ - or  $F$ -distributions, at least approximately. For example, if  $x_1, x_2, \dots, x_n$  are the values of the predictor in a simple regression, then the  $t$ -statistic for testing the hypothesis that the slope is zero has an exact  $t$ -distribution with  $n - 2$   $df$  if the errors are independently and normally distributed with equal variance. Even if the errors are nonnormal, the test statistic is approximately normally distributed

as long as gets close to zero as  $n$  increases (Huber & Ronchetti, 2009).

### 5.3.1 Wald Tests

Wald tests are based on a Wald statistic, which is the ratio of an estimate that has mean zero under the null hypothesis to its standard error. The  $t$ -values in the standard output produced by `S()` for a linear model, along with the associated  $p$ -values, are Wald tests that each coefficient is zero against a two-sided alternative, with the values of the other regression coefficients unspecified.

We also can easily compute a Wald test that a coefficient has *any* specific value, not necessarily zero. For the regression of measured on reported weight in the Davis data set (introduced in [Section 4.2.1](#) and used to illustrate the delta method in [Section 5.1.4](#)), for example, we can test that the slope is equal to 1, which would be the case if individuals were unbiased reporters of their weights:

```
tval <- (coef(davis.mod)[2] - 1)/sqrt(vcov(davis.mod)[2, 2])
pval <- 2*pt(abs(tval), df.residual(davis.mod),
             lower.tail=FALSE)
c(t=tval, p=pval)

t.repwt      p.repwt
-3.42803422 0.00075353
```

The estimated variance of the slope coefficient is `vcov(davis.mod)[2, 2]`, extracted from the second row, second column of the coefficient covariance matrix returned by `vcov()`. The `pt()` function is used to compute a two-tailed  $p$ -value, by taking the area to the right of  $|t|$  and doubling it. Wald tests like this, as well as Wald tests for several parameters simultaneously, can be computed more conveniently using the `linearHypothesis()` function in the `car` package (see [Section 5.3.5](#)).

Wald tests for coefficients in linear models are equivalent to likelihood-ratio tests of the same hypotheses (discussed in the next section), but this is not necessarily true for other kinds of regression models, such as the generalized linear models introduced in the next chapter.

### 5.3.2 Likelihood-Ratio Tests and the Analysis of

# Variance

One of the fundamental ideas in hypothesis testing is that of model comparison according to the following paradigm: As with all classical hypothesis tests, we compare a null and an alternative hypothesis. The two hypotheses differ only by the terms included in the model, with the null-hypothesis model obtainable from the alternative-hypothesis model by setting some of the parameters in the latter to zero or to other prespecified values. We have already seen one instance of this approach in the overall  $F$ -test (in [Chapter 4](#), page 179), for which the null model has the mean function  $E(y|\mathbf{x}) = \beta_0$  versus the alternative model with mean function  $E(y|\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$ , testing that all the coefficients with the exception of the intercept  $\beta_0$  are zero. The null hypothesis is tested by fitting both models and then computing an incremental  $F$ -statistic.

We use the Prestige data to provide examples. First, we invoke the standard R `anova()` function to compute the overall  $F$ -test statistic:

```
prestige.mod.1 <- lm(prestige ~ education + log2(income) + type,
                      data=na.omit(Prestige)) # full model
prestige.mod.0 <- update(prestige.mod.1, . ~ 1) # intercept only
anova(prestige.mod.0, prestige.mod.1) # compare models

Analysis of Variance Table
Model 1: prestige ~ 1
Model 2: prestige ~ education + log2(income) + type
Res.Df   RSS Df Sum of Sq    F Pr(>F)
1      97 28347
2      93 4096  4      24251 138 <2e-16
```

The test requires that each model fit to the data uses the same cases. Because the variable `type` in the Prestige data set has some missing values, we are careful to use `na.omit()` to filter the Prestige data frame for missing data prior to fitting the models, ensuring that both models are fit to the same subset of cases with complete data on all four variables in the larger model.<sup>9</sup>

<sup>9</sup> See [Section 2.3.2](#) on handling missing data in R.

The overall  $F$ -test is based on the difference between the residual sums of

squares for the two models, on the difference in  $df$  for error, and on the estimate of the error variance  $\sigma^2$  from the larger model. In this case,  $F$  has (4, 93)  $df$ . The overall  $F$ -statistic is also printed in the summary output for a linear model. Here, the  $p$ -value ( $p < 2 \times 10^{-16}$ ) is essentially zero, and so the null hypothesis is emphatically rejected.

We can use this method for any comparison of nested models, where one model is a specialization of the other. For example, to test the null hypothesis that only the coefficient for `log2(income)` is zero against the same alternative hypothesis as before:

```
prestige.mod.0inc <- update(prestige.mod.1,
  . ~ . - log2(income))
anova(prestige.mod.0inc, prestige.mod.1) # compare models
```

Analysis of Variance Table

Model 1: prestige ~ education + type

Model 2: prestige ~ education + log2(income) + type

|   | Res.Df | RSS  | Df | Sum of Sq | F    | Pr(>F)  |
|---|--------|------|----|-----------|------|---------|
| 1 | 94     | 5740 |    |           |      |         |
| 2 | 93     | 4096 | 1  | 1644      | 37.3 | 2.3e-08 |

Once again, the null hypothesis is clearly rejected. For a test such as this of a single coefficient, the  $F$ -statistic is just  $t^2$  from the corresponding Wald test.

The standard `anova()` function is a generic function, with methods for many classes of statistical models.<sup>10</sup> For example, in the next chapter, we use `anova()` to obtain likelihood-ratio  $\chi^2$  tests for generalized linear models.

<sup>10</sup> Generic functions and their methods are discussed in detail in [Sections 1.7](#) and [10.9](#).

### 5.3.3 Sequential Analysis of Variance

As we have seen, the `anova()` function is useful for comparing two nested models. We can also apply `anova()` to an individual "lm" object, producing a

*sequential analysis-of-variance table:*

**anova (prestige.mod. 1)**

Analysis of Variance Table

Response: prestige

|              | Df | Sum Sq | Mean Sq | F value | Pr (>F) |
|--------------|----|--------|---------|---------|---------|
| education    | 1  | 21282  | 21282   | 483.19  | < 2e-16 |
| log2(income) | 1  | 2499   | 2499    | 56.74   | 3.2e-11 |
| type         | 2  | 469    | 235     | 5.32    | 0.0065  |
| Residuals    | 93 | 4096   | 44      |         |         |

Sequential analysis of variance is sometimes known by the less descriptive name SAS *Type I analysis of variance*, or just *Type I analysis of variance*.<sup>11</sup> The ANOVA table includes three tests, produced by fitting a sequence of models to the data; these tests are for

1. education *ignoring* log2(income) and type;
2. log2(income) *after* education but *ignoring* type; and
3. type *after* education and log2(income).

<sup>11</sup> The terms Type I, Type II, and Type III tests do not have consistent definitions, and the way that we use the terms in this text and in the Anova () function in the **car** package, which we explain in this chapter, doesn't correspond precisely to how the terms are implicitly defined by the GLM (linear model) procedure in SAS. Although we are tempted to introduce new terms for these tests, we feel that our current usage is sufficiently common that doing so could invite further confusion.

The successive lines in the ANOVA table show tests for comparing the following models, as specified by their formulas:

prestige ~ 1 versus prestige ~ education

`prestige ~ education` versus

`prestige ~ education + log2(income)`

`prestige ~ education + log2(income)` versus

`prestige ~ education + log2(income) + type`

The first two of these hypothesis tests are generally not of interest, because the test for education fails to control for `log2(income)` and `type`, while the test for `log2(income)` fails to control for `type`. The third test *does* correspond to a sensible hypothesis, for equality of intercepts of the three levels of `type`—that is, no effect of `type` controlling for `education` and `income`.

A more subtle difference between the sequential tests produced by the `anova()` function applied to a single model and the incremental  $F$ -test produced when `anova()` is applied to a pair of nested models concerns the manner in which the error variance  $\sigma^2$  in the denominator of the  $F$ -statistics is estimated: When we apply `anova()` to a pair of nested models, the error variance is estimated from the residual sum of squares and degrees of freedom for the larger of the two models, which corresponds to the alternative hypothesis. In contrast, *all* of the  $F$ -statistics in the sequential ANOVA table produced when `anova()` is applied to a single model are based on the estimated error variance from the full model, `prestige ~ education + log2(income) + type` in our example, which is the largest model in the sequence. Although this model includes terms that are extraneous to (i.e., excluded from the alternative hypothesis for) all but the last test in the sequence, it still provides an unbiased estimate of the error variance for all of the tests. It is traditional in formulating an ANOVA table to base the estimated error variance on the largest model fit to the data.

### 5.3.4 The `Anova()` Function

The `Anova()` function in the `car` package, with an uppercase A to distinguish it from `anova()`, calculates hypotheses that are generally of more interest, by default reporting so-called *Type II tests*:

## Anova (*prestige.mod.1*)

Anova Table (Type II tests)

Response: prestige

|              | Sum Sq | Df | F value | Pr (>F) |
|--------------|--------|----|---------|---------|
| education    | 1285   | 1  | 29.17   | 5.1e-07 |
| log2(income) | 1644   | 1  | 37.32   | 2.3e-08 |
| type         | 469    | 2  | 5.32    | 0.0065  |
| Residuals    | 4096   | 93 |         |         |

The hypotheses tested here compare the following pairs of models:<sup>12</sup>

prestige ~ log2(income) + type versus

prestige ~ education + log2(income) + type

prestige ~ education + type versus

prestige ~ education + log2(income) + type

prestige ~ education + log2(income) versus

prestige ~ education + log2(income) + type

[12](#) Anova () is able to perform these computations without refitting the model.

In each case, we get a test for adding one of the predictors to a model that includes all of the others. The last of the three tests is identical to the third test from the sequential analysis of variance, but the other two tests are different and are more generally sensible. Because the Type II tests for education and log2(income) are tests of terms that are each represented by a single coefficient, the reported  $F$ -statistics are equal to the squares of the corresponding  $t$ -statistics in the model summary.

## Tests for Models With Factors and Interactions

Tests for models with factors and interactions are generally summarized in an analysis-of-variance table. We recommend using the Type II tests computed by default by the Anova () function. For example:

```
prestige.mod.3 <- update(prestige.mod.1,  
  . ~ . + education:type + log2(income):type)  
Anova(prestige.mod.3)
```

Anova Table (Type II tests)

Response: prestige

|                   | Sum Sq | Df | F value | Pr (>F) |
|-------------------|--------|----|---------|---------|
| education         | 1209   | 1  | 29.44   | 4.9e-07 |
| log2(income)      | 1691   | 1  | 41.17   | 6.6e-09 |
| type              | 469    | 2  | 5.71    | 0.0046  |
| education:type    | 179    | 2  | 2.18    | 0.1195  |
| log2(income):type | 290    | 2  | 3.53    | 0.0333  |
| Residuals         | 3655   | 89 |         |         |

Type II analysis of variance obeys the *principle of marginality* (Nelder, 1977), summarized in [Table 5.1](#) for the example.<sup>13</sup>

<sup>13</sup> See [Section 4.6.2](#) for a discussion of the principle of marginality in the formulation of linear models in R.

**Table 5.1**

| <i>Sum of Squares for</i> | <i>after...</i>                                     | <i>ignoring ...</i>                  |
|---------------------------|---|--------------------------------------|
| education                 | log2(income), type,<br>log2(income):type            | education:type                       |
| log2(income)              | education, type,<br>education:type                  | log2(income):type                    |
| type                      | education, log2(income)                             | education:type,<br>log2(income):type |
| education:type            | education, log2(income),<br>type, log2(income):type |                                      |
| log2(income):type         | education, log2(income),<br>type, education:type    |                                      |

All of the tests compare two models. For example, the test for  $\text{log2}(\text{income})$  compares a smaller model consisting of all terms that do not involve  $\text{log2}(\text{income})$  to a model that includes all this plus  $\text{log2}(\text{income})$ . The general principle is that the test for a *lower-order term*, such as a *main effect* (i.e., a separate partial effect) like  $\text{log2}(\text{income})$ , is never computed after fitting a *higher-order term*, such as an interaction, that *includes* the lower-order term, as  $\text{log2}(\text{income}):type$  includes  $\text{log2}(\text{income})$ . The error variance for all of the tests is estimated from the full model—that is, the largest model fit to the data. As we mentioned in connection with the `anova()` function, although it would also be correct to estimate the error variance for each test from the larger model for that test, using the largest model produces an unbiased estimate of  $\sigma^2$  even when it includes extraneous terms.

If the regressors for different terms in a linear model are mutually orthogonal (that is, uncorrelated), then Type I and Type II tests are identical. When the regressors are not orthogonal, Type II tests address more generally sensible hypotheses than Type I tests.

## Using a Nonstandard Coefficient Covariance Matrix

Like the `S()`, `Confint()`, and `linearHypothesis()` functions (the latter to be discussed in [Section 5.3.5](#)), the `Anova()` function has an optional `vcov` argument, allowing the user to supply a nonstandard coefficient covariance matrix. For example, for the regression model that we fit to the `Transact` data (in [Section 5.1.1](#)), we can take account of nonconstant error variance by using a sandwich estimate of the coefficient covariance matrix in formulating tests:<sup>14</sup>

### **Anova(trans.1, vcov.=hccm)**

Coefficient covariances computed by hccm

Analysis of Deviance Table (Type II tests)

Response: time

|               | Df | F     | Pr (>F) |
|---------------|----|-------|---------|
| t1            | 1  | 56.2  | 1.1e-12 |
| t2            | 1  | 154.9 | < 2e-16 |
| Residuals 258 |    |       |         |

[14](#) The term *analysis of deviance* is more general than *analysis of variance*, here signifying the use of a non-standard coefficient covariance matrix; we'll encounter this term again in [Section 6.3.4](#) in connection with tests for generalized linear models.

When, as here, a nonstandard coefficient covariance matrix is used, no sums of squares are reported. In this example, because each term in the model has 1 *df*, the *F*-tests reported by Anova () are the squares of the *t*-tests reported by S () (cf. [Section 5.1.2](#)), but Anova () is more generally useful in that it is applicable to models with multiple-*df* terms and interactions.

## **Unbalanced ANOVA**

How to formulate hypotheses, contrasts, and sums of squares in unbalanced two-way and higher-way analysis of variance is the subject of a great deal of controversy and confusion. For balanced data, with all cell counts equal, none of these difficulties arise. This is not the place to disentangle the issue, but we will nevertheless make the following brief points:[15](#)

[15](#) See Fox (2016, Section 8.2) for an extended discussion.

- The essential goal in analysis of variance is to test sensible hypotheses about differences among cell means and averages of the cell means. Sums

of squares and tests should follow from the hypotheses.

- It is difficult to go wrong if we construct tests that conform to the principle of marginality, always ignoring higher-order relatives—for example, ignoring the A:B interaction when testing the A main effect. This approach produces Type II tests.
- Some people do test lower-order terms *after* their higher-order relatives—for example, the main effect of A after the B main effect *and* the A:B interaction. The main effect of A, in this scheme, represents the effect of A averaged over the levels of B. Whether or not this effect really is of interest is another matter, which depends upon context. The incremental  $F$ -test for a term after everything else in the model is called a *Type III test*. The Type I and Type II tests are the same for any choice of contrasts for a factor, and therefore the same hypotheses are tested regardless of the contrasts. This is not so for Type III tests, and the hypotheses tested are complicated functions of the parameters and the counts in each cell of the contingency table cross-classifying the two factors. If we choose to perform Type III tests, then we should use `contr.sum()`, `contr.helmert()`, or `contr.poly()` to code factors (see [Section 4.7](#)); in particular, we should not use the default `contr.treatment` with Type III tests.<sup>16</sup>
- [Figure 5.3](#) illustrates these points for a two-way ANOVA with factors A and B, which have three and two levels, respectively. The two graphs in the figure show imagined patterns of population cell means  $\mu_{ab}$  for the six combinations of levels of the two factors, along with the *population marginal mean*  $\mu_a$  for each level of factor A averaging across the two levels of factor B; for example,  $\mu_{1\cdot} = (\mu_{11} + \mu_{12})/2$ . In panel (a) of [Figure 5.3](#), there is no interaction between factors A and B. Under these circumstances, the profiles of cell means are parallel, and the profile of marginal means for factor A captures the common pattern of cell-means profiles at the two levels of factor B. In panel (b), the factors interact—the partial effect of factor A is different at the two levels of factor B—as reflected in the nonparallel profiles of cell means.

The Type II test for the main effect of factor A is the maximally powerful test that the population marginal means for A are identical,  $H_0: \mu_{1\cdot} = \mu_{2\cdot} = \mu_{3\cdot}$ , under the assumption of no interaction—that is, assuming that the profiles of cell means are parallel. If the profiles are not parallel, the Type II test doesn't test this hypothesis. The properly formulated Type III test of the

main effect of A, in contrast, tests the hypothesis  $H_0: \mu_{1.} = \mu_{2.} = \mu_{3.}$  whether or not the profiles of cell means are parallel but is less powerful than the Type II test of the main effect of A if there is no interaction.

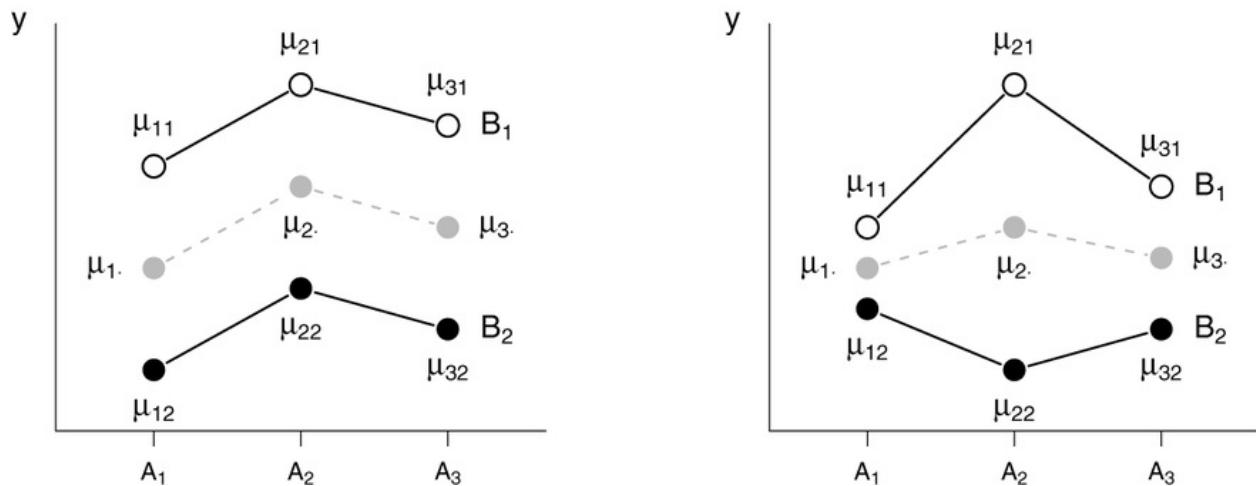
- These considerations also apply to the analysis of covariance: Imagine a regression model with one factor (A), coded by 0/1 dummy regressors, and one quantitative predictor (or covariate, X). Suppose that we test the hypothesis that the main effect of A is zero in the model that includes interaction between A and X. This tests that the intercepts for the different levels of A are the same. If the slopes for X vary across the levels of A, then the separation among the levels varies with the value of X, and assessing this separation at  $X = 0$  is probably not sensible. To justify Type III tests, we could express X as deviations from its mean, making the test for differences in intercepts a test for differences among levels of A at the average score of X. Proceeding in this manner produces a sensible, but not necessarily interesting, test. We invite the reader to draw pictures for analysis of covariance analogous to [Figure 5.3](#) for two-way ANOVA.

[16](#) \* The contrasts produced by `contr.sum()`, `contr.helmert()`, and `contr.poly()` for different factors are orthogonal in the row basis of the model matrix, but those produced by `contr.treatment()` are not.

**Figure 5.3** Population cell means for factors A and B in a  $3 \times 2$  ANOVA (hollow and solid black dots connected by solid black lines), along with the population marginal means for factor A (solid gray dots connected by broken gray lines): (a) no interaction between A and B; (b) interaction between A and B.

(a) No Interaction

(b) Interaction



Source: Adapted from Fox (2016, Figure 8.2).

In light of these considerations, we obtain the analysis-of-variance table for the two-way ANOVA model that we fit to the GSS vocabulary data in [Section 4.6.3](#). For Type II tests, we can use the default dummy-coded contrasts computed by `contr.treatment()`:<sup>17</sup>

```
GSS16 <- na.omit(subset(GSSvocab, year=="2016",
  select=c("vocab", "ageGroup", "nativeBorn")))
mod.vocab <- lm(vocab ~ nativeBorn*ageGroup, data=GSS16)
Anova(mod.vocab)
```

Anova Table (Type II tests)

Response: vocab

|                     | Sum Sq | Df   | F value | Pr (>F) |
|---------------------|--------|------|---------|---------|
| nativeBorn          | 153    | 1    | 42.68   | 8.3e-11 |
| ageGroup            | 66     | 4    | 4.61    | 0.0011  |
| nativeBorn:ageGroup | 14     | 4    | 1.01    | 0.4010  |
| Residuals           | 6615   | 1848 |         |         |

[17](#) The GSS vocabulary data are unbalanced, so Type I, II, and III tests differ; for balanced data, the three types of tests are identical, when the Type III tests, which depend on contrast coding, are computed correctly, as we do below.

There is, therefore, strong evidence for the main effects of ageGroup and native-Born on vocabulary score but no evidence of an interaction between the two.

To get sensible Type III tests, we must change the factor coding, which requires refitting the model;<sup>18</sup> one approach is as follows:

```
contrasts(GSS16$ageGroup) <-
  contrasts(GSS16$nativeBorn) <- "contr.sum"
contrasts(GSS16$ageGroup)
```

|       | [,1] | [,2] | [,3] | [,4] |
|-------|------|------|------|------|
| 18-29 | 1    | 0    | 0    | 0    |
| 30-39 | 0    | 1    | 0    | 0    |
| 40-49 | 0    | 0    | 1    | 0    |
| 50-59 | 0    | 0    | 0    | 1    |
| 60+   | -1   | -1   | -1   | -1   |

```
contrasts(GSS16$nativeBorn)
```

|     | [,1] |
|-----|------|
| no  | 1    |
| yes | -1   |

```
mod.vocab.3 <- update(mod.vocab)
```

```
Anova(mod.vocab.3, type="III")
```

Anova Table (Type III tests)

Response: vocab

|                     | Sum Sq | Df   | F value | Pr(>F)  |
|---------------------|--------|------|---------|---------|
| (Intercept)         | 25397  | 1    | 7094.92 | < 2e-16 |
| nativeBorn          | 155    | 1    | 43.31   | 6.1e-11 |
| ageGroup            | 50     | 4    | 3.47    | 0.0078  |
| nativeBorn:ageGroup | 14     | 4    | 1.01    | 0.4010  |
| Residuals           | 6615   | 1848 |         |         |

[18](#) The contrast coding has no effect, however, on the Type II tests—try it!

In this instance, the Type II and Type III tests produce very similar results.[19](#) The

reader may wish to verify that repeating the analysis with Helmert contrasts (`contr.helmert()`) produces the same Type III ANOVA, while the ANOVA table produced using the default `contr.treatment()` is different (and incorrect).

[19](#) The Type III ANOVA table produced by `Anova()` also includes a test that the intercept is zero. This hypothesis is not usually of interest. In two-way ANOVA, the Type II and Type III tests for the interaction of the factors are identical, as the example illustrates.

### 5.3.5 Testing General Linear Hypotheses\*

The *general linear hypothesis* generalizes the Wald  $t$ -test and, for a linear model, is an alternative way of formulating model comparison  $F$ -tests without refitting the model. A matrix formulation of the linear models considered in this chapter is

Other

$$\log \left[ \frac{\mu(\mathbf{x})}{1 - \mu(\mathbf{x})} \right] = \eta(\mathbf{x})$$

or, equivalently,

Other

$$\mu(\mathbf{x}) = \frac{1}{1 + \exp[-\eta(\mathbf{x})]}$$

where  $\mathbf{y}$  is an  $n \times 1$  vector containing the response;  $\mathbf{X}$  is an  $n \times (k + 1)$  model matrix, the first column of which usually contains ones;  $\beta$  is a  $(k + 1) \times 1$  parameter vector, usually including the intercept; and  $\epsilon$  is an  $n \times 1$  vector of errors. Assuming that  $\mathbf{X}$  is of full-column rank, the least-squares regression coefficients are

Other

$$\log \left[ \frac{\hat{\mu}(\mathbf{x})}{1 - \hat{\mu}(\mathbf{x})} \right] = b_0 + b_1 x_1 + \cdots + b_k x_k$$

All of the hypotheses described in this chapter, and others that we have not discussed, can be expressed as general linear hypotheses, of the form  $H_0: \mathbf{L}\beta = \mathbf{c}$ , where  $\mathbf{L}$  is a  $q \times (k+1)$  hypothesis matrix of rank  $q$  containing prespecified constants, and  $\mathbf{c}$  is a prespecified  $q \times 1$  vector, most often containing zeros. Under  $H_0$ , the test statistic

Other

$$(5.4) \quad \frac{\widehat{\mu}(\mathbf{x})}{1 - \widehat{\mu}(\mathbf{x})} = \exp(b_0) \times \exp(b_1x_1) \times \cdots \times \exp(b_kx_k)$$

follows an  $F$ -distribution with  $q$  and  $n - (k + 1)$   $df$ ; is the estimated error variance.

Here are two nonstandard examples:

### **Example: Transaction Data**

In the transaction data introduced in [Section 5.1.1](#), there are two regressors,  $t1$  and  $t2$ , which count the numbers of two types of transactions in each of 261 bank branches. Because the coefficients for these two regressors have the same units, minutes per transaction, we may wish to test that the coefficients are equal. Using the `linearHypothesis()` function in the **car** package:

```
brief(trans.1)
```

|            | (Intercept) | t1    | t2     |
|------------|-------------|-------|--------|
| Estimate   | 144         | 5.462 | 2.0345 |
| Std. Error | 171         | 0.433 | 0.0943 |

Residual SD = 1143 on 258 df, R-squared = 0.909

```
linearHypothesis(trans.1, c(0, 1, -1))
```

Linear hypothesis test

Hypothesis:

$t1 - t2 = 0$

Model 1: restricted model

Model 2: time ~ t1 + t2

|   | Res.Df | RSS      | Df | Sum of Sq | F    | Pr(>F)  |
|---|--------|----------|----|-----------|------|---------|
| 1 | 259    | 3.96e+08 |    |           |      |         |
| 2 | 258    | 3.37e+08 | 1  | 59054218  | 45.2 | 1.1e-10 |

In this case, the hypothesis matrix consists of a single row,  $\mathbf{L} = (0, 1, -1)$ , contrasting the  $t1$  and  $t2$  coefficients, and the right-hand-side vector for the hypothesis is implicitly  $\mathbf{c} = (0)$ . The  $p$ -value for the test is tiny, suggesting that the two types of transactions *do not* take the same number of minutes on average.

We can also perform the test using the sandwich coefficient covariance matrix estimator in place of the usual coefficient covariance matrix estimator ([Section 5.1.1](#)):

```
linearHypothesis(trans.1, c(0, 1, -1), vcov=hccm)
```

Linear hypothesis test

Hypothesis:

$$t_1 - t_2 = 0$$

Model 1: restricted model

Model 2: time ~ t1 + t2

Note: Coefficient covariance matrix supplied.

|   | Res.Df | Df | F    | Pr(>F)  |
|---|--------|----|------|---------|
| 1 | 259    |    |      |         |
| 2 | 258    | 1  | 15.3 | 0.00012 |

Although the  $F$ -statistic for the test is now much smaller, the hypothesis is still associated with a small  $p$ -value, even without the assumption of constant error variance.

## Example: Davis's Regression of Measured on Reported Weight

For the linear regression fit to Davis's data in [Section 5.1.4](#), we can test the hypothesis of unbiased prediction of weight from reported weight by simultaneously testing that the intercept is equal to zero and the slope is equal to 1, formulating the test as a linear hypothesis:

```

diag(2) # order-2 identity matrix

[,1] [,2]
[1,]    1    0
[2,]    0    1

linearHypothesis(davis.mod, diag(2), c(0, 1))

Linear hypothesis test

Hypothesis:
(Intercept) = 0
repwt = 1

Model 1: restricted model
Model 2: weight ~ repwt



	Res.Df	RSS	Df	Sum of Sq	F	Pr(>F)
1	182	974				
2	180	914	2	59.7	5.88	0.0034


```

The hypothesis matrix  $\mathbf{L}$  is just an order-two identity matrix, constructed in R by `diag(2)`, while the right-hand-side vector is  $\mathbf{c} = (0, 1)'$ . Even though the regression coefficients are close to zero and 1, variability is sufficiently small to provide evidence against the hypothesis of unbiased prediction.

The `linearHypothesis()` function also has a more convenient interface for specifying hypotheses using the coefficient names rather than matrices and vectors. Here are alternative but equivalent ways of specifying the hypotheses considered above for the Transact and Davis regressions:

```
linearHypothesis(trans.1, "t1 = t2")
```

```
linearHypothesis(trans.1, "t1 - t2 = 0")
```

```
linearHypothesis(davis.mod, c ("(Intercept) = 0", "repwt = 1"))
```

For a hypothesis like the last one that includes the intercept, we must write "(Intercept)" (i.e., with parentheses), which is the name of the intercept in the coefficient vector.

## 5.4 Complementary Reading and References

- Hypothesis tests and  $p$ -values are widely misunderstood and even controversial topics. In 2016, the American Statistical Association (Wasserstein & Lazar, 2016) issued a general statement concerning their use. The statement also contains references to previous approaches to these topics, and many follow-up articles have appeared (Benjamin et al., 2017).
- Problems with multiple testing, in which tens, hundreds, or even thousands of tests are performed in a single problem, have attracted great interest in recent years. Ioannidis (2005) articulates an important prospective oriented toward medical research, and Bretz et al. (2011) provide a book-length treatment of simultaneous statistical inference, along with R software.

# 6 Fitting Generalized Linear Models

John Fox and Sanford Weisberg

In an article on the history of statistics, Efron (2003) traces the important trends in statistical analysis. At the beginning of the 20th century, statistical methodology and ideas were largely discipline specific, with methods for economics, agronomy, psychology, and so on. The first half of the century saw rapid change from an applications-oriented field to a field dominated by mathematical theory, perhaps best symbolized by the rise of methods that relied on the centrality of the normal distribution, until midcentury when most of the basic theory was worked out. The ascendance of the computer in the second half of the century meant that practical problems could be solved even if they lacked analytical solutions. In the last 40 years or so, statistics has become a synthesis of theory, computations, and practical applications.

According to Efron (2003), one of the central advances in statistics during the second half of the 20th century was the development of *generalized linear models (GLMs)* and their most important special case, *logistic regression*. The basic ideas appeared almost fully formed in a single paper by Nelder and Wedderburn (1972). The fundamental breakthrough was the extension of the elegant and well-understood linear model to problems in which the response is categorical or discrete rather than a continuous numeric variable. Although generalized linear models also permit continuous responses, including normal responses, the categorical response problem remains the most important application of GLMs. Their use is now routine, made possible by the confluence of the general theory laid out by Nelder and Wedderburn, progress in computation, and the need to solve real-world data analysis problems.

This chapter explains how generalized linear models are implemented in R. We also discuss statistical models, such as the multinomial and proportional-odds logit models for *polytomous* (multicategory) responses, that, while not generalized linear models in the strict sense, are closely related to GLMs.

- Although we assume some familiarity with GLMs, we review their structure in [Section 6.1](#), establishing terminology and notation.
- [Section 6.2](#) introduces `glm ()`, the standard R function for fitting generalized linear models, and a close relative of the `lm ()` function described in [Chapter 4](#).
- The most important GLMs in applications are for categorical and count data. [Section 6.3](#) discusses GLMs, chiefly the logistic-regression model, for binary data, where the response for each case takes one of two values, conventionally termed *success* and *failure*.

- [Section 6.4](#) takes up GLMs for binomial data, where the response for each case is the number or proportion of successes in a given number of “trials.”
- [Section 6.5](#) describes Poisson GLMs for count data.
- Poisson GLMs applied to counts in contingency tables are called loglinear models and are the subject of [Section 6.6](#).
- Models for multicategory responses are described in [Section 6.7](#) on the multinomial logit model, [Section 6.8](#) on logit models for nested dichotomies, and [Section 6.9](#) on the proportional-odds model for ordinal data.
- [Section 6.10](#) briefly develops several extensions to GLMs, including additional information about tests provided by the `Anova()`, `linearHypothesis()`, and `deltaMethod()` functions in the **car** package; gamma GLMs for positive continuous responses; and models for overdispersed count and binomial data.
- [Section 6.11](#) systematically describes the arguments to the `glm()` function.
- Finally, [Section 6.12](#) explains how maximum-likelihood estimates for GLMs are computed by iterated weighted least squares.

## 6.1 Review of the Structure of GLMs

The structure of a GLM is very similar to that of the linear models discussed in [Chapter 4](#). In particular, we have a response variable  $y$  and  $m$  predictors, and we are interested in understanding how the mean of  $y$  varies as the values of the predictors change.

A GLM consists of three components:

1. A *random component*, specifying the conditional or “error” distribution of the response variable,  $y$ , given the predictors. Nelder and Wedderburn (1972) considered conditional distributions that are from an *exponential family*. Both the binomial and Poisson distributions are in the class of exponential families, and so problems with categorical or discrete responses can be studied with GLMs. Other less frequently used exponential families are the gamma and the inverse-Gaussian distributions for strictly positive, positively skewed continuous data. Since the initial introduction of GLMs, the class of error distributions has been enlarged, as we will discuss later. The linear models in [Chapter 4](#) don’t require the specification of the random component, but if we assume that the response has a Gaussian (i.e., normal) distribution, then linear models are a special case of GLMs.
2. As in linear models, the  $m$  predictors in a GLM are translated into a vector of  $k + 1$  regressor variables,  $\mathbf{x} = (x_0, x_1, \dots, x_k)$ , possibly using contrast regressors for factors, polynomials, regression splines, transformations, and

interactions.<sup>1</sup> In this formulation,  $x_0 = 1$  is the constant regressor and is present when, as is typically the case, there's an intercept in the model. In a GLM, the response depends on the predictors only through a linear function of the regressors, called the *linear predictor*,

$$\eta(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$$

- The connection between the conditional mean  $E(y|\mathbf{x})$  of the response and the linear predictor  $\eta(\mathbf{x})$  in a linear model is direct,

$$E(y|\mathbf{x}) = \eta(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$$

and so the mean is equal to a linear combination of the regressors. This direct relationship is not appropriate for all GLMs because  $\eta(\mathbf{x})$  can take on any value in  $(-\infty, +\infty)$ , whereas, for example, the mean of a binary (i.e., 0/1) response variable must be in the interval  $(0, 1)$ . We therefore introduce an invertible *link function*  $g$  that translates from the scale of the mean response to the scale of the linear predictor. As is standard in GLMs, we write  $\mu(\mathbf{x}) = E(y|\mathbf{x})$  for the

conditional mean of the response. Then we have  $g[\mu(\mathbf{x})] = \eta(\mathbf{x})$

Reversing this relationship produces the *inverse-link function*,  $g^{-1}[\eta(\mathbf{x})] = \mu(\mathbf{x})$ . The inverse of the link function is sometimes called the *mean function* (or the *kernel mean function*).<sup>2</sup>

Standard link functions and their inverses are shown in [Table 6.1](#). The logit, probit, and complementary log-log links map the linear predictor to the interval  $(0, 1)$  and are used with binomial responses, where  $y$  represents the observed proportion and  $\mu$  the expected proportion of successes in  $N$  binomial trials, and  $\mu$  is the probability of success on a single trial.<sup>3</sup> For the probit link,  $\Phi$  is the standard-normal cumulative distribution function, and  $\Phi^{-1}$  is the standard-normal quantile function. An important special case of binomial data is *binary response data*, where each  $y$  is either a success or failure on a single binomial trial, and  $y$  therefore follows a Bernoulli distribution. In this case, it is usual to code  $y$  as either zero or 1, representing respectively “failure” and “success.”

<sup>1</sup> If you are unfamiliar with vector notation, simply think of  $\mathbf{x}$  as the collection of the regressors.

<sup>2</sup> The math in this last result can be daunting, so here is an explanation in words: A nonlinear transformation of the mean, given by the expression  $g[\mu(\mathbf{x})]$ , can be modeled as a linear combination of the regressors, given by  $\eta(\mathbf{x})$ .

[3](#) The notation here is potentially confusing: Binomial data comprise  $n$  binomial observations; the  $i$ th binomial observation,  $y^i$ , in turn consists of  $N^i$  binomial trials and is the proportion of successes in these trials.

**Table 6.1**  $\mu_{yx\eta\beta_0\beta_1x_1\beta_kx_k^{-1}exe^x}$

| Link                  | $\eta = g(\mu)$            | $\mu = g^{-1}(\eta)$      | Inverse Link        |
|-----------------------|----------------------------|---------------------------|---------------------|
| identity              | $\mu$                      | $\eta$                    | identity            |
| log                   | $\log \mu$                 | $e^\eta$                  | exponential         |
| inverse               | $\mu^{-1}$                 | $\eta^{-1}$               | inverse             |
| inverse square        | $\mu^{-2}$                 | $\eta^{-1/2}$             | inverse square root |
| square root           | $\sqrt{\mu}$               | $\eta^{1/2}$              | square              |
| logit                 | $\log \frac{\mu}{1 - \mu}$ | $\frac{1}{1 + e^{-\eta}}$ | logistic            |
| probit                | $\Phi(\mu)$                | $\Phi^{-1}(\eta)$         | normal quantile     |
| complementary log-log | $\log[-\log(1 - \mu)]$     | $1 - \exp[-\exp(\eta)]$   | —                   |

**Table 6.2**  $\phi\mu_{uxyN}$  [Table 6.3](#)

| Family           | Default Link | Range of $y$               | $\text{Var}(y \mathbf{x})$ |
|------------------|--------------|----------------------------|----------------------------|
| gaussian         | identity     | $(-\infty, +\infty)$       | $\phi$                     |
| binomial         | logit        | $\frac{0, 1, \dots, N}{N}$ | $\frac{\mu(1 - \mu)}{N}$   |
| poisson          | log          | $0, 1, 2, \dots$           | $\mu$                      |
| Gamma            | inverse      | $(0, \infty)$              | $\phi\mu^2$                |
| inverse.gaussian | $1/\mu^2$    | $(0, \infty)$              | $\phi\mu^3$                |

GLMs are typically fit to data by the method of maximum likelihood using the iteratively weighed least-squares procedure outlined in [Section 6.12](#). Denote the maximum-likelihood estimates of the regression parameters as  $b_0, b_1, \dots, b_k$  and the estimated value of the linear predictor as . The estimated mean of the response is .

The variance of distributions in an exponential family is a product of a positive

*dispersion* (or *scale*) parameter  $\phi$  and a function of the mean given the linear predictor:  $\text{Var}(y|\mathbf{x}) = \phi \times V[\mu(\mathbf{x})]$

The variances for the several exponential families are shown in the last column of [Table 6.2](#). For the binomial and Poisson distributions, the dispersion parameter  $\phi = 1$ , and so the variance depends only on  $\mu$ . For the Gaussian distribution,  $V[\mu(\mathbf{x})] = 1$ , and the variance depends only on the dispersion parameter  $\phi$ . For Gaussian data, it is usual to replace  $\phi$  by  $\sigma^2$ , as we have done for linear models in [Chapter 4](#). Only the Gaussian family has constant variance, and in all other GLMs, the conditional variance of  $y$  at  $\mathbf{x}$  depends on  $\mu(\mathbf{x})$ .

The *deviance*, based on the maximized value of the log-likelihood, provides a measure of the fit of a GLM to the data, much as the residual sum of squares does for a linear model. We write  $p[y; \mu(\mathbf{x}), \phi]$  for the probability-mass or probability-density function of a single response  $y$  given the regressors  $\mathbf{x}$ . Then the value of the log-likelihood evaluated at the maximum-likelihood estimates of the regression coefficients for fixed dispersion is

$$\log L_0 = \sum \log p[y_i; \hat{\mu}(\mathbf{x}_i), \phi]$$

where the sum is over the  $i = 1, \dots, n$  independent observations in the data.

Similarly, imagine fitting another model, called a *saturated model*, with one parameter for each of the  $n$  observations; under the saturated model, the estimated value of the mean response for each observation is just its observed value. Consequently, the log-likelihood for the saturated model is

$$\log L_1 = \sum \log p(y_i; y_i, \phi)$$

The *residual deviance* is defined as twice the difference between these log-likelihoods,  $D(\mathbf{y}; \hat{\boldsymbol{\mu}}) = 2(\log L_1 - \log L_0)$

Because the saturated model must fit the data at least as well as any other model, the deviance is never negative. The larger the deviance, the less well the model of interest matches the data. In families with a known value of the dispersion parameter  $\phi$ , such as the binomial and Poisson families, the deviance provides a basis for testing lack of fit of the model and for other tests that compare different specifications of the linear predictor. If  $\phi$  is estimated from the data, then the

*scaled deviance* is the basis for hypothesis tests. The *df* associated with the residual deviance is equal to the number of observations *n* minus the number of estimated regression parameters, including the intercept  $\beta^0$  if it is in the linear predictor.

## 6.2 The `glm()` Function in R

Most GLMs in R are fit with the `glm()` function. The most important arguments of `glm()` are `formula`, `family`, `data`, and `subset`. As with the `lm()` function discussed in [Chapter 4](#), the response variable and predictors are given in the model formula, and the `data` and `subset` arguments determine the data to which the model is fit. The `family` argument supplies a *family generator function*, which provides the random component of the model; additional optional arguments (usually just a `link` argument—see below) to the family generator function specify the link function for the model.

The family generator functions for the five standard exponential families are given in the *Family* column of [Table 6.2](#). All family names start with lowercase letters, except for the Gamma family, which is capitalized to avoid confusion with the `gamma()` function in R. Each family has its own *canonical link*, which is used by default if a link isn't given explicitly; in most cases, the canonical link is a reasonable choice. Also shown in the table are the range of the response and the variance function for each family.

[Table 6.3](#) displays the links available for each family generator function. Nondefault links are selected via a `link` argument to the family generator functions: for example, `binomial(link=probit)`. The `quasi()`, `quasibinomial()`, and `quasipoisson()` family generators do not correspond to exponential families; these family generators are described in [Section 6.10](#). If no `family` argument is supplied to `glm()`, then the `gaussian()` family, with the identity link, is assumed, resulting in a fit identical to that of `lm()`, albeit computed less efficiently—like using a sledgehammer to set a tack.

## 6.3 GLMs for Binary Response Data

We begin by considering data in which each case provides a *binary response*, say “success” or “failure,” the cases are independent, and the probability of success  $\mu(\mathbf{x})$  is the same for all cases with the same values  $\mathbf{x}$  of the regressors. In R, the

response may be a variable or an R expression that evaluates to 0 (failure) or 1 (success); a logical variable or expression, with FALSE representing failure and TRUE representing success; or a factor, typically, but not necessarily with two levels, in which case the first level is taken to represent failure and the others success. In [Section 6.4](#), we will consider the more general binomial responses, in which the response is the *number* of successes in one or more trials.

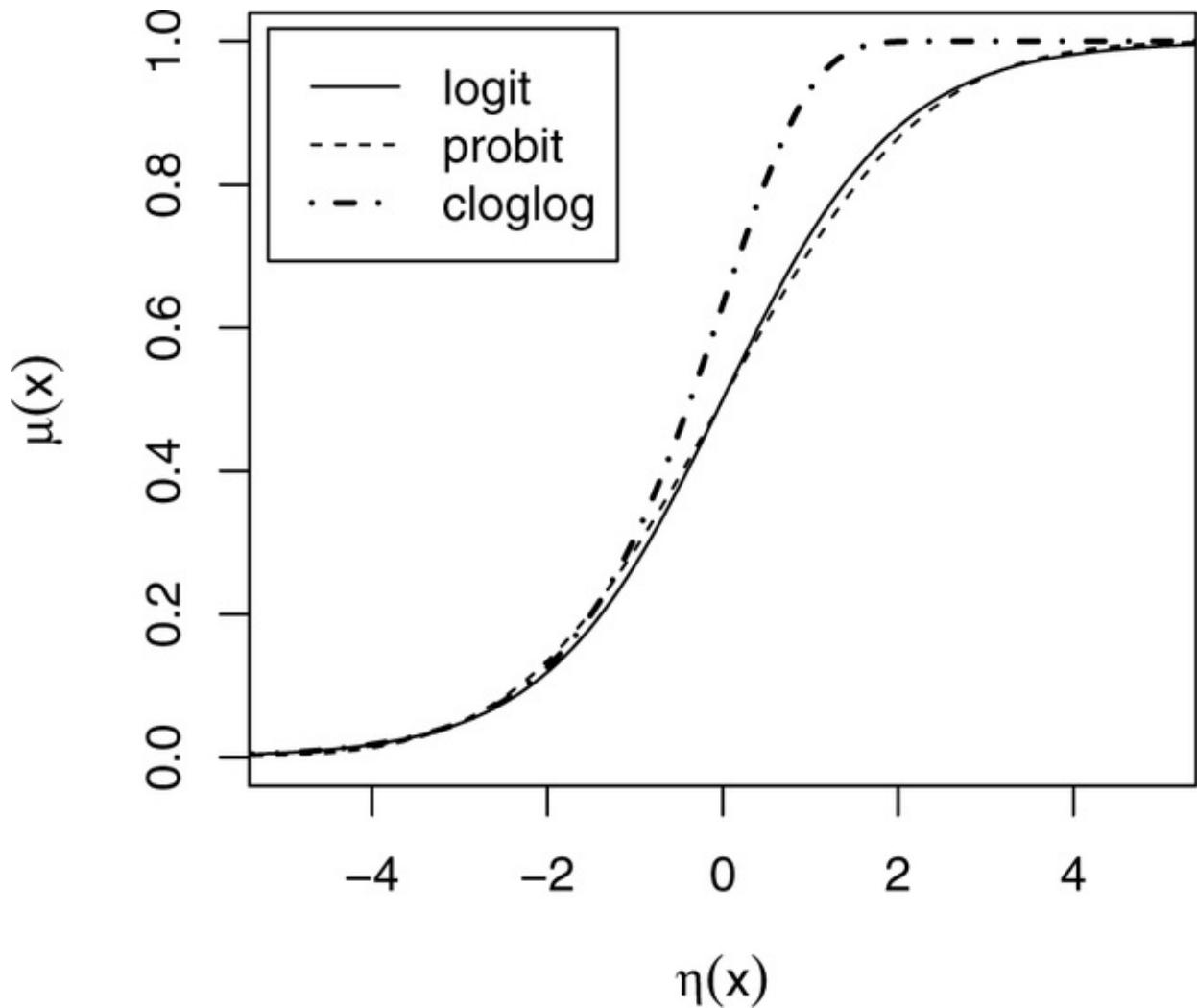
When the response is binary, we think of the mean function  $\mu(\mathbf{x})$  as the conditional probability that the response is a success given the values  $\mathbf{x}$  of the regressors. The most common link function used with binary response data is the logit link, for which

$$(6.1) \quad \log \left[ \frac{\mu(\mathbf{x})}{1 - \mu(\mathbf{x})} \right] = \eta(\mathbf{x})$$

**Table 6.3**

| family           | identity | inverse | log | logit | probit | cloglog | sqrt | 1/mu^2 |
|------------------|----------|---------|-----|-------|--------|---------|------|--------|
| gaussian         | •        | ○       | ○   |       |        |         |      |        |
| binomial         |          |         | ○   | •     | ○      | ○       |      |        |
| poisson          | ○        |         | •   |       |        |         | ○    |        |
| Gamma            | ○        | •       | ○   |       |        |         |      |        |
| inverse.gaussian | ○        | ○       | ○   |       |        |         |      | •      |
| quasi            | •        | ○       | ○   | ○     | ○      | ○       | ○    | ○      |
| quasibinomial    |          |         |     | •     | ○      | ○       |      |        |
| quasipoisson     | ○        |         | •   |       |        |         | ○    |        |

**Figure 6.1** Comparison of the logit, probit, and complementary log-log links. The probit link is rescaled to match the variance of the logistic distribution,  $\pi^2/3$ .



The quantity on the left of Equation 6.1 is called the *logit* or the *log-odds*, where the *odds* are the probability of success divided by the probability of failure.

$$\mu(\mathbf{x}) = \frac{1}{1 + \exp[-\eta(\mathbf{x})]}$$

Solving for  $\mu(\mathbf{x})$  gives the mean function,

Other link functions used less often include the probit and the complementary log-log links. These three links are drawn as functions of the linear predictor  $\eta(\mathbf{x})$  in [Figure 6.1](#). The logit and probit links are very similar except in the extreme tails, which aren't well resolved in a graph of the link functions, while the complementary log-log link has a different shape and is asymmetric, approaching  $\mu(\mathbf{x}) = 1$  more abruptly than  $\mu(\mathbf{x}) = 0$ ; the complementary log-log link may be appropriate for modeling rare phenomena, where the probability of success is small.<sup>4</sup>

[4](#) To model a binary response that approaches  $\mu(\mathbf{x}) = 0$  more abruptly than  $\mu(\mathbf{x}) = 1$ , simply reverse the meaning of “success” and “failure” and use the complementary log-log link with the redefined response.

The binomial model with the logit link is often called the *logistic-regression model* because the inverse of the logit link, in Equation 6.1 and [Table 6.1](#), is the logistic function. The names *logit regression* and *logit model* are also used.

### 6.3.1 Example: Women’s Labor Force Participation

To illustrate logistic regression, we turn to an example from Long (1997), which draws on data from the 1976 U.S. Panel Study of Income Dynamics and in which the response variable is married women’s labor force participation. The data were originally used in a different context by Mroz (1987), and the same data appear in Berndt (1991) as an exercise in logistic regression. The data are in the data frame `Mroz` in the **carData** package, which loads along with the **car** package:

```
library("car")
summary(Mroz)
```

|         | lfp           | k5             | k618           | age |
|---------|---------------|----------------|----------------|-----|
| no :325 | Min. :0.000   | Min. :0.00     | Min. :30.0     |     |
| yes:428 | 1st Qu.:0.000 | 1st Qu.:0.00   | 1st Qu.:36.0   |     |
|         | Median :0.000 | Median :1.00   | Median :43.0   |     |
|         | Mean :0.238   | Mean :1.35     | Mean :42.5     |     |
|         | 3rd Qu.:0.000 | 3rd Qu.:2.00   | 3rd Qu.:49.0   |     |
|         | Max. :3.000   | Max. :8.00     | Max. :60.0     |     |
|         | wc            | hc             | lwg            | inc |
| no :541 | no :458       | Min. :-2.054   | Min. :-0.029   |     |
| yes:212 | yes:295       | 1st Qu.: 0.818 | 1st Qu.:13.025 |     |
|         |               | Median : 1.068 | Median :17.700 |     |
|         |               | Mean : 1.097   | Mean :20.129   |     |
|         |               | 3rd Qu.: 1.400 | 3rd Qu.:24.466 |     |
|         |               | Max. : 3.219   | Max. :96.000   |     |

The definitions of the variables in the Mroz data set are shown in [Table 6.4](#). With the exception of lwg, these variables are straightforward. The log of each woman's estimated wage rate, lwg, is based on her actual earnings if she is in the labor force; if the woman is not in the labor force, then this variable is imputed (i.e., filled in) as the predicted value from a regression of log wages on the other predictors for women in the labor force. As we will see in [Section 8.6.3](#), this definition of expected earnings creates a problem for the logistic regression.

**Table 6.4**

| Variable | Description                           | Remarks         |
|----------|---------------------------------------|-----------------|
| lfp      | wife's labor force participation      | factor: no, yes |
| k5       | number of children ages 5 and younger | 0–3, few 3s     |
| k618     | number of children ages 6 to 18       | 0–8, few > 5    |
| age      | wife's age in years                   | 30–60           |
| wc       | wife's college attendance             | factor: no, yes |
| hc       | husband's college attendance          | factor, no, yes |
| lwg      | log of wife's estimated wage rate     | see text        |
| inc      | family income excluding wife's income | \$1000s         |

The variable lfp is a factor with two levels, and if we use this variable as the response, then the first level, "no", corresponds to failure (zero) and the second level, "yes", to success (1):

```
mroz.mod <- glm (lfp ~ k5 + k618 + age + wc + hc + lwg + inc,
family=binomial, data=Mroz)
```

The only features that differentiate this command from fitting a linear model are the change of function from lm () to glm () and the addition of the family argument, which is set to the binomial family generator function. The first argument to glm () is the model formula, the right-hand side of which specifies the linear predictor for the logistic regression, not the mean function directly, as it did in linear regression. Because the link function is not given explicitly, the default logit link is used; the command is therefore equivalent to

```
mroz.mod <- glm (lfp ~ k5 + k618 + age + wc + hc + lwg + inc,
family=binomial (link=logit), data=Mroz)
```

Applying the S () function to a logistic-regression model produces results very similar to those for a linear model, but by default we get an additional table of exponentiated coefficients and their confidence limits:

**S (mroz.mod)**

Call: glm(formula = lfp ~ k5 + k618 + age + wc + hc + lwg + inc, family = binomial, data = Mroz)

Coefficients:

|             | Estimate | Std. Error | z value | Pr(> z ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 3.18214  | 0.64438    | 4.94    | 7.9e-07  |
| k5          | -1.46291 | 0.19700    | -7.43   | 1.1e-13  |
| k618        | -0.06457 | 0.06800    | -0.95   | 0.34234  |
| age         | -0.06287 | 0.01278    | -4.92   | 8.7e-07  |
| wcyes       | 0.80727  | 0.22998    | 3.51    | 0.00045  |
| hcyes       | 0.11173  | 0.20604    | 0.54    | 0.58762  |
| lwg         | 0.60469  | 0.15082    | 4.01    | 6.1e-05  |
| inc         | -0.03445 | 0.00821    | -4.20   | 2.7e-05  |

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 1029.75 on 752 degrees of freedom  
Residual deviance: 905.27 on 745 degrees of freedom

| logLik  | df | AIC    | BIC    |
|---------|----|--------|--------|
| -452.63 | 8  | 921.27 | 958.26 |

Number of Fisher Scoring iterations: 4

Exponentiated Coefficients and Confidence Bounds

|             | Estimate | 2.5 %   | 97.5 %   |
|-------------|----------|---------|----------|
| (Intercept) | 24.09828 | 6.93772 | 87.03479 |
| k5          | 0.23156  | 0.15553 | 0.33707  |
| k618        | 0.93747  | 0.82004 | 1.07108  |
| age         | 0.93907  | 0.91548 | 0.96258  |
| wcyes       | 2.24179  | 1.43475 | 3.53876  |
| hcyes       | 1.11821  | 0.74677 | 1.67664  |
| lwg         | 1.83069  | 1.36892 | 2.47682  |
| inc         | 0.96614  | 0.95028 | 0.98140  |

The Wald tests for the regression coefficients, given by the ratios of the coefficient estimates to their standard errors, are labeled as z values: The large-sample reference distribution for the test statistics is the standard-normal distribution, not the *t*-distribution used with linear models. Thus, the predictors k5, age, wc, lwg, and inc have estimated coefficients associated with very small *p*-values, while the coefficients for k618 and hc are associated with large *p*-values.

The dispersion parameter, fixed to  $\phi = 1$  for the binomial family, is also shown in the output. Additional output includes the null deviance and *df*, which are for a model with all parameters apart from the intercept set to zero; the residual deviance and *df* for the model actually fit to the data; and the maximized log-likelihood, model degrees of freedom (number of estimated coefficients), *Akaike Information Criterion* (or *AIC*), and *Bayesian Information Criterion* (or *BIC*). The AIC and BIC are alternative measures of fit sometimes used for model selection (see, e.g., Fox, 2016, Section 22.1). The number of iterations required to obtain the maximum-likelihood estimates is also printed.<sup>5</sup>

[5](#) The iterative algorithm employed by `glm()` to maximize the likelihood is described in [Section 6.12](#).

The estimated logistic-regression model is given by

$$\log \left[ \frac{\hat{\mu}(\mathbf{x})}{1 - \hat{\mu}(\mathbf{x})} \right] = b_0 + b_1 x_1 + \cdots + b_k x_k$$

If we exponentiate both sides of this equation, we get

$$\frac{\hat{\mu}(\mathbf{x})}{1 - \hat{\mu}(\mathbf{x})} = \exp(b_0) \times \exp(b_1 x_1) \times \cdots \times \exp(b_k x_k)$$

where the left-hand side of the equation,  $\mu(\mathbf{x}) / [1 - \mu(\mathbf{x})]$ , gives the *fitted odds* of success, the fitted probability of success divided by the fitted probability of failure. Exponentiating the model removes the logarithms and changes from a model that is additive in the log-odds scale to one that is multiplicative in the odds scale. For example, increasing the age of a woman by one year, holding the other predictors constant, *multiplies* the fitted odds of her being in the workforce by  $\exp(b_3) = \exp(-0.06287) = 0.9391$ —that is, reduces the odds of working by

$100(1 - 0.94) = 6\%$ . To take another example, compared to a woman who did not attend college, a college-educated woman with all other predictors the same has fitted odds of working about 2.24 times higher, with a 95% confidence interval from 1.43 to 3.54. The exponentials of the coefficient estimates are called *risk factors* or *odds ratios*.

The `S()` and `Confint()` functions (which can be called directly to produce confidence intervals for the coefficients on the logit scale) provide confidence intervals for GLMs based on profiling the log-likelihood rather than on the Wald statistics used for linear models (Venables & Ripley, 2002, [Section 8.4](#)).<sup>6</sup> Confidence intervals for generalized linear models based on the log-likelihood take longer to compute but tend to be more accurate than those based on the Wald statistic. Even before exponentiation, the log-likelihood-based confidence intervals need not be symmetric about the estimated coefficients.

[6](#) A linear model with normally distributed errors has a quadratic log-likelihood, and so confidence intervals based on the Wald statistic are identical to those based directly on the likelihood; this is not true generally for GLMs.

### 6.3.2 Example: Volunteering for a Psychological Experiment

Cowles and Davis (1987) collected data on the willingness of students in an introductory psychology class to volunteer for a psychological experiment. The data are in the Cowles data set in the `carData` package:

### ***brief(Cowles)***

```
1421 x 4 data.frame (1416 rows omitted)
  neuroticism extraversion sex volunteer
  [i]      [i]   [f]      [f]
1       16    13 female no
2        8    14 male  no
3        5    16 male  no
. . .
1420     19    20 female yes
1421     15    20 male  yes
```

```
sum(Cowles$volunteer == "yes") # number yes
```

```
[1] 597
```

The data set contains several variables:

- The personality dimension neuroticism, a numeric variable with integer scores on a scale potentially ranging from zero to 24
- The personality dimension extraversion, also a numeric variable with a potential range of zero to 24
- The factor sex, with levels "female" and "male"
- The factor volunteer, with levels "no" and "yes"; thus, 597 of the 1421 students volunteered

The researchers expected volunteering to depend on sex and on the interaction of the personality dimensions. They consequently formulated the following logistic-regression model for their data, which includes a linear-by-linear interaction (see [Section 4.6.4](#)) between neuroticism and extraversion:

```

cowles.mod <- glm(volunteer ~ sex + neuroticism*extraversion,
  data=Cowles, family=binomial)
brief(cowles.mod, pvalues=TRUE)

              (Intercept) sexmale neuroticism extraversion
Estimate      -2.36e+00 -0.2472     0.11078   1.67e-01
Std. Error     5.01e-01  0.1116     0.03765   3.77e-02
Pr(>|t|)      2.55e-06  0.0268     0.00326   9.75e-06
exp(Estimate)  9.46e-02  0.7810     1.11715   1.18e+00
                neuroticism:extraversion
Estimate        -0.00855
Std. Error       0.00293
Pr(>|t|)        0.00355
exp(Estimate)  0.99148

```

Residual deviance = 1897 on 1416 df

We use the `brief()` function to obtain a compact summary of the fitted logistic regression partly because the interaction makes it difficult to interpret the results directly from the estimated coefficients. We can see that males are less likely than equivalent females to volunteer for the experiment and that the coefficient for the sex dummy regressor is associated with the moderately small  $p$ -value of .0268. The coefficient for the interaction regressor has a very small  $p$ -value. We pursue the interpretation of this example in the next section on effect plots and in [Section 6.3.4](#) on analysis of deviance for logistic regression.

### 6.3.3 Predictor Effect Plots for Logistic Regression

The **effects** package, introduced in [Section 4.3](#) for linear models, can draw predictor effect plots (and other effect plots) for generalized linear models, including logistic regression. For example, predictor effect plots for the logistic-regression model that we fit to Mroz's labor force participation data (in [Section 6.3.1](#)), are shown in [Figure 6.2](#):

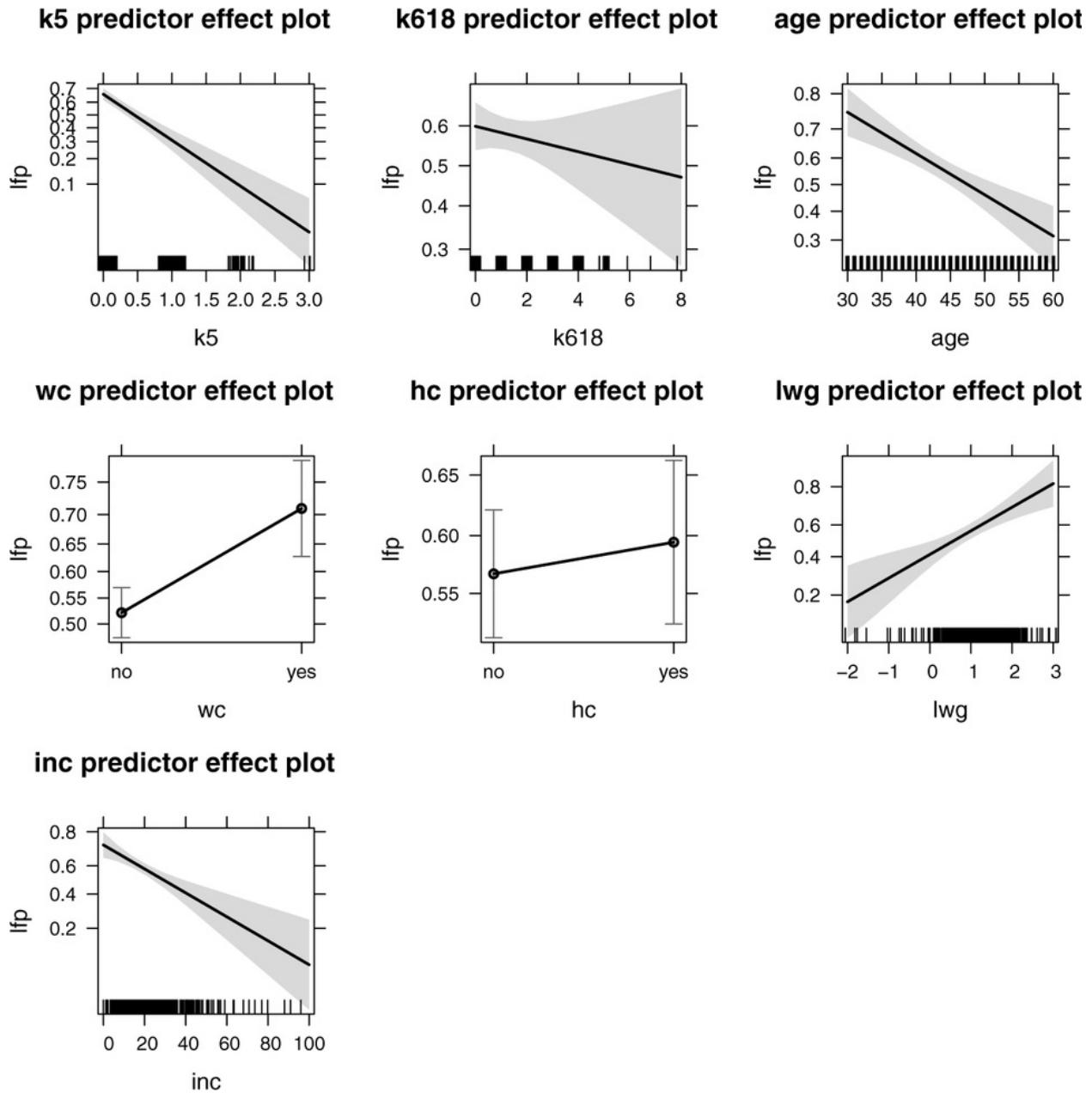
```
library("effects")
```

```
plot (predictorEffects (mroz.mod))
```

The default vertical axis of each plot is drawn on the scale of the linear predictor where the structure of the GLM is linear, but the tick marks for the axis are labeled on the more familiar scale of the mean of the response variable. For this logistic-regression model, the vertical axis is therefore in the log-odds or logit scale, and the tick marks are labeled by converting the logits to the more familiar probability scale. Tick marks that would be equally spaced on the probability scale are not equally spaced in the graph, but the lines representing the fitted model for numeric predictors are straight. As for a linear model, pointwise 95% confidence intervals and envelopes are shown in the predictor effect plots, and rug-plots displaying the marginal distributions of numeric predictors appear at the bottom of the corresponding panels of [Figure 6.2](#). In each panel, the other predictors in the model are held to typical values, as for a linear model. In this example, the effect plots simply visualize the estimated regression coefficients and display their uncertainty.

Effect plots are much more useful for models with transformed predictors, polytomous factors, interactions, and polynomial or spline regressors. For example, the logistic-regression model that we fit to Cowles and Davis's data on volunteering for a psychological experiment in [Section 6.3.2](#) includes a main effect of the predictor sex that is easily interpreted from its coefficient but also a linear-by-linear interaction between the numeric predictors neuroticism and extraversion. Effect plots for the interacting predictors, shown in [Figure 6.3](#), are produced by the command:

**Figure 6.2** Predictor effect plots for the logistic-regression model fit to Mroz's labor force participation data.



```
plot (predictorEffects (cowles.mod, ~ neuroticism + extraversion,
xlevels=list (neuroticism=seq (0, 24, by=8),
extraversion=seq (0, 24, by=8))), lines=list (multiline=TRUE))
```

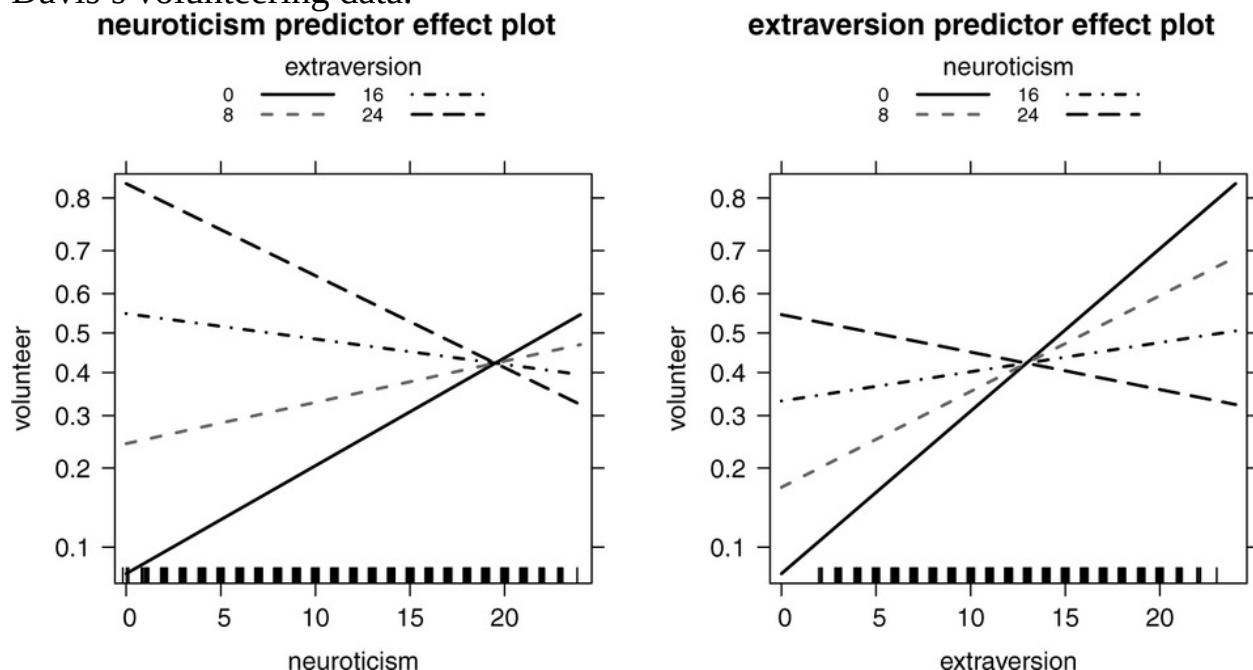
We opt to display the effect plots as multiline graphs by setting `lines=list (multiline= TRUE)` in the call to `plot ()` and use the `xlevels` argument to `predictorEffects ()` to set each personality dimension to four values equally

spaced over its potential range.<sup>7</sup> See help ("predictorEffects") and help ("plot.eff") for more information about these and other optional arguments.

<sup>7</sup> The default is to use five values for each numeric predictor, approximately equally spaced over its range. In the case of extraversion, we extrapolate slightly beyond the observed range of the data, from 2 to 23.

The two panels of [Figure 6.3](#) show two views of the same interaction between neuroticism and extraversion: In the panel at the left, we see that volunteering is positively related to neuroticism when extraversion is low and negatively related to neuroticism when extraversion is high. In the panel at the right, volunteering is positively related to extraversion at most displayed levels of neuroticism but slightly negatively related to extraversion at the highest level of neuroticism. The linear-by-linear interaction constrains the lines in each panel to intersect at a common point, although the point of intersection need not be in the range of the horizontal axis, as it is in this example.

**Figure 6.3** Predictor effect plots for the interacting personality dimensions neuroticism and extraversion in the logistic-regression model fit to Cowles and Davis's volunteering data.



### 6.3.4 Analysis of Deviance and Hypothesis Tests for Logistic Regression

## Model Comparisons and Sequential Tests

As is true for linear models (see [Section 5.3.2](#)), the `anova()` function can be used to compare two or more nested GLMs that differ by one or more terms. For example, we can remove the two predictors that count the number of children from the logistic-regression model fit to the Mroz data, to test the hypothesis that labor force participation does not depend on the number of children versus the alternative that it depends on the number of young children, the number of older children, or both:

```
mroz.mod.2 <- update(mroz.mod, . ~ . - k5 - k618)
anova(mroz.mod.2, mroz.mod, test="Chisq")
```

Analysis of Deviance Table

|           |   |     |          |              |
|-----------|---|-----|----------|--------------|
| Model 1:  | lfp ~ age + wc + hc + lwg + inc             |     |          |              |
| Model 2:  | lfp ~ k5 + k618 + age + wc + hc + lwg + inc |     |          |              |
| Resid. Df | Resid. Dev                                  | Df  | Deviance | Pr(>Chi)     |
| 1         | 747   | 972 |          |              |
| 2         | 745   | 905 | 2        | 66.5 3.7e-15 |

The likelihood-ratio test statistic is the change in deviance between the two fitted models, and the  $p$ -value for the test is computed by comparing the test statistic to the  $\chi^2$  distribution with  $df$  equal to the change in degrees of freedom for the two models. For the example, the change in deviance is 66.5 on 2  $df$ , reflecting the two regressors removed from the model; when the test statistic is compared to the  $\chi^2(2)$  distribution, we get a  $p$ -value that is effectively zero. That the probability of a woman's participation in the labor force depends on the number of children she has is, of course, unsurprising. Because this test is based on deviances rather than variances, the output is called an *analysis of deviance* table.

We can use this model comparison approach to test for the interaction between the predictors neuroticism and extraversion in Cowles and Davis's logistic regression, removing the interaction from the model:

```

brief(cowles.mod.0 <- update(cowles.mod,
 $\cdot \sim \cdot - \text{neuroticism:extraversion})$ 

(Intercept) sexmale neuroticism extraversion
Estimate      -1.116   -0.235     0.00636    0.0663
Std. Error      0.249    0.111     0.01136    0.0143
exp(Estimate)    0.327    0.790     1.00638    1.0686

```

Residual deviance = 1906 on 1417 df

```
anova(cowles.mod.0, cowles.mod, test="Chisq")
```

Analysis of Deviance Table

Model 1: volunteer ~ sex + neuroticism + extraversion

Model 2: volunteer ~ sex + neuroticism \* extraversion

| Resid. | Df   | Resid. | Dev  | Df   | Deviance | Pr(>Chi) |
|--------|------|--------|------|------|----------|----------|
| 1      | 1417 |        | 1906 |      |          |          |
| 2      | 1416 | 1897   | 1    | 8.62 | 0.0033   |          |

The  $p$ -value for the hypothesis of no interaction is small,  $p = .0033$ . Unlike in linear models, likelihood-ratio tests are *not* identical to Wald tests of the same hypotheses. Although they are asymptotically equivalent, the two approaches to testing can produce quite different  $p$ -values in some circumstances. This isn't the case for our example, where the  $p$ -value for the Wald test of the interaction coefficient is very similar,  $p = .0035$  (on page 282). The likelihood-ratio test is generally more reliable than the Wald test.

The `anova()` function does not by default compute hypothesis tests for a GLM. To obtain likelihood-ratio  $\chi^2$  tests for binary regression models and other GLMs, we have to include the argument `test="Chisq"`. For GLMs with a dispersion parameter estimated from the data (discussed, for example, in [Section 6.10.4](#)), specifying `test="F"` produces  $F$ -tests in the analysis-of-deviance table.

As for linear models (see [Section 5.3.3](#)), the `anova()` function can be used to compute a Type I or sequential analysis-of-deviance table for a GLM, and as for linear models, these tables are rarely useful. We instead recommend using the `Anova()` function to compute Type II tests, as described immediately below.

## Type II Tests and the *Anova ()* Function

The *Anova ()* function in the **car** package can be used for GLMs as well as for linear models. For the logistic-regression model fit to the Mroz data, we get:

### ***Anova (mroz.mod)***

Analysis of Deviance Table (Type II tests)

Response: lfp

|      | LR | Chisq | Df | Pr(>Chisq) |
|------|----|-------|----|------------|
| k5   |    | 66.5  | 1  | 3.5e-16    |
| k618 |    | 0.9   | 1  | 0.34204    |
| age  |    | 25.6  | 1  | 4.2e-07    |
| wc   |    | 12.7  | 1  | 0.00036    |
| hc   |    | 0.3   | 1  | 0.58749    |
| lwg  |    | 17.0  | 1  | 3.7e-05    |
| inc  |    | 19.5  | 1  | 1.0e-05    |

Each line of the analysis-of-deviance table provides a likelihood-ratio test based on the change in deviance comparing two models. These tests are analogous to the corresponding Type II tests for a linear model ([Section 5.3.4](#)). For an additive model in which all terms have a single degree of freedom—as is the case in the current example—the Type II likelihood-ratio statistics, comparing models that drop each term in turn to the full model, test the same hypotheses as the Wald statistics in the *S ()* output for the model (given on page 280). As we explained, in GLMs, likelihood-ratio and Wald tests are not identical, but as is often the case, they produce similar results in this example.

The logit model fit to the Cowles data includes an interaction term, and so the Type II tests for the main effects are computed without the interaction term in the model:

## Anova (cowles.mod)

Analysis of Deviance Table (Type II tests)

Response: volunteer

|                          | LR | Chisq | Df | Pr(>Chisq) |
|--------------------------|----|-------|----|------------|
| sex                      |    | 4.92  | 1  | 0.0266     |
| neuroticism              |    | 0.31  | 1  | 0.5753     |
| extraversion             |    | 22.14 | 1  | 2.5e-06    |
| neuroticism:extraversion |    | 8.62  | 1  | 0.0033     |

The *p*-value for the interaction is identical to the value we computed directly by model comparison using the `anova()` function because exactly the same models are being compared.

The `Anova()` function is considerably more flexible than is described here, including options to compute Type II Wald tests, even for models with multiple-*df* terms and interactions; to compute *F*-tests for models with an estimated dispersion parameter; and to compute Type III tests.<sup>8</sup>

<sup>8</sup> Additional details are provided in [Section 6.10.1](#).

## Other Hypothesis Tests

The `linearHypothesis()` and `deltaMethod()` functions, described for linear models in [Sections 5.3.5](#) and [5.1.4](#), also work for generalized linear models. The `linearHypothesis()` function computes general Wald chi-square tests. For example, we can test the hypothesis that both the `k5` and `k618` predictors in our logistic regression for the `Mroz` data have zero coefficients, a hypothesis for which we earlier computed a likelihood-ratio test:

```
linearHypothesis(mroz.mod, c("k5", "k618"))
```

Linear hypothesis test

Hypothesis:

k5 = 0

k618 = 0

Model 1: restricted model

Model 2: lfp ~ k5 + k618 + age + wc + hc + lwg + inc

|   | Res.Df | Df | Chisq | Pr(>Chisq) |
|---|--------|----|-------|------------|
| 1 | 747    |    |       |            |
| 2 | 745    | 2  | 55.2  | 1.1e-12    |

In this case, the Wald and likelihood-ratio tests once again produce similar results, with very small *p*-values.

Similarly, to test the linear hypothesis that the k5 and k618 predictors have *equal* coefficients,

```
linearHypothesis(mroz.mod, "k5 = k618")
```

Linear hypothesis test

Hypothesis:

k5 - k618 = 0

Model 1: restricted model

Model 2: lfp ~ k5 + k618 + age + wc + hc + lwg + inc

|   | Res.Df | Df | Chisq | Pr(>Chisq) |
|---|--------|----|-------|------------|
| 1 | 746    |    |       |            |
| 2 | 745    | 1  | 49.5  | 2e-12      |

This hypothesis too can easily be rejected.

We illustrate the use of the deltaMethod () function in [Section 6.10.2](#), in an application to a gamma GLM.

### 6.3.5 Fitted and Predicted Values

As for linear models, the predict () function can be used to get fitted and predicted values for GLMs. By default, predict () returns the estimated linear predictor for each observation. For example, for the logit model that we fit to the Mroz data:

```
head(predict(mroz.mod)) # first few values  
1           2           3           4           5           6  
0.063337  0.693821 -0.174106  0.672295  0.677721  0.388719
```

These are the fitted log-odds of success for the first six cases. To get fitted probabilities (i.e., fitted values on the scale of the response) rather than fitted logits, we use the argument type="response":

```
head(predict(mroz.mod, type="response"))  
1           2           3           4           5           6  
0.51583  0.66682  0.45658  0.66202  0.66323  0.59597
```

Fitted values on the scale of the response can also be obtained with the fitted () function.

Finally, we can use the predict () function to compute predicted values for arbitrary combinations of predictor values. For example, let's compare predicted probabilities of volunteering for women and men at moderate levels of neuroticism and extraversion based on the logit model fit to the Cowles data:

```

predict.data <- data.frame(sex=c("female", "male"),
                           neuroticism=rep(12, 2), extraversion=rep(12, 2))
predict.data$p.volunteer <- predict(cowles.mod,
                                      newdata=predict.data, type="response")
predict.data

      sex neuroticism extraversion p.volunteer
1 female           12             12     0.43570
2 male            12             12     0.37618

```

## 6.4 Binomial Data

In binomial response data, the response variable  $y_i$  for each case  $i$  is the number of successes in a fixed number  $N_i$  of independent trials, each with the same probability of success. Binary regression is a limiting case of binomial regression with all the  $N_i = 1$ .

For binomial data, it is necessary to specify not only the number of successes but also the number of trials for each of the  $n$  binomial observations. We can do this in R by setting the response variable in the model formula to a matrix with two columns, the first giving the number of successes  $y$  and the second the number of failures,  $N - y$  (as opposed to the number of trials). Alternatively for binomial data, the response can be the *proportion* (rather than a *count*) of successes for each observation,  $y/N$ , in which case the number of trials,  $N$ , is specified by the *weights* argument to `glm()`. We find this second approach confusing because the *weights* argument has another meaning for most other generalized linear models (see [Section 6.11.1](#)).

**Table 6.5** Source of data:

| <i>Perceived Closeness</i> | <i>Intensity of Preference</i> | <i>Turnout</i> |              | <i>Logit</i>  |
|----------------------------|--------------------------------|----------------|--------------|---|
|                            |                                | Voted          | Did Not Vote | $\log\left(\frac{\text{Voted}}{\text{Did Not Vote}}\right)$ |
| One-sided                  | Weak                           | 91             | 39           | 0.847   |
|                            | Medium                         | 121            | 49           | 0.904   |
|                            | Strong                         | 64             | 24           | 0.981   |
| Close                      | Weak                           | 214            | 87           | 0.900   |
|                            | Medium                         | 284            | 76           | 1.318   |
|                            | Strong                         | 201            | 25           | 2.084   |

Regardless of how the model is specified, `glm()` considers the response in a binomial GLM to be  $y/N$ , the observed proportion of successes. The mean of  $y/N$ ,  $\mu(\mathbf{x})$ , consequently has the same interpretation as in binary regression and represents the probability of success on one trial when the regressors are equal to  $\mathbf{x}$ .

To illustrate, the data in [Table 6.5](#) are from *The American Voter* (Campbell, Converse, Miller, & Stokes, 1960), a classic study of voting in the 1956 U.S. presidential election. The body of the table, in the columns labeled “voted” and “did not vote,” shows frequency counts. The rows correspond to the six possible combinations of the two predictor factors, and the two columns correspond to the response, the number  $y$  of “successes,” or voters, and the number  $N - y$  of “failures,” or non-voters. There are therefore  $n = 6$  binomial observations. [Table 6.5](#) is a three-way *contingency table* for the categorical variables perceived closeness of the election, intensity of partisan preference, and voter turnout, the last of which is treated as the response variable.

For small data sets like this one, typing data directly into an R script or an R Markdown document is not particularly hard, as described in [Section 2.1.2](#):

```

Campbell <- data.frame(
  closeness = factor(rep(c("one.sided", "close"), c(3, 3)),
  levels=c("one.sided", "close")),
  preference = factor(rep(c("weak", "medium", "strong"), 2),
  levels=c("weak", "medium", "strong")),
  voted = c(91, 121, 64, 214, 284, 201),
  did.not.vote = c(39, 49, 24, 87, 76, 25)
)
Campbell
  1 one.sided      weak     91      39
  2 one.sided    medium    121      49
  3 one.sided    strong     64      24
  4   close       weak    214      87
  5   close    medium    284      76
  6   close    strong    201      25

```

We begin analysis of these data in our usual fashion by visualizing the data. With tabular data like these, we can start by fitting a model, called a *saturated model*, that has as many parameters in the linear predictor as there are cells in the data:

```

campbell.mod <- glm (cbind (voted, did.not.vote) ~
  closeness*preference, family=binomial, data=Campbell)

```

The model fits an intercept, one dummy regressor for the main effect of closeness, two dummy regressors for the main effect of preference, and two regressors for the interaction of the two factors—that is, six parameters fit to the six binomial observations. It should not be surprising that the fitted values for this model perfectly reproduce the observed values, and therefore the residuals are all equal to zero:

```
predict(campbell.mod)
```

| 1       | 2       | 3       | 4       | 5       | 6       |
|---------|---------|---------|---------|---------|---------|
| 0.84730 | 0.90397 | 0.98083 | 0.90007 | 1.31824 | 2.08443 |

```
residuals(campbell.mod)
```

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

The fitted values computed by predict () match the logits (log-odds) shown in the last column of [Table 6.5](#) because the default link function for binomial data is the logit link. The residuals are the scaled differences between the observed logits and the fitted logits, and so they are all zero for the saturated model.

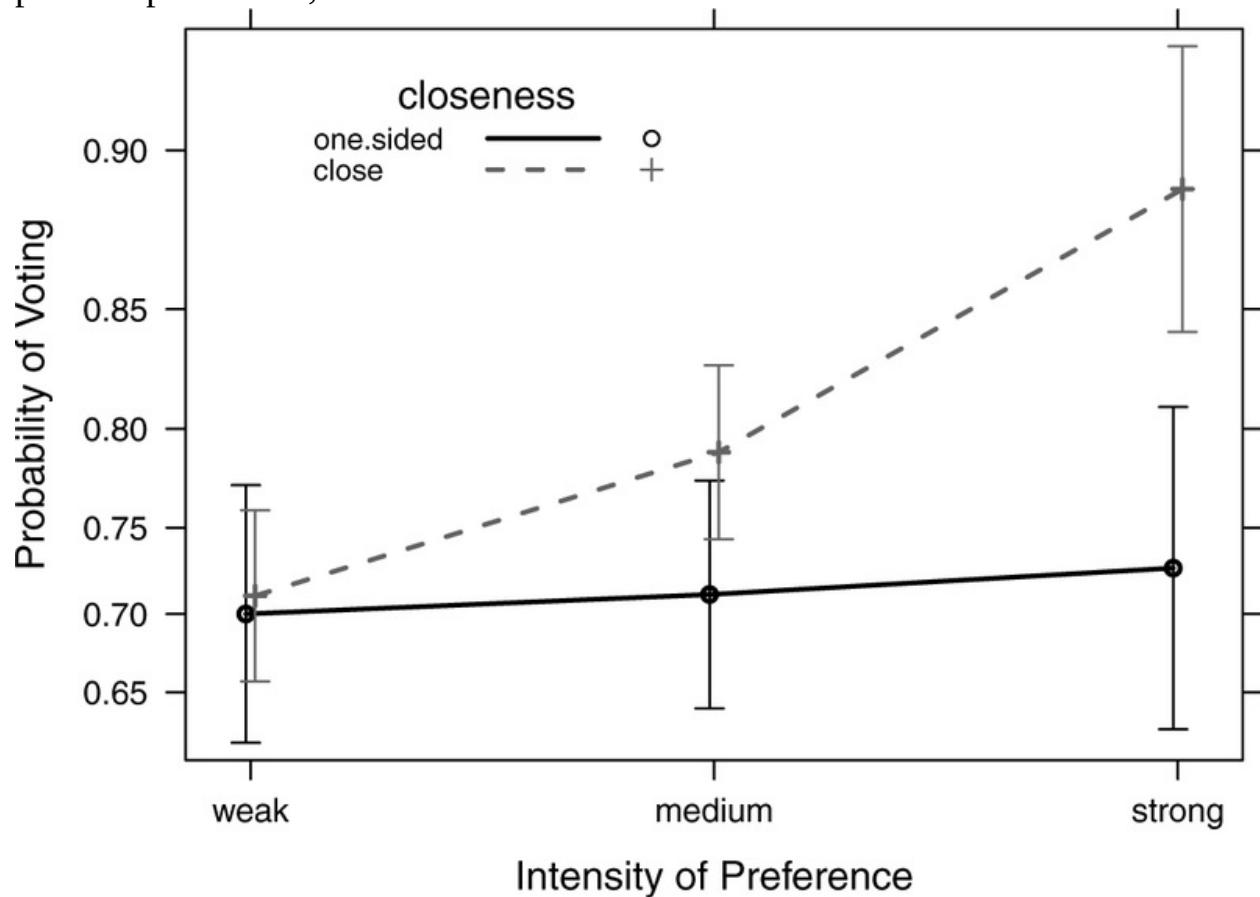
The correspondence of the fitted values to the directly observed data for the saturated model allows us to use an effect plot for the model to visualize the data on the logit scale ([Figure 6.4](#)):

```
plot(predictorEffects(campbell.mod, ~ preference),
  main="", confint=list(style="bars"),
  lines=list(multiline=TRUE),
  xlab="Intensity of Preference",
  ylab="Probability of Voting",
  lattice=list(key.args=list(x=0.1, y=0.95, corner=c(0, 1))))
```

This plot clearly shows that when closeness = "one.sided", the log-odds of voting hardly vary as intensity of preference changes. In contrast, when closeness = "close", the log-odds increase with intensity of preference. For a model that does not include the interaction between closeness and preference to be useful, the two lines in this graph should be approximately parallel, which is apparently not the case here. This observation suggests that the closeness:intensity interaction is likely required to model these data, and so the saturated model is appropriate. The error bars on the graph, for pointwise 95% confidence intervals, show the substantial sampling variability of each of the estimated log-odds. The confidence intervals can be computed despite fitting a saturated model because of the moderately large binomial sample sizes at each combination of factor levels.

**Figure 6.4** Voter turnout by perceived closeness of the election and intensity of

partisan preference, for the *American Voter* data.



The `emmeans()` function in the **emmeans** package, introduced in [Section 4.5.1](#), can be used to test differences between the two levels of closeness for each level of preference:

```

library("emmeans")
emmeans(campbell.mod,
        pairwise ~ closeness | preference)$contrasts

preference = weak:
contrast           estimate      SE  df z.ratio p.value
one.sided - close -0.05277 0.22978 Inf -0.230  0.8184

preference = medium:
contrast           estimate      SE  df z.ratio p.value
one.sided - close -0.41427 0.21296 Inf -1.945  0.0517

preference = strong:
contrast           estimate      SE  df z.ratio p.value
one.sided - close -1.10360 0.31979 Inf -3.451  0.0006

Results are given on the log odds ratio (not the response) scale.

```

The tests provided by emmeans are based on the asymptotic normality of the fitted logits and hence use the normal distribution rather than a *t*-distribution. Consequently, the number of degrees of freedom is listed as Inf (infinite). As we can see in the effect plot for the saturated model, the difference in turnout between the two levels of closeness grows with preference, and the *p*-value for this difference is very small at the highest level of preference, very large at the lowest level of preference, and intermediate at the middle level of preference.

The no-interaction model corresponds to parallel profiles of logits in the population analog of [Figure 6.4](#). Rather than using emmeans (), we could test for interaction with either the anova () function, comparing the saturated and no-interaction models, or the Anova () function:

```
campbell.mod.2 <- update(campbell.mod,
  . ~ . - closeness:preference) # no interactions
anova(campbell.mod.2, campbell.mod, test="Chisq")
```

Analysis of Deviance Table

Model 1: cbind(voted, did.not.vote) ~ closeness + preference  
Model 2: cbind(voted, did.not.vote) ~ closeness \* preference

| Resid. | Df | Resid. | Dev  | Df   | Deviance | Pr(>Chi) |
|--------|----|--------|------|------|----------|----------|
| 1      | 2  |        | 7.12 |      |          |          |
| 2      | 0  | 0.00   | 2    | 7.12 | 0.028    |          |

### **Anova (campbell.mod)**

Analysis of Deviance Table (Type II tests)

| Response: cbind(voted, did.not.vote) | LR | Chisq | Df | Pr(>Chisq) |
|--------------------------------------|----|-------|----|------------|
| closeness                            |    | 8.29  | 1  | 0.004      |
| preference                           |    | 19.11 | 2  | 7.1e-05    |
| closeness:preference                 |    | 7.12  | 2  | 0.028      |

The test for the interaction is the same in both cases. Moreover, the alternative hypothesis for this test is the saturated model with zero residual deviance, and so the likelihood-ratio test statistic can also be computed as the residual deviance for the fit of the no-interaction model:

### **deviance (campbell.mod.2)**

```
[1] 7.1186
```

### **df.residual (campbell.mod.2)**

```
[1] 2
```

```
pchisq(deviance(campbell.mod.2), 2, lower.tail=FALSE)
```

```
[1] 0.028458
```

The Anova command also provides Type II tests for the main effects, but if we judge the interaction to be different from zero, then these tests, which ignore the

interaction, should not be interpreted.<sup>9</sup>

<sup>9</sup> Had we specified Type III tests in the Anova () command and used contr.sum () to generate contrasts for the factors (see [Section 5.3.4](#)), we could have interpreted each of the main-effects tests as an average over the levels of the other factor. These tests would be of dubious interest in light of the interaction.

Rather than fitting a *binomial* logit model to the contingency table, we can alternatively fit a *binary* logit model to the 1,275 individual observations comprising Campbell et al.’s data. Let’s regenerate the 1,275 cases from the contingency table. Manipulating data frames in this manner is often complicated, even in relatively small examples.<sup>10</sup> Here’s one way to do it. We first use the reshape () function, introduced in [Section 2.3.7](#), to stack the voted and did.not.vote counts into one column in the new data frame Campbell.temp:

```
(Campbell.temp <- reshape(Campbell[, 1:4],  
    varying=list(response=3:4), direction="long",  
    timevar="turnout", v.names="count"))
```

|     | closeness | preference | turnout | count | id |
|-----|-----------|------------|---------|-------|----|
| 1.1 | one.sided | weak       | 1       | 91    | 1  |
| 2.1 | one.sided | medium     | 1       | 121   | 2  |
| 3.1 | one.sided | strong     | 1       | 64    | 3  |
| 4.1 | close     | weak       | 1       | 214   | 4  |
| 5.1 | close     | medium     | 1       | 284   | 5  |
| 6.1 | close     | strong     | 1       | 201   | 6  |
| 1.2 | one.sided | weak       | 2       | 39    | 1  |
| 2.2 | one.sided | medium     | 2       | 49    | 2  |
| 3.2 | one.sided | strong     | 2       | 24    | 3  |
| 4.2 | close     | weak       | 2       | 87    | 4  |
| 5.2 | close     | medium     | 2       | 76    | 5  |
| 6.2 | close     | strong     | 2       | 25    | 6  |

[10](#) We discuss data management in a general way in [Chapter 2](#), and the programming constructs described in [Chapter 10](#) are often useful for manipulating data.

The id column refers to rows of the original Campbell data frame and is not of interest.

We need to repeat each row of Campbell.temp count times to get one row for each of the observations. For this we use a for () loop (see [Section 10.4.2](#)):

```
Campbell.long <- NULL
for (i in 1:nrow(Campbell.temp)) {
  rows <- Campbell.temp[rep(i, Campbell.temp$count[i]), 1:3]
  Campbell.long <- rbind(Campbell.long, rows)
}
```

The variable turnout is coded 1 for those who voted and 2 for those who did not vote. We'd like these codes to be 0 and 1, respectively, as is appropriate for a binary response. That's easily achieved with the ifelse () function (see [Section 10.4.1](#)):

```
Campbell.long$turnout <- ifelse (Campbell.long$turnout == 2, 0, 1)
```

We check our work by rebuilding the contingency table for the three variables:

```
ftable (xtabs (~ closeness + preference + turnout, data=Campbell.long))
```

|           |            | turnout | 0  | 1   |
|-----------|------------|---------|----|-----|
| closeness | preference |         |    |     |
| one.sided | weak       |         | 39 | 91  |
|           | medium     |         | 49 | 121 |
|           | strong     |         | 24 | 64  |
| close     | weak       |         | 87 | 214 |
|           | medium     |         | 76 | 284 |
|           | strong     |         | 25 | 201 |

The `xtabs()` function creates the three-way contingency table, and then `ftable()` “flattens” the table for printing.

We proceed to fit a binary logistic-regression model to the newly generated `Campbell.long` data, comparing the results to the binomial logit model that we fit previously:

```

campbell.mod.long <- glm(turnout ~ closeness*preference,
  family=binomial, data=Campbell.long)
compareCoefs(campbell.mod, campbell.mod.long)

Calls:
1: glm(formula = cbind(voted, did.not.vote) ~ closeness *
  preference, family = binomial, data = Campbell)
2: glm(formula = turnout ~ closeness * preference, family =
  binomial, data = Campbell.long)

```

|                                 | Model 1 | Model 2 |
|---------------------------------|---------|---------|
| (Intercept)                     | 0.847   | 0.847   |
| SE                              | 0.191   | 0.191   |
| <br>                            |         |         |
| closenessclose                  | 0.0528  | 0.0528  |
| SE                              | 0.2298  | 0.2298  |
| <br>                            |         |         |
| preferencemedium                | 0.0567  | 0.0567  |
| SE                              | 0.2555  | 0.2555  |
| <br>                            |         |         |
| preferencestrong                | 0.134   | 0.134   |
| SE                              | 0.306   | 0.306   |
| <br>                            |         |         |
| closenessclose:preferencemedium | 0.362   | 0.362   |
| SE                              | 0.313   | 0.313   |
| <br>                            |         |         |
| closenessclose:preferencestrong | 1.051   | 1.051   |
| SE                              | 0.394   | 0.394   |

### Anova (campbell.mod.long)

Analysis of Deviance Table (Type II tests)

Response: turnout

|                      | LR | Chisq | Df | Pr(>Chisq) |
|----------------------|----|-------|----|------------|
| closeness            |    | 8.29  | 1  | 0.004      |
| preference           |    | 19.11 | 2  | 7.1e-05    |
| closeness:preference |    | 7.12  | 2  | 0.028      |

The two approaches produce identical coefficient estimates and standard errors and, as the Anova () output demonstrates, identical likelihood-ratio tests based on *differences* in residual deviance for alternative models. The residual deviance is different for the two models, however, because the binomial logit model campbell.mod is fit to six binomial observations, while the binary logit model campbell.mod.long is fit to 1,275 individual binary observations:

```
round(deviance(campbell.mod), 10) # binomial logit model  
[1] 0  
  
df.residual(campbell.mod)  
[1] 0  
  
deviance(campbell.mod.long)      # binary logit model  
[1] 1356.4  
  
df.residual(campbell.mod.long)  
[1] 1269
```

The saturated model with six parameters fit to the six binomial observations is not a saturated model for the 1,275 binary obervations.

## 6.5 Poisson GLMs for Count Data

Poisson generalized linear models arise in two distinct contexts. The first, covered in this section, is more straightforward, where the conditional distribution of the response variable given the predictors follows a Poisson distribution. The second, presented in the next section, is the use of loglinear models for analyzing associations in contingency tables. In most instances, cell counts in contingency tables have multinomial, not Poisson, conditional distributions, but it turns out that with appropriate specification of the model and interpretation of parameters, the multinomial maximum-likelihood estimators can be obtained *as if* the counts were Poisson random variables. Thus, the same Poisson-GLM approach can be used for fitting Poisson regression models and loglinear models for contingency tables.

The default link for the `poisson()` family generator is the log link, and all the models discussed in this section and the next use the log link. The Poisson GLM assumes that the conditional variance of the response is equal to its conditional mean  $\mu$ . This is often a problematic assumption for count data, and we explore some alternatives to the Poisson model in [Section 6.10.4](#).

By way of example, we return to Ornstein's data (from Ornstein, 1976), introduced in [Chapter 3](#), on interlocking director and top-executive positions among 248 major Canadian firms:

### ***brief(Ornstein)***

| 248 x 4 data.frame (243 rows omitted) |               |               |               |                   |
|---------------------------------------|---------------|---------------|---------------|-------------------|
|                                       | assets<br>[i] | sector<br>[f] | nation<br>[f] | interlocks<br>[i] |
| 1                                     | 147670        | BNK           | CAN           | 87                |
| 2                                     | 133000        | BNK           | CAN           | 107               |
| 3                                     | 113230        | BNK           | CAN           | 94                |
| ...                                   |               |               |               |                   |
| 247                                   | 119           | AGR           | CAN           | 6                 |
| 248                                   | 62            | MIN           | US            | 0                 |

Ornstein performed a least-squares regression of the response `interlocks`, the number of interlocks maintained by each firm, on the firm's assets, in millions of dollars; sector of operation, a factor with 10 levels; and nation of control, a factor with four levels. Because the response variable `interlocks` is a count, a Poisson GLM might be preferable.<sup>11</sup> Indeed, the marginal distribution of number of interlocks, in [Figure 6.5](#), shows many zero counts and a substantial positive skew. To construct this graph, we first use the `xtabs()` function to find the frequency distribution of interlocks:

```
(tab <- xtabs(~ interlocks, data=Ornstein))

  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
28  19  14  11   8  14  11   6  12   7   4  12  12   9   8   4   3
. . .
94 107
1   1
```

[11](#) Poisson regression and related GLMs for count data were essentially unknown to sociologists at the time, and so we don't mean to imply criticism of Ornstein's work.

The numbers on top of the frequencies are the different values of interlocks: Thus, there are 28 firms with zero interlocks, 19 with one interlock, 14 with two, and so on. The graph is produced by plotting the counts in tab against the category labels converted into numbers:

```
x <- as.numeric(names(tab)) # distinct values of interlocks
plot(x, tab,
      type="h",
      xlab="Number of Interlocks", ylab="Frequency")
points(x, tab, pch=16)
```

Specifying type="h" in the call to plot () produces the "histogram-like" vertical lines on the graph, while the points () function adds the filled circles (pch=16) at the tops of the lines.

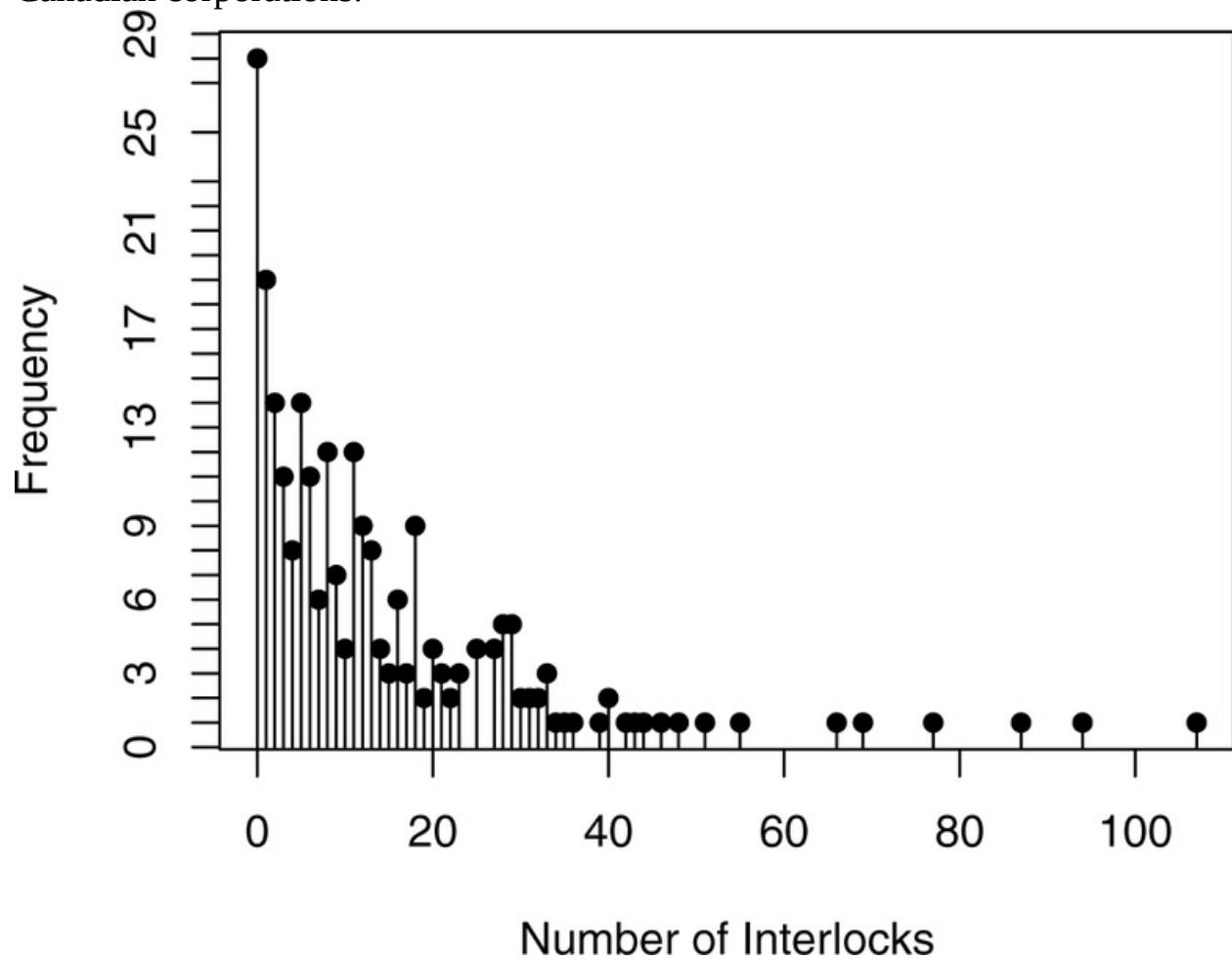
Preliminary examination of the data suggests the use of  $\log_2(\text{assets})$  in place of assets in the regression.[12](#)

```
mod.ornstein <- glm (interlocks ~ log2(assets) + nation + sector,
family=poisson, data=Ornstein)
```

**S (mod.ornstein)**

[12](#) Ornstein used both assets and the log of assets in his least-squares regression.

**Figure 6.5** Distribution of number of interlocks maintained by 248 large Canadian corporations.



```
Call: glm(formula = interlocks ~ log2(assets) + nation +  
        sector, family = poisson, data = Ornstein)
```

Coefficients:

|              | Estimate | Std. Error | z value | Pr(> z ) |
|--------------|----------|------------|---------|----------|
| (Intercept)  | -0.8394  | 0.1366     | -6.14   | 8.1e-10  |
| log2(assets) | 0.3129   | 0.0118     | 26.58   | < 2e-16  |
| nationOTH    | -0.1070  | 0.0744     | -1.44   | 0.15030  |
| nationUK     | -0.3872  | 0.0895     | -4.33   | 1.5e-05  |
| nationUS     | -0.7724  | 0.0496     | -15.56  | < 2e-16  |
| sectorBNK    | -0.1665  | 0.0958     | -1.74   | 0.08204  |
| sectorCON    | -0.4893  | 0.2132     | -2.29   | 0.02174  |
| sectorFIN    | -0.1116  | 0.0757     | -1.47   | 0.14046  |
| sectorHLD    | -0.0149  | 0.1192     | -0.13   | 0.90051  |
| sectorMAN    | 0.1219   | 0.0761     | 1.60    | 0.10949  |
| sectorMER    | 0.0616   | 0.0867     | 0.71    | 0.47760  |
| sectorMIN    | 0.2498   | 0.0689     | 3.63    | 0.00029  |
| sectorTRN    | 0.1518   | 0.0789     | 1.92    | 0.05445  |
| sectorWOD    | 0.4983   | 0.0756     | 6.59    | 4.4e-11  |

(Dispersion parameter for poisson family taken to be 1)

```
Null deviance: 3737.0 on 247 degrees of freedom  
Residual deviance: 1547.1 on 234 degrees of freedom
```

| logLik  | df | AIC    | BIC    |
|---------|----|--------|--------|
| -1222.5 | 14 | 2473.1 | 2522.3 |

Number of Fisher Scoring iterations: 5

## Exponentiated Coefficients and Confidence Bounds

|              | Estimate | 2.5 %   | 97.5 %  |
|--------------|----------|---------|---------|
| (Intercept)  | 0.43198  | 0.33005 | 0.56390 |
| log2(assets) | 1.36741  | 1.33628 | 1.39938 |
| nationOTH    | 0.89853  | 0.77526 | 1.03781 |
| nationUK     | 0.67894  | 0.56758 | 0.80634 |
| nationUS     | 0.46191  | 0.41888 | 0.50885 |
| sectorBNK    | 0.84661  | 0.70187 | 1.02160 |
| sectorCON    | 0.61306  | 0.39318 | 0.91022 |
| sectorFIN    | 0.89439  | 0.77136 | 1.03795 |
| sectorHLD    | 0.98520  | 0.77524 | 1.23792 |
| sectorMAN    | 1.12960  | 0.97266 | 1.31111 |
| sectorMER    | 1.06351  | 0.89587 | 1.25875 |
| sectorMIN    | 1.28383  | 1.12233 | 1.47031 |
| sectorTRN    | 1.16394  | 0.99684 | 1.35847 |
| sectorWOD    | 1.64585  | 1.41857 | 1.90817 |

## Anova (mod. ornstein)

Analysis of Deviance Table (Type II tests)

Response: interlocks

|              | LR  | Chisq | Df | Pr (>Chisq) |
|--------------|-----|-------|----|-------------|
| log2(assets) | 731 | 1     |    | <2e-16      |
| nation       | 276 | 3     |    | <2e-16      |
| sector       | 103 | 9     |    | <2e-16      |

The Type II analysis of deviance, produced by the Anova () function in the **car** package, shows that all three predictors have very small *p*-values.

The coefficients of the model are effects on the log-count scale of the linear predictor, and consequently exponentiating the coefficients produces multiplicative effects on the count scale. The exponentiated coefficients are

displayed by default, along with confidence limits, in the `S()` output for a Poisson GLM fit with the log link. We thus estimate, for example, that doubling assets (i.e., increasing the  $\log_2$  of assets by 1), holding the other predictors constant, multiplies the expected number of interlocks by 1.37 (i.e., increases expected interlocks by 37%). Likewise, compared to a similar Canadian firm, which is the baseline level for the factor nation, a U.S. firm on average maintains only 46% as many interlocks.

We can use the **effects** package to visualize a Poisson GLM. For the model fit to Ornstein's data, the following command produces the predictor effect plots shown in [Figure 6.6](#):

```
plot (predictorEffects (mod.ornstein, xlevels=list (assets=100)), axes=list
(x=list (rotate=40)))
```

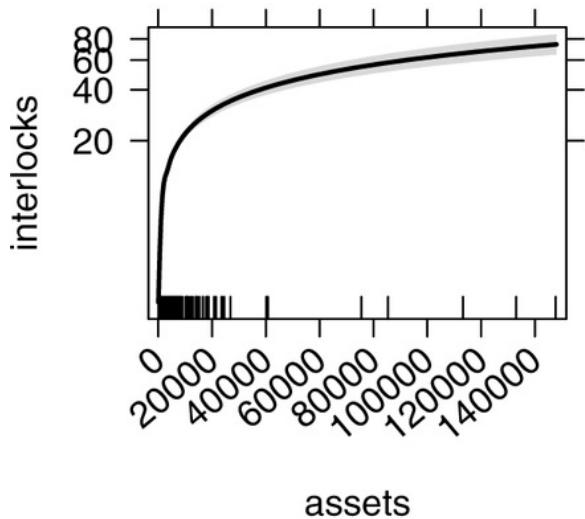
- By default, the vertical axis in these graphs is on the scale of the linear predictor, which is the log-count scale for Ornstein's Poisson regression. The axis tick marks are labeled on the scale of the response, number of interlocks in our example. This is similar to a logistic regression, where we plotted on the logit scale but labeled the vertical axis on the probability scale.
- Also by default, the range of the vertical axis is different in the several graphs, a feature of the graphs to which we should attend in assessing the relative impact on interlocks of the three predictors.<sup>13</sup>
- The rug-plot at the bottom of the effect display for assets shows the highly skewed marginal distribution of this predictor: About 95% of the values of assets are less than \$20,000, so the right portion of this plot may be suspect.
- The levels of the factor sector are ordered alphabetically; the effect plot for this factor would be easier to decode if we rearranged the levels so that they were in the same order as the effects, as happened by happy accident for the factor nation.
- The plot for assets is curved because the model uses  $\log_2(\text{assets})$  as the regressor. We specify the argument `xlevels=list(assets=100)` to `predictorEffects()` so that the curve is evaluated at 100 equally spaced values of assets rather than the default 50 equally spaced *quantiles*, producing a more accurate representation of the fit at the right of the graph for this very highly skewed predictor.

- Rotating the tick labels on the horizontal axis, via `axes=list (x=list (rotate=40))`, prevents the labels from overwriting each other.

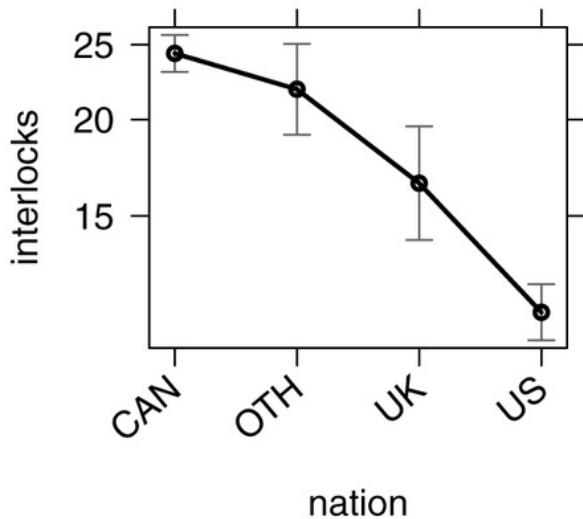
[13](#) We could set a *common* scale for the vertical axes by, for example, specifying the argument `axes=list (y=list (lim=log (c (5, 80))))` to plot (), but then the graphs for all three predictors would be considerably compressed—try it and see!

**Figure 6.6** Predictor effect plots for the Poisson regression model fit to Ornstein’s interlocking-directorate data. The band and bars show pointwise 95% confidence limits around the fitted effects.

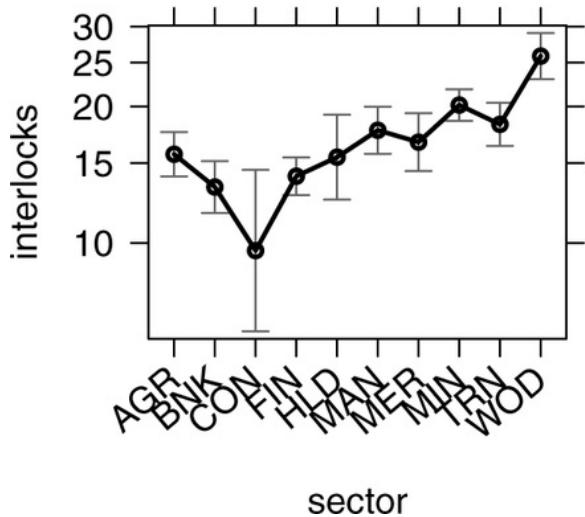
**assets predictor effect plot**



**nation predictor effect plot**



**sector predictor effect plot**



## 6.6 Loglinear Models for Contingency Tables

A path-breaking paper by Birch (1963) defined the class of *loglinear models* for contingency tables, and in the early years following the publication of Birch's paper, specialized software was developed for fitting these models. Nelder and Wedderburn (1972) made it clear that Poisson regression could be used to fit loglinear models. References to comprehensive treatments of the subject are given in the complementary readings at the end of the chapter.

### 6.6.1 Two-Dimensional Tables

A *contingency table* is an array of counts in two or more dimensions. Most familiar is the two-dimensional or two-way contingency table. As an example, we will construct a two-way table from a higher-dimensional cross-classification of all the PhDs awarded in the mathematical sciences in the United States in 2008–2009 (from Phipps, Maxwell, & Rose, 2009, Supp. Table IV), which is in the data frame `AMSSurvey` in the `car` package. There are several variables in `AMSSurvey`, including<sup>14</sup>

- type of institution, a factor in which levels "I (Pu)" and "I (Pr)" are math departments in high-quality public and private universities; levels "II" and "III" are math departments in progressively lower-quality universities; level "IV" represents statistics and biostatistics departments; and level "Va" is applied mathematics
- sex of the degree recipient, with levels "Female" and "Male"
- citizen, the citizenship of the degree recipient, a factor with levels "US" and "Non-US"
- count, the number of individuals for each combination of type, sex, and citizen in 2008–2009. The `AMSSurvey` data frame has one row for each of the cells in the table, thus  $6 \times 2 \times 2 = 24$  rows in all.

<sup>14</sup> The `AMSSurvey` data set includes counts for additional years after 2008–2009. We invite the reader to redo the analysis reported here for these other years. It is also possible to reorganize the data set to treat year as an additional variable, a possibility that we won't pursue.

```

brief(AMSSurvey[, 1:4]) # a few rows, first 4 columns

24 x 4 data.frame (19 rows omitted)
  type      sex citizen count
  [f]      [f]     [f]    [i]
1 I(Pu)   Male    US     132
2 I(Pu) Female   US      35
3 I(Pr)   Male    US      87
...
23 Va     Male Non-US     28
24 Va     Female Non-US    12

```

To examine the association between sex and citizen, for instance, we can create a two-way table by adding over the levels of the third variable, type:

```

(tab.sex.citizen <- xtabs(count ~ sex + citizen,
data=AMSSurvey)

            citizen
sex      Non-US   US
Female    260  202
Male      501  467

```

For example, 260 female PhD recipients were non-U.S. citizens in this period. The usual analysis of a two-dimensional table like this consists of a test of independence of the row and column classifications:

```
chisq.test(tab.sex.citizen,  
           correct=FALSE) # suppress Yates correction
```

Pearson's Chi-squared test

```
data: tab.sex.citizen  
X-squared = 2.5673, df = 1, p-value = 0.1091
```

The test statistic is the uncorrected Pearson's  $X^2$ , defined by the familiar formula

$$X^2 = \sum \frac{(O - E)^2}{E}$$

where the sum is over all cells of the table,  $O$  are the observed cell counts, and  $E$  are the estimated expected cell counts computed under the assumption that the null hypothesis of independence is true. The approximate  $p$ -value for the test is determined by comparing the obtained  $X^2$  to a  $\chi^2$  distribution with degrees of freedom depending on the dimensions of the table and on the number of parameters estimated under the null hypothesis. For an  $r \times c$  table, the  $df$  for the test of independence are  $(r - 1)(c - 1)$ ; here,  $(2 - 1)(2 - 1) = 1$   $df$ . The chisq.test() function can also approximate the  $p$ -value using a simulation, an approach that is preferred when cell counts are small. The default argument correct=TRUE produces a corrected version of  $X^2$  that is also more accurate in small samples. The  $p$ -value close to .1 suggests at best weak evidence that the proportion of women is different for citizens and noncitizens, or, equivalently, that the proportion of noncitizens is different for males and females.

If there were clear predictor and response variables in the table, it would also be standard practice to convert the frequency counts into percentages within each category of the predictor. For example, were we to regard citizen as the predictor and sex as the response, which isn't entirely sensible, we could calculate the column percentage table as

```
100*prop.table(tab.sex.citizen, margin=2)
```

|        |  | citizen |        |
|--------|--|---------|--------|
| sex    |  | Non-US  | US     |
| Female |  | 34.166  | 30.194 |
| Male   |  | 65.834  | 69.806 |

The sex distribution is similar for U.S. citizens in the second column of the percentage table and noncitizens in the first column.

A loglinear model can also be fit to a two-way contingency table by treating the cell counts as independent Poisson random variables. First, we need to change the two-way table into a data frame with rows corresponding to the cells of the table, where the new variable Freq is the cell frequency count:

```
(AMS2 <- as.data.frame(tab.sex.citizen))
```

|   | sex    | citizen | Freq |
|---|--------|---------|------|
| 1 | Female | Non-US  | 260  |
| 2 | Male   | Non-US  | 501  |
| 3 | Female | US      | 202  |
| 4 | Male   | US      | 467  |

Expected cell counts in the model of independence depend on each of the factors separately but not jointly:

```

phd.mod.indep <- glm(Freq ~ sex + citizen, family=poisson,
                      data=AMS2)
# estimated expected cell counts assuming independence:
predict(phd.mod.indep, type="response")

      1       2       3       4
245.86 515.14 216.14 452.86

deviance(phd.mod.indep) # L.R. test statistic for independence
[1] 2.5721

df.residual(phd.mod.indep)

[1] 1

```

Modeling dependence—that is, an association between sex and citizen—requires adding the sex:citizen interaction to the model, producing the saturated model. The residual deviance is the difference in deviance between the model we fit and the saturated model.<sup>15</sup>

[15](#) It is traditional in loglinear models to call terms such as sex:citizen “interactions,” even though they represent not interaction in the usual sense of the word—that is, a change in the partial relationship between two variables across different levels of a third variable—but rather the *association* between a pair of variables.

The residual deviance of 2.5721 for the independence model is thus a lack-of-fit test for this model or, equivalently, a test that the association between sex and citizen is zero. Like Pearson’s  $X^2$ , the residual deviance is compared to the  $\chi^2$  distribution with  $df$  equal to the residual  $df$  for the model. A  $p$ -value for this test is not automatically reported because the residual deviance is a lack-of-fit test only for cross-classified data with one row in the data set for each cell. We can compute the  $p$ -value as

```

pchisq (deviance (phd.mod.indep), df=df.residual (phd.mod.indep),
lower.tail=FALSE)

```

```
[1] 0.10876
```

The test based on the deviance is a *likelihood-ratio test* of the hypothesis of independence, while Pearson's  $X^2$  is a *score test* of the same hypothesis. The two tests are asymptotically equivalent, but in small samples, Pearson's  $X^2$  can give more accurate inferences. The change in deviance is generally preferred because it is more useful for comparing models other than the independence and saturated models. To compute Pearson's  $X^2$  for any GLM fit, simply sum the squared Pearson residuals; here:

```
sum (residuals (phd.mod.indep, type = "pearson") ^ 2)  
[1] 2.5673
```

## 6.6.2 Three-Dimensional Tables

The AMSSurvey data comprise a three-dimensional table, the first dimension representing the type of institution; the second, the sex of the degree recipient; and the third, citizenship status (citizen). In the preceding section, we collapsed the data over type to analyze a two-way table for sex and citizen. With three dimensions, there is a much richer set of models that can be fit to the data. The saturated model consists of an intercept, all possible main effects, all two-factor interactions or associations, and the three-factor interaction.<sup>16</sup> The formula for the saturated model is or, equivalently and more compactly,

```
count ~ type * sex * citizen
```

<sup>16</sup> As in the case of two-way tables, the terms *main effects* and *interactions* can be misleading. The “main effects” pertain to the marginal distributions of the three variables, and the “two-way interactions” to the partial associations between pairs of variables. The “three-way interaction” represents interaction in the more usual sense, in that the presence of this term in the model implies that the partial association between each pair of variables varies over the levels of the third variable.

```

count ~ type + sex + citizen
+ type:sex + type:citizen + sex:citizen
+ type:sex:citizen

```

Additional models are obtained from the saturated model by deleting terms, subject to the constraints of the marginality principle. The Anova () function can be used to test all Type II hypotheses conforming to the principle of marginality:

```

phd.mod.all <- glm(count ~ type*sex*citizen, # saturated model
                     family=poisson, data=AMSSurvey)
Anova(phd.mod.all)

```

Analysis of Deviance Table (Type II tests)

Response: count

|                  | LR    | Chisq | Df      | Pr(>Chisq) |
|------------------|-------|-------|---------|------------|
| type             | 233.3 | 5     | < 2e-16 |            |
| sex              | 183.0 | 1     | < 2e-16 |            |
| citizen          | 5.9   | 1     | 0.01494 |            |
| type:sex         | 69.1  | 5     | 1.6e-13 |            |
| type:citizen     | 24.0  | 5     | 0.00021 |            |
| sex:citizen      | 0.5   | 1     | 0.46346 |            |
| type:sex:citizen | 1.4   | 5     | 0.92220 |            |

As is usual when there are higher-order terms such as interactions in a model, we read the analysis-of-deviance table from bottom to top. The  $p$ -values for the three-factor interaction and for the sex:citizen interaction are very large, and so these terms can probably be ignored. The  $p$ -values for the remaining two-factor interactions are small, suggesting that these interactions are nonzero. The tests for the main effects are generally irrelevant in loglinear models and are usually ignored because they pertain to the marginal distributions of the factors. The model with two interactions is a model of *conditional independence*: Within levels of institution type, which is the variable common to the two interactions, sex and citizen are independent (see, e.g., Agresti, 2007, [Section 3.1.2](#)).

We update the saturated model, removing the small interactions:

```
phd.mod.1 <- update(phd.mod.all,
  . ~ . - sex:citizen - type:sex:citizen)
deviance(phd.mod.1)
```

```
[1] 1.9568
```

```
df.residual(phd.mod.1)
```

```
[1] 6
```

This model reproduces the data quite well, as reflected in a lack-of-fit test obtained by comparing the residual deviance to the  $\chi^2$  distribution with six *df*:

```
pchisq(deviance(phd.mod.1), df.residual(phd.mod.1),
lower.tail=FALSE)
```

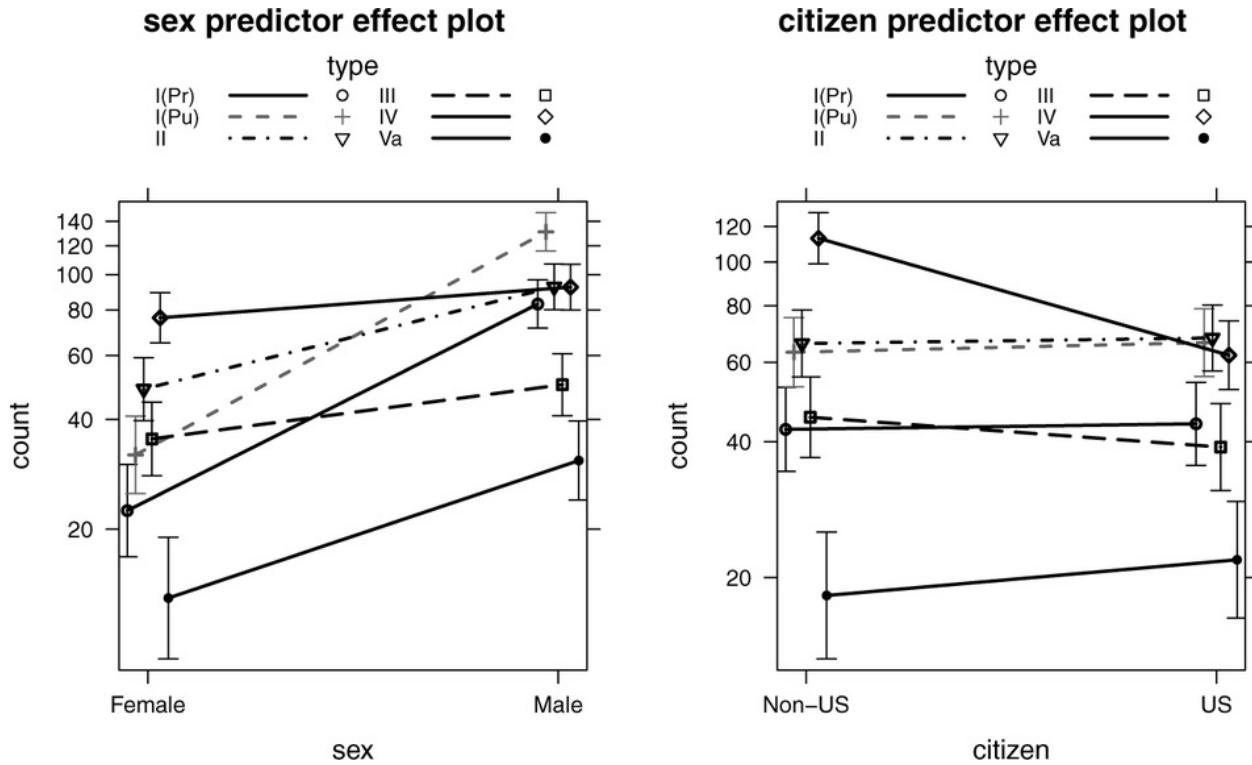
```
[1] 0.92363
```

The very large *p*-value suggests no evidence of lack of fit.

Although it is possible to interpret this model using the coefficients (which we don't show), predictor effect plots, in [Figure 6.7](#), provide a more effective summary:

```
plot(predictorEffects(phd.mod.1, ~ sex + citizen), lines=list
  (multiline=TRUE), lattice=list(key.args=list(columns=2)), confint=list
  (style="bars"))
```

**Figure 6.7** Predictor effect plots for the Mathematical Sciences PhD model.



From the plot for sex, we see that in all types of institutions, males are modeled to outnumber females, particularly in the public and private Type I institutions. At the other extreme, Type IV departments (statistics and biostatistics) are nearly gender balanced. With respect to citizen, we see that most types of institutions have nearly equal numbers of U.S. and non-U.S. degrees awarded, because the lines shown are nearly horizontal. The exception is statistics and biostatistics (Type IV) departments, where non-U.S. degrees dominate.

### 6.6.3 Sampling Plans for Loglinear Models

The AMSsurvey data simply classify each 2008–2009 PhD degree awarded according to type of institution, sex, and citizenship, leading to a plausible assumption that the count in each cell of the table has a Poisson distribution and therefore to the Poisson GLM that we used. Not all contingency tables are constructed in this way. Consider this imaginary study: We will collect data to learn about the characteristics of customers who use human tellers in banks to make transactions. We will study transactions between tellers and customers, classifying each transaction by the factor age of the customer, A, with, say, three age groups; the factor gender, B, male or female; and the factor C, whether or not the transaction was simple enough to be done at an ATM rather than with a

human teller. Each of the following sampling plans leads to a different distribution for the counts, but the Poisson GLM can legitimately be used to estimate parameters and perform tests in all of these cases:

**Poisson sampling** Go to the bank for a fixed period of time, observe transactions, and classify them according to the three variables. In this case, even the sample size  $n$  is random. Poisson regression models are appropriate here, and all parameters are potentially of interest.

**Multinomial sampling** Fix the sample size  $n$  in advance, and sample as long as necessary to get  $n$  transactions. This scheme differs from the first sampling plan only slightly: The counts are no longer independent Poisson random variables, because their sum is constrained to equal the fixed sample size  $n$ , and so the counts instead follow a multinomial distribution. A Poisson GLM can be used but the overall mean parameter, the intercept of the Poisson model, is determined by the sampling plan and consequently must be included in the model.

**Product-multinomial sampling** We choose to sample equal numbers of men and women. In this case, the intercept and the B main effect, which reflects the marginal distribution of B, are fixed by the design and must be in the model.

**Fix two predictors** We could sample a fixed number in each age  $\times$  sex combination to get a different multinomial sampling scheme. All models fit to the data must include the terms  $1 + A + B + A:B = A^*B$ , because these are all fixed by the sampling design.

**Retrospective sampling** Suppose that complex transactions are rare, and so we decide to sample  $n/2$  complex transactions and  $n/2$  simple ones. This scheme differs from the earlier sampling plans because factor C is effectively a response variable, and we have guaranteed that we will have an equal number of observations in each of the response categories. All models must include  $1 + C$ . The terms  $A:C$ ,  $B:C$ , and  $A:B:C$  tell us whether and how the response is related to the predictors, and all of these terms are estimable. This type of sampling is usually called *case-control sampling* or *retrospective sampling* and is often used in medical studies of rare diseases.

## 6.6.4 Response Variables

In the AMSsurvey data, none of the three classification variables can reasonably be considered response variables. In other loglinear models, any, or all, the

classification variables could be considered responses. When there are response variables, the interesting terms in a loglinear model are typically the interactions or associations between the predictors and the responses. All models will generally include the highest-order interaction among the predictors. As usual, the models to be considered should obey the marginality principle.

An example is provided by the *American Voter* data in [Table 6.5](#) (page 290), to which we previously fit a binomial logistic-regression model in which turnout was the response. We can also view these data as forming a three-dimensional contingency table and fit loglinear models to the table. The three dimensions of the table are closeness, with two levels; preference, with three levels; and the response turnout, with two levels. In this case, then, there are two predictors and a single response. The data here were drawn from a national survey; ignoring the complexity of the sampling design of the survey, we will treat the data as if they arose from a Poisson or simple multinomial sampling plan.<sup>17</sup> In [Section 6.4](#), we reformatted the Campbell data frame into Campbell.long with one row for each of the observations. Along the way, we created a data frame called Campbell.temp that is almost the form of the data that we now need:

```

Campbell.temp$turnout <- factor(Campbell.temp$turnout,
  labels=c("voted", "did.not.vote"))
Campbell.temp

```

|     | closeness | preference | turnout      | count | id |
|-----|-----------|------------|--------------|-------|----|
| 1.1 | one.sided | weak       | voted        | 91    | 1  |
| 2.1 | one.sided | medium     | voted        | 121   | 2  |
| 3.1 | one.sided | strong     | voted        | 64    | 3  |
| 4.1 | close     | weak       | voted        | 214   | 4  |
| 5.1 | close     | medium     | voted        | 284   | 5  |
| 6.1 | close     | strong     | voted        | 201   | 6  |
| 1.2 | one.sided | weak       | did.not.vote | 39    | 1  |
| 2.2 | one.sided | medium     | did.not.vote | 49    | 2  |
| 3.2 | one.sided | strong     | did.not.vote | 24    | 3  |
| 4.2 | close     | weak       | did.not.vote | 87    | 4  |
| 5.2 | close     | medium     | did.not.vote | 76    | 5  |
| 6.2 | close     | strong     | did.not.vote | 25    | 6  |

[17](#) Complex survey data can be properly analyzed in R using the **survey** package, which, among its many facilities, has a function for fitting generalized linear models (see the online appendix to the *R Companion* on survey data). We do not have the original survey data set from which our contingency table was constructed.

We redefine turnout, previously a numeric variable coded 1 and 2, as a factor, although that is not strictly necessary because it only has two levels. We simply ignore the id variable in the data set, which is irrelevant to our current purpose.

Because we are interested in finding the important associations with the response, we start by fitting the saturated model and examining tests for various terms:

```

mod.loglin <- glm(count ~ closeness*preference*turnout,
  family=poisson, data=Campbell.temp)
Anova(mod.loglin)

```

Analysis of Deviance Table (Type II tests)

Response: count

|                              | LR | Chisq | Df | Pr(>Chisq) |
|------------------------------|----|-------|----|------------|
| closeness                    |    | 201   | 1  | < 2e-16    |
| preference                   |    | 56    | 2  | 6.8e-13    |
| turnout                      |    | 376   | 1  | < 2e-16    |
| closeness:preference         |    |       | 2  | 0.540      |
| closeness:turnout            |    | 8     | 1  | 0.004      |
| preference:turnout           |    | 19    | 2  | 7.1e-05    |
| closeness:preference:turnout |    | 7     | 2  | 0.028      |

The only terms of interest are those that involve the response variable turnout, and all three of these terms, including the highest-order term closeness:preference:turnout, have small *p*-values.

As long as a loglinear model with a dichotomous response variable includes the highest-order interaction among the predictors, here closeness:preference, and obeys the principle of marginality, the loglinear model is equivalent to a logistic-regression model. All of the parameter estimates in the logistic regression also appear in the loglinear model, but they are labeled differently. For example, the closeness main effect in the logistic regression corresponds to the closeness:turnout term in the loglinear model; similarly, the closeness:preference interaction in the logistic regression corresponds to closeness:preference:turnout in the loglinear model. Likelihood-ratio tests for corresponding terms are identical for the logistic-regression and loglinear models, as the reader can verify for the example (cf. page 293). The only important difference is that the residual deviance for the loglinear model provides a goodness-of-fit test for that model, but it is not a goodness-of-fit test for the logistic regression.<sup>18</sup>

<sup>18</sup> These conclusions extend to polytomous (i.e., multicategory) responses, where loglinear models that fit the highest-order interaction among the predictors are equivalent to multinomial logit models (described in [Section 6.7](#)).

## 6.7 Multinomial Response Data

The data frame `Womenlf` in the **carData** package contains data on  $n = 263$  women between the ages of 21 and 30 drawn from a social survey of the Canadian population in 1977:

**summary(Womenlf)**

|           | partic | hincome      | children    | region       |
|-----------|--------|--------------|-------------|--------------|
| fulltime: | 66     | Min. : 1.0   | absent : 79 | Atlantic: 30 |
| not.work: | 155    | 1st Qu.:10.0 | present:184 | BC : 29      |
| parttime: | 42     | Median :14.0 |             | Ontario :108 |
|           |        | Mean :14.8   |             | Prairie : 31 |
|           |        | 3rd Qu.:19.0 |             | Quebec : 65  |
|           |        | Max. :45.0   |             |              |

**nrow(Womenlf)**

[1] 263

The following variables are included in this data set:

- `partic`: labor force participation, a factor, with levels "not.work", "parttime", and "fulltime"
- `hincome`: husband's income, in thousands of dollars
- `children`: presence of children in the household, a factor with levels "absent" and "present"
- `region`: a factor with levels "Atlantic", "Quebec", "Ontario", "Prairie", and "BC"

We listed the levels of `partic` in their natural order, and of `region` from East to West, but the levels are in alphabetical order in the data frame.

If husband's income were recoded as a factor with a relatively small number of levels, we could view these data as a four-dimensional contingency table. There is one response variable, `partic`, which has three possibly ordered categories. Assuming Poisson sampling or the equivalent, we could then analyze the data using appropriate loglinear models with one response. In particular, we would fit models conforming to the principle of marginality that always include the

highest-order interactions among the three predictors and then examine the interesting interactions (i.e., associations) between the predictors and the response variable.

Just as a contingency table with a single binomial response variable is more conveniently modeled using a binomial-regression model than an equivalent loglinear model, a contingency table with a single multinomial response can be modeled with a *multinomial-regression model* that generalizes the binomial model. Multinomial-regression models can also be fit to data that include continuous numeric predictors, such as hincome in the Womenlf data set.

In the *multinomial logit model*, one of the  $q$  response levels is selected as a baseline, and we effectively fit logistic-regression models comparing each of the remaining levels to that baseline. As usual, R selects the first level to be the baseline. The arbitrary choice of a baseline level for the response is inconsequential in the sense that it doesn't influence the fit of the model to the data.

Suppose that we let  $\eta_j(\mathbf{x})$  be the linear predictor for the comparison of category  $j$  to the baseline,  $j = 2, \dots, q$ ; assuming that an intercept is included in the model,

$$\eta_j(\mathbf{x}) = \beta_{0j} + \beta_{1j}x_1 + \cdots + \beta_{kj}x_k$$

Then, for  $j = 2, \dots, q$ ,

$$\log \left[ \frac{\Pr(\text{response is level } j|\mathbf{x})}{\Pr(\text{response is baseline level 1}|\mathbf{x})} \right] = \eta_j(\mathbf{x})$$

In terms of probabilities, if  $\mu_j(\mathbf{x}) = \Pr(\text{response is level } j|\mathbf{x})$ ,

(6.2)

$$\mu_j(\mathbf{x}) = \frac{\exp [\eta_j(\mathbf{x})]}{1 + \sum_{\ell=2}^q \exp [\eta_\ell(\mathbf{x})]} \quad \text{for } j = 2, \dots, q$$

$$\mu_1(\mathbf{x}) = 1 - \sum_{\ell=2}^q \mu_\ell(\mathbf{x}) \quad \text{for } j = 1$$

The log-odds between *any* pair of response levels  $j$  and  $j' \neq 1$  is  
 The log-odds between *any* pair of response levels  $j$  and  $j' \neq 1$  is

$$\begin{aligned}\log \frac{\mu_j(\mathbf{x})}{\mu_{j'}(\mathbf{x})} &= \log \left[ \frac{\mu_j(\mathbf{x})/\mu_1(\mathbf{x})}{\mu_{j'}(\mathbf{x})/\mu_1(\mathbf{x})} \right] \\ &= \log \frac{\mu_j(\mathbf{x})}{\mu_1(\mathbf{x})} - \log \frac{\mu_{j'}(\mathbf{x})}{\mu_1(\mathbf{x})} \\ &= \eta_j(\mathbf{x}) - \eta_{j'}(\mathbf{x}) \\ &= (\beta_{0j} - \beta_{0j'}) + (\beta_{1j} - \beta_{1j'})x_1 + \cdots + (\beta_{kj} - \beta_{kj'})x_k\end{aligned}$$

The logistic-regression coefficients for the log-odds of membership in level  $j$  versus  $j'$  are given by *differences* in the corresponding parameters of the multinomial logit model.

The multinomial logit model is not a traditional GLM and cannot be fit by the `glm()` function. Instead, we use the `multinom()` function in the **nnet** package (Venables & Ripley, 2002), one of the recommended packages that are part of the standard R distribution.<sup>19</sup> Just as `glm()` can fit regression models to both binomial and binary data, `multinom()` can fit models in which the observations represent counts in the several response categories, as one would have in a contingency table, or individual responses, as in our example.

<sup>19</sup> There are other CRAN packages that also can fit multinomial logit models. Of particular note is the **VGAM** package (Yee, 2015), which fits a wide variety of regression models for categorical and other data.

We begin our analysis of the Canadian women's labor force data by reordering the levels of the response so that the baseline level is `not.work` and then fit a multinomial logit model with the predictors `hincome`, `children`, and `region`, entertaining the possibility that the effect of husband's income depends on presence of children by including the interaction between `hincome` and `children` in the model:

```

library("nnet")
Womenlf$partic <- factor(Womenlf$partic,
  levels=c("not.work", "parttime", "fulltime"))
mod.multinom <- multinom(partic ~ hincome*children + region,
  data=Womenlf)

# weights: 27 (16 variable)
initial value 288.935032
iter 10 value 208.338461
iter 20 value 207.072018
final value 207.071960
converged

```

The call to multinom () prints a summary of the iteration history required to find the estimates unless the argument trace=FALSE is supplied.

A Type II analysis of deviance for this model, computed by the Anova () function in the **car** package, shows that the region term and the hincome×children interaction have large *p*-values, while the main effects of both hincome and children, the tests for which assume that the interaction is absent, have very small *p*-values:

### **Anova (mod.multinom)**

Analysis of Deviance Table (Type II tests)

Response: partic

|                  | LR | Chisq | Df | Pr(>Chisq) |
|------------------|----|-------|----|------------|
| hincome          |    | 14.6  | 2  | 0.00066    |
| children         |    | 65.2  | 2  | 6.9e-15    |
| region           |    | 7.3   | 8  | 0.50614    |
| hincome:children |    | 1.3   | 2  | 0.51642    |

We simplify the model by removing the interaction and region effects, and summarize the resulting model:

```
mod.multinom.1 <- update(mod.multinom,
  . ~ . - region - hincome:children)
```

**s(mod.multinom.1)**

```
Call: multinom(formula = partic ~ hincome + children, data =
  Womenlf)
```

Coefficients:

parttime

|                 | Estimate | Std. Error | z value | Pr(> z ) |
|-----------------|----------|------------|---------|----------|
| (Intercept)     | -1.43232 | 0.59246    | -2.42   | 0.016    |
| hincome         | 0.00689  | 0.02345    | 0.29    | 0.769    |
| childrenpresent | 0.02146  | 0.46904    | 0.05    | 0.964    |

fulltime

|                 | Estimate | Std. Error | z value | Pr(> z ) |
|-----------------|----------|------------|---------|----------|
| (Intercept)     | 1.9828   | 0.4842     | 4.10    | 4.2e-05  |
| hincome         | -0.0972  | 0.0281     | -3.46   | 0.00054  |
| childrenpresent | -2.5586  | 0.3622     | -7.06   | 1.6e-12  |

Residual Deviance: 423

| logLik  | df | AIC    | BIC    |
|---------|----|--------|--------|
| -211.44 | 6  | 434.88 | 456.31 |

The first coefficient table corresponds to the comparison between "parttime" and the baseline level "not.working" and the second table to the comparison between "fulltime" and "not.working". Examining the Wald statistics in the column in the tables labeled "z value" and the associated *p*-values, the coefficients for the logit of "parttime" versus "not.work" are, apart from the regression constant, small with large *p*-values, while the coefficients for the logit of "fulltime" versus "not.work" are much larger and have very small *p*-values. Apparently, husband's income and presence of children primarily affect the decision to work full-time, as opposed to part-time or not to work outside the home.[20](#)

[20](#) Adding the argument `exponentiate=TRUE` to the call to `S()` would print tables of exponentiated coefficients and confidence limits, representing multiplicative effects on the odds scale. We skip this step in favor of the predictor effect plots discussed immediately below.

Interpreting estimated coefficients in a multinomial regression can be challenging, even in the absence of interactions, polynomial regressors, and regression splines, in part because coefficients depend on the choice of the baseline level. A useful way to visualize the fitted model is with predictor effect plots, which, unlike the coefficients, are unaffected by the arbitrary choice of baseline level for the response. Predictor effect plots for husband's income and presence of children in the current example appear in [Figure 6.8](#) and are produced by the following command:

```
plot (predictorEffects (mod.multinom.1))
```

The fitted probabilities  $\mu_j(\mathbf{x})$  in the effect plots are calculated according to Equation 6.2, separately for each level of partic. Pointwise confidence intervals on the probability scale are computed by the delta method (Fox & Andersen, 2006).

Alternatively, we can show how the response depends simultaneously on `hincome` and `children`, even though these two predictors don't interact in model `mod.multinom.1`, generating [Figure 6.9](#):

```
plot (Effect (c ("hincome", "children"), mod.multinom.1))
```

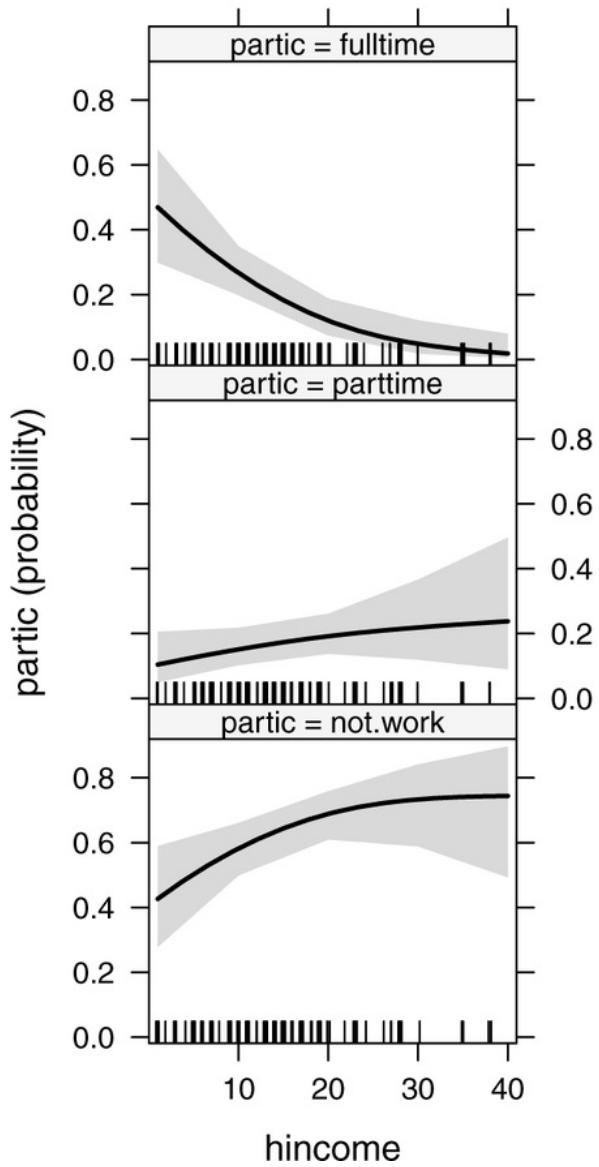
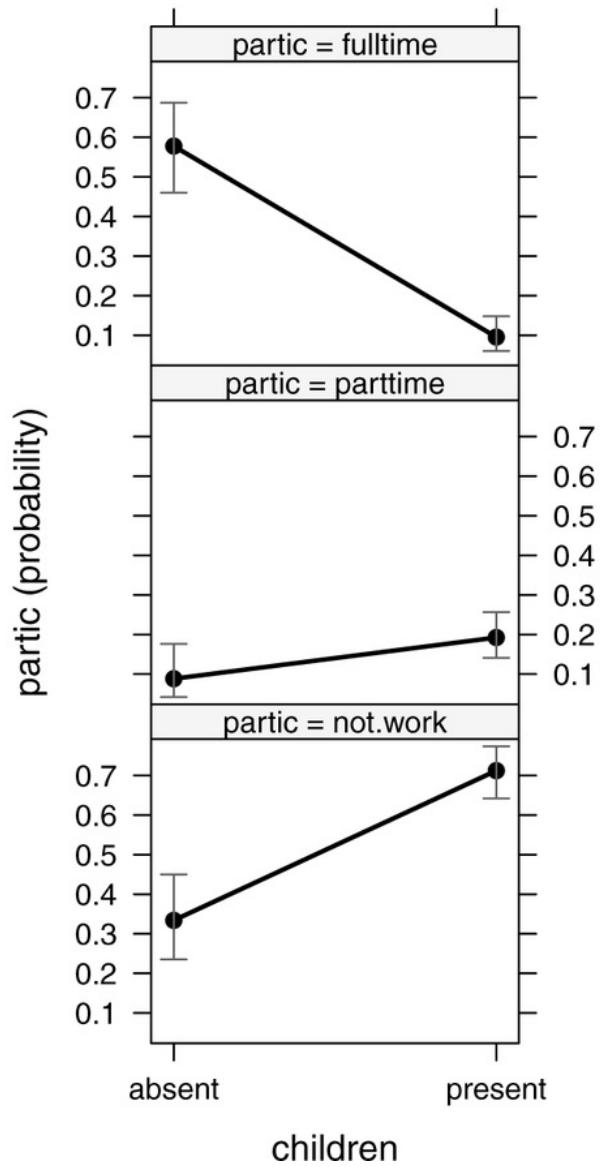
The probability of full-time work decreases with husband's income and decreases when children are present, while the pattern is the opposite for not working. The probability of part-time work increases slightly with husband's income and at fixed levels of husband's income is slightly higher when children are present, but it is not strongly related to either predictor.

A different way of displaying fitted probabilities for a polytomous response is as a *stacked area graph*, produced by the following command; the result appears in [Figure 6.10](#):

```
plot (Effect (c ("hincome", "children"), mod.multinom.1, xlevels=50),  
axes=list (y=list (style="stacked")), lines=list (col=gray (c (0.2, 0.5, 0.8))))
```

Confidence limits aren't shown in the stacked area graph. We specified the optional argument xlevels=50 to `Effect ()` to produce smoother boundaries between the areas (the default is to use five values) and the `lines` argument to `plot ()` to set gray levels for the areas in the monochrome graph produced for this book.

**Figure 6.8** Predictor effect plots for `hincome` and `children` in the multinomial logit model `mod.multinom.1` fit to the Canadian women's labor force data.

**hincome predictor effect plot****children predictor effect plot**

## 6.8 Nested Dichotomies

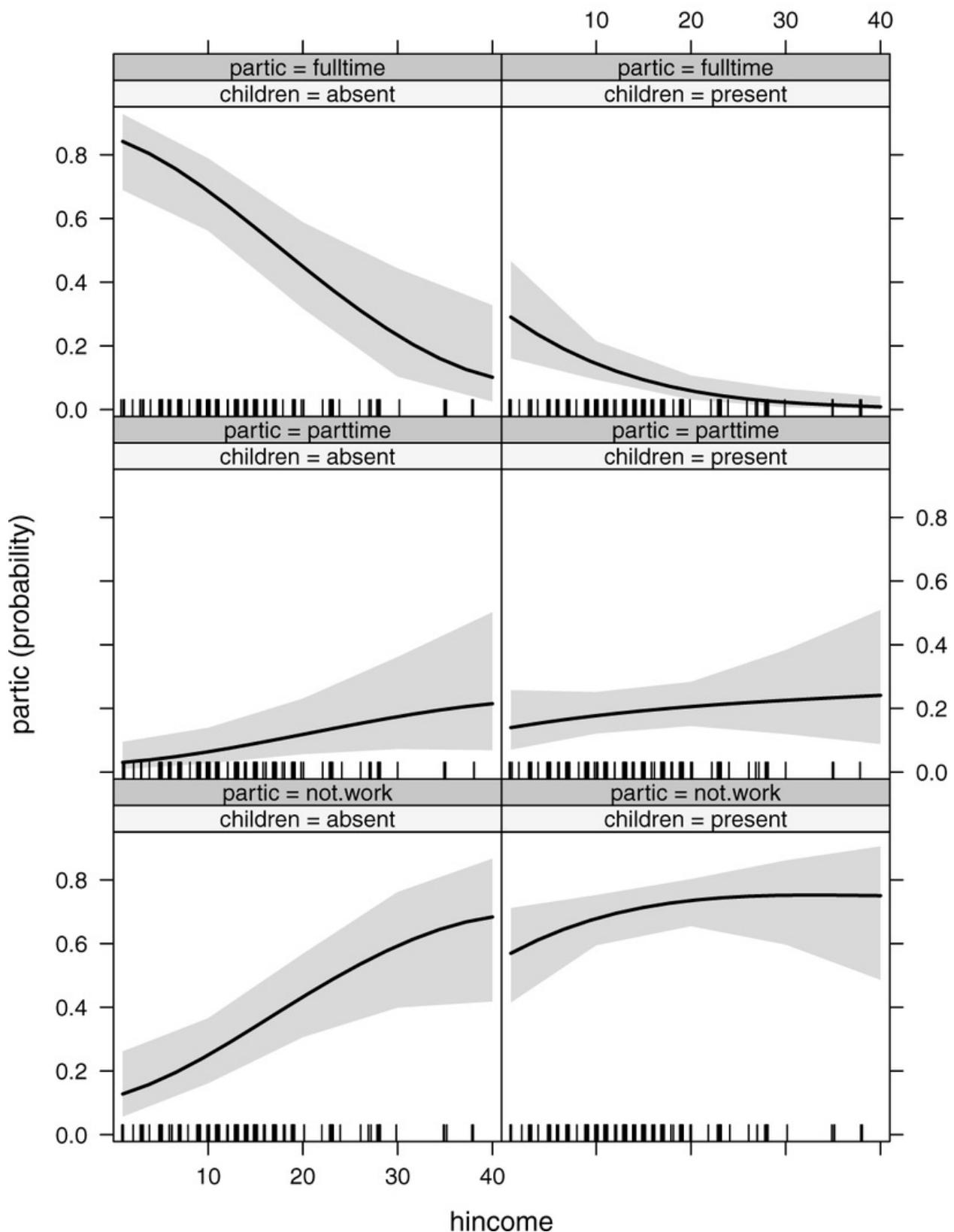
The method of *nested dichotomies* provides an alternative to the multinomial logit model when a single response variable has more than two categories. Nested dichotomies are constructed by successive binary divisions of the levels of the response factor, and then a separate binary (or binomial) logit model is fit to each dichotomy. These binary logit models, which are statistically independent, may be combined to produce a model for the polytomous response.

If the terms are the same for all of the binary logit models, then the combined nested-dichotomies model has the same number of parameters as a multinomial logit model fit to the data. The two models are not equivalent, however: In general, the nested-dichotomies model and the multinomial logit model produce different fitted response probabilities.

Even when there are just three response categories, as in the example of the preceding section, there is more than one way of defining nested dichotomies, and, unlike the multinomial logit model, where arbitrary choice of a baseline level for the response doesn't affect the fit of the model to the data, the combined model for nested dichotomies depends on the specific set of dichotomies selected. As a consequence, this approach is sensible only when a *particular* choice of nested dichotomies is substantively compelling.

**Figure 6.9** Fitted probability of working full-time, part-time, and not working outside the home by husband's income and presence of children, from the multinomial logit model fit to the Canadian women's labor force data.

### hincome\*children effect plot

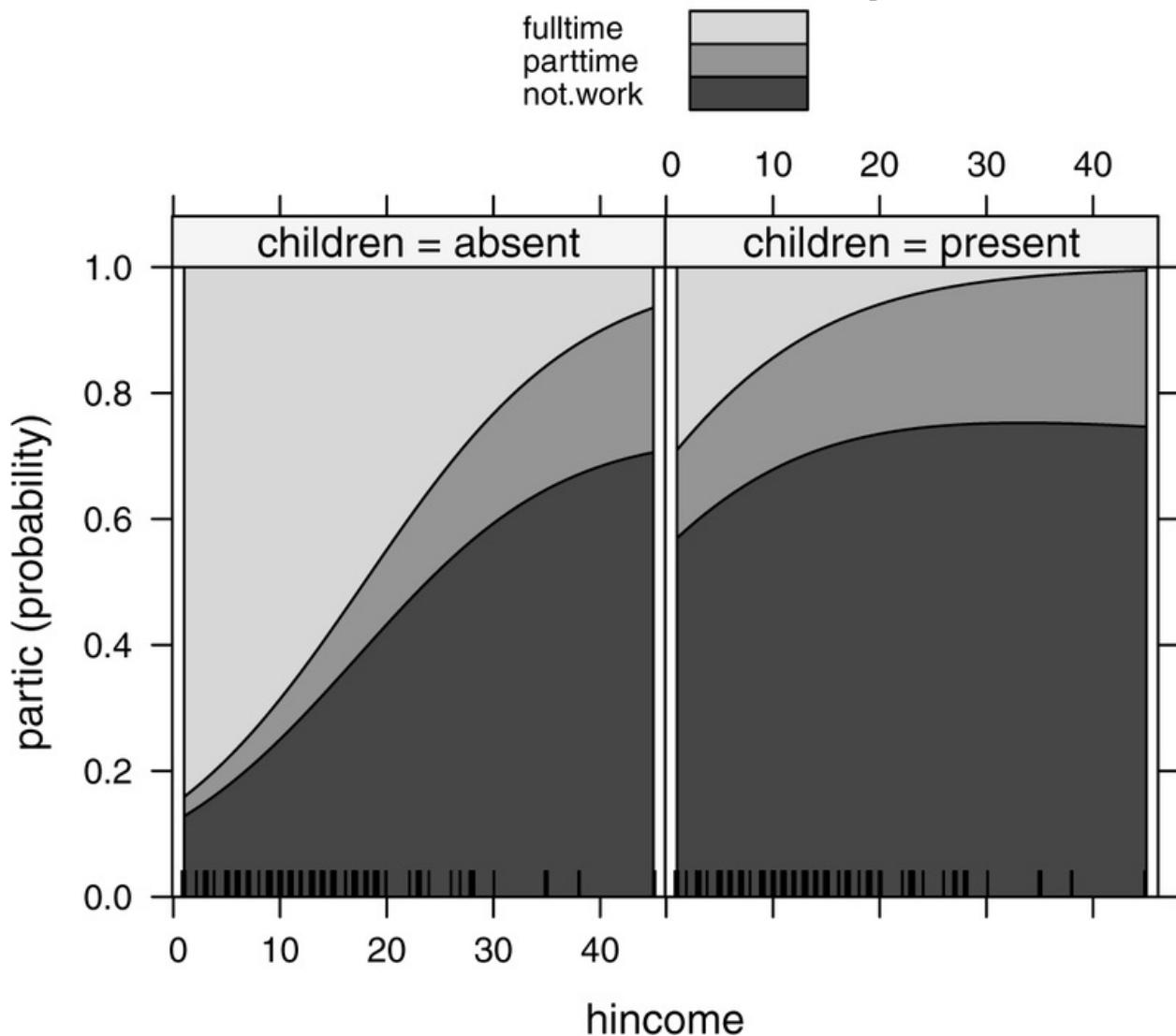


One area in which application of nested dichotomies is natural is in research on educational attainment. In the interest of brevity, we'll simply sketch an example here: Suppose that we have the four-category response factor education with levels "less than high school", "high school graduate", "some postsecondary", and "postsecondary degree". Because individuals normally proceed serially through these levels, it makes sense to focus on the following set of nested dichotomies:

1. "less than high school" versus {"high school graduate", "some postsecondary", or "postsecondary degree"}
2. "high school graduate" versus {"some postsecondary" or "postsecondary degree"}
3. "some postsecondary" versus "postsecondary degree"

**Figure 6.10** Stacked area graph showing the fitted probability of working full-time, part-time, and not working outside the home by husband's income and presence of children.

## hincome\*children effect plot



The first dichotomy represents graduation from high school; the second, moving on to post-secondary education having graduated high school; and the third, obtaining a degree having entered postsecondary education. Only the first dichotomy includes all of the individuals in the data set; for example, the second dichotomy includes data only for high school graduates—those with less than high school education are excluded. Typically, the same predictors are used in the binary logit models fit to the three dichotomies, but this isn't necessarily the case, and in any event, coefficients are not constrained to be the same in the three models.

## 6.9 The Proportional-Odds Model

The nested dichotomies for level of education described in the previous section rely on the order of the categories in the response factor. There are several other statistical models for ordinal responses, developed, for example, in Agresti (2010), Clogg and Shihadeh (1994), and Powers and Xie (2008). The most common ordinal-regression model is the *proportional-odds logistic-regression model*, discussed in this section.

The proportional-odds model can be derived from a linear regression with a *latent response variable*  $\xi$ , a continuous response variable that isn't directly observable. To fix ideas, think of  $\xi$  as the achievement level of students in a particular class. Suppose, further, that if  $\xi$  were observable, it could be expressed as a linear combination of the regressors  $x_1, \dots, x_k$  plus an intercept and an error term,<sup>21</sup>

$$\xi = \alpha + \beta_1 x_1 + \cdots + \beta_k x_k + \varepsilon$$

<sup>21</sup> It is convenient to use  $\alpha$  for the intercept here rather than  $\beta^0$  for reasons that will become clear below.

If  $\xi$  were observable and we knew the distribution of the errors  $\varepsilon$ , then we could estimate the regression coefficients using standard methods such as least-squares regression. Rather than observing  $\xi$ , we instead observe a variable  $y$  that is derived from  $\xi$  by dissecting the range  $\xi$  into class intervals. In our imaginary example, the observable variable might be the grade the students received in the course, an ordinal variable consisting of the ordered categories F, D, C, B, and A, from lowest to highest, corresponding consecutively to  $y = 1, 2, 3, 4, 5$ . In general,  $y$  is an ordinal variable with  $q$  levels,  $1, \dots, q$ , such that

$$y = \begin{cases} \text{level 1} & \text{if } -\infty < \xi \leq \alpha_1 \\ \text{level 2} & \text{if } \alpha_1 < \xi \leq \alpha_2 \\ \vdots & \\ \text{level } q-1 & \text{if } \alpha_{q-2} < \xi \leq \alpha_{q-1} \\ \text{level } q & \text{if } \alpha_{q-1} < \xi \leq \infty \end{cases}$$

where  $\alpha_1 < \dots < \alpha_{q-1}$  are *thresholds* between each level of  $y$  and the next. Thus,  $y$  is a crudely measured version of  $\xi$ . The dissection of  $\xi$  into  $q$  levels introduces  $q$

– 1 additional parameters into the model, because the values of the thresholds are not known in advance. In our student grades example,  $q = 5$ , and so there are four unknown thresholds.

The cumulative probability distribution of  $y$  is given by

$$\begin{aligned}\Pr(y \leq j|\mathbf{x}) &= \Pr(\xi \leq \alpha_j|\mathbf{x}) \\ &= \Pr(\alpha + \beta_1 x_1 + \cdots + \beta_k x_k + \varepsilon \leq \alpha_j|\mathbf{x}) \\ &= \Pr(\varepsilon \leq \alpha_j - \alpha - \beta_1 x_1 - \cdots - \beta_k x_k|\mathbf{x})\end{aligned}$$

for  $j = 1, 2, \dots, q - 1$ .

for  $j = 1, 2, \dots, q - 1$ .

The next step depends on a distributional assumption concerning the errors. If we assume that the errors follow a standard logistic distribution, then we get the *proportional-odds logistic-regression model*,

$$\begin{aligned}\text{logit}[\Pr(y > j|\mathbf{x})] &= \log \frac{\Pr(y > j|\mathbf{x})}{\Pr(y \leq j|\mathbf{x})} \\ &= (\alpha - \alpha_j) + \beta_1 x_1 + \cdots + \beta_k x_k\end{aligned}$$

for  $j = 1, 2, \dots, q - 1$ . If we instead assume a standard-normal distribution for  $\varepsilon$ , then we get the *ordered probit model*,

$$\Phi^{-1}[\Pr(y > j|\mathbf{x})] = (\alpha - \alpha_j) + \beta_1 x_1 + \cdots + \beta_k x_k$$

where as usual  $\Phi^{-1}$  is the quantile function of the standard-normal distribution. The logistic and normal distributions are very similar symmetric and unimodal distributions (recall [Figure 6.1](#) on page 278), although the logistic distribution has somewhat heavier tails. As a consequence, in most applications the proportional-odds and ordered probit models produce similar results; we will discuss only the proportional-odds model here.

Because the equations for  $\text{logit}[\Pr(y > j|\mathbf{x})]$  for different values of  $j$  differ only in their intercepts, the regression curves for the cumulative probabilities  $\Pr(y > j|\mathbf{x})$  are parallel in the log-odds scale. There is a constant ratio of the odds—hence

the name “proportional-odds model.”

Assuming that the errors follow a standard logistic distribution fixes the scale of the latent response  $\xi$  but not its origin, and so the general intercept  $\alpha$  is not identified independently of the thresholds  $\alpha^j$ . Setting  $\alpha = 0$  to fix the origin of  $\xi$ , the negatives of the category boundaries (i.e., the  $-\alpha_j$ ) become the intercepts for the  $q - 1$  logistic-regression equations.

The proportional-odds model may be fit in R using the `polr()` function in the **MASS** package (Venables & Ripley, 2002).<sup>22</sup> For the women’s labor force data, we may proceed as follows:<sup>23</sup>

```
library("MASS")
mod.polr <- polr(partic ~ hincome + children, data=Womenlf)
S(mod.polr)
```

Re-fitting to get Hessian

```
Call: polr(formula = partic ~ hincome + children, data =
           Womenlf)
Coefficients:
```

|                 | Estimate | Std. Error | z value | Pr(> z ) |
|-----------------|----------|------------|---------|----------|
| hincome         | -0.0539  | 0.0195     | -2.77   | 0.0057   |
| childrenpresent | -1.9720  | 0.2869     | -6.87   | 6.3e-12  |

Intercepts (Thresholds) :

|                   | Estimate | Std. Error | z value | Pr(> z ) |
|-------------------|----------|------------|---------|----------|
| not.work parttime | -1.852   | 0.386      | -4.79   | 1.6e-06  |
| parttime fulltime | -0.941   | 0.370      | -2.54   | 0.011    |

Residual Deviance: 441.66

| logLik  | df | AIC    | BIC    |
|---------|----|--------|--------|
| -220.83 | 4  | 449.66 | 463.95 |

[22](#) The polr () function can also be used to fit some other similar models for an ordered response, including the ordered probit model. The **ordinal** package includes three functions clm (), clm2(), and clmm () (R. H. B. Christensen, 2018) that fit ordinal regression models with restrictions on the thresholds and with the inclusion of random effects. The vglm () function in the **VGAM** package (Yee, 2015) can also fit a variety of threshold-based models to ordinal data.

[23](#) The message “Re-fitting to get Hessian” is produced because S () needs the standard errors for the coefficients of the proportional-odds model. The *Hessian matrix* of second-order partial derivatives of the log-likelihood is used to compute variances and covariances for the coefficients. By default, the Hessian isn’t computed by polr ().

The fit of the proportional-odds model, shown in [Figure 6.11](#), is quite different from that of the multinomial logit model ([Figure 6.9](#) on page 315):

```
plot(Effect(c("hincome", "children"), mod.polr))
```

Re-fitting to get Hessian

Alternative effect plots for the proportional-odds model are shown in [Figure 6.12](#), using stacked-area plots, and in [Figure 6.13](#), plotting the estimated latent response:

```
plot(Effect(c("hincome", "children"), mod.polr),
      axes=list(y=list(style="stacked")),
      lines=list(col=gray(c(.2, .5, .8))))
```

Re-fitting to get Hessian

```
plot(Effect(c("hincome", "children"), mod.polr, latent=TRUE))
```

Re-fitting to get Hessian

## 6.9.1 Testing for Proportional Odds

An attractive feature of the proportional-odds model is that it uses a single linear predictor rather than  $q - 1$  linear predictors, as in the multinomial logit and

nested-dichotomies models, reducing the number of parameters to just  $q - 1$  thresholds and  $k$  regression coefficients, rather than  $(k + 1) \times (q - 1)$  parameters. Of course, the proportional-odds model is therefore more restrictive than the other two models and may not fit the data well.

Because the proportional-odds model is not a specialization of either the nested-dichotomies model or of the multinomial logit model, we can't use likelihood-ratio tests to compare these models, but comparisons based on the AIC, for example, can be informative. To illustrate, the multinomial logit model we fit to the Canadian women's labor force data has a much smaller AIC than the proportional-odds model, suggesting the inadequacy of the latter:

**AIC (mod.multinom.1)**

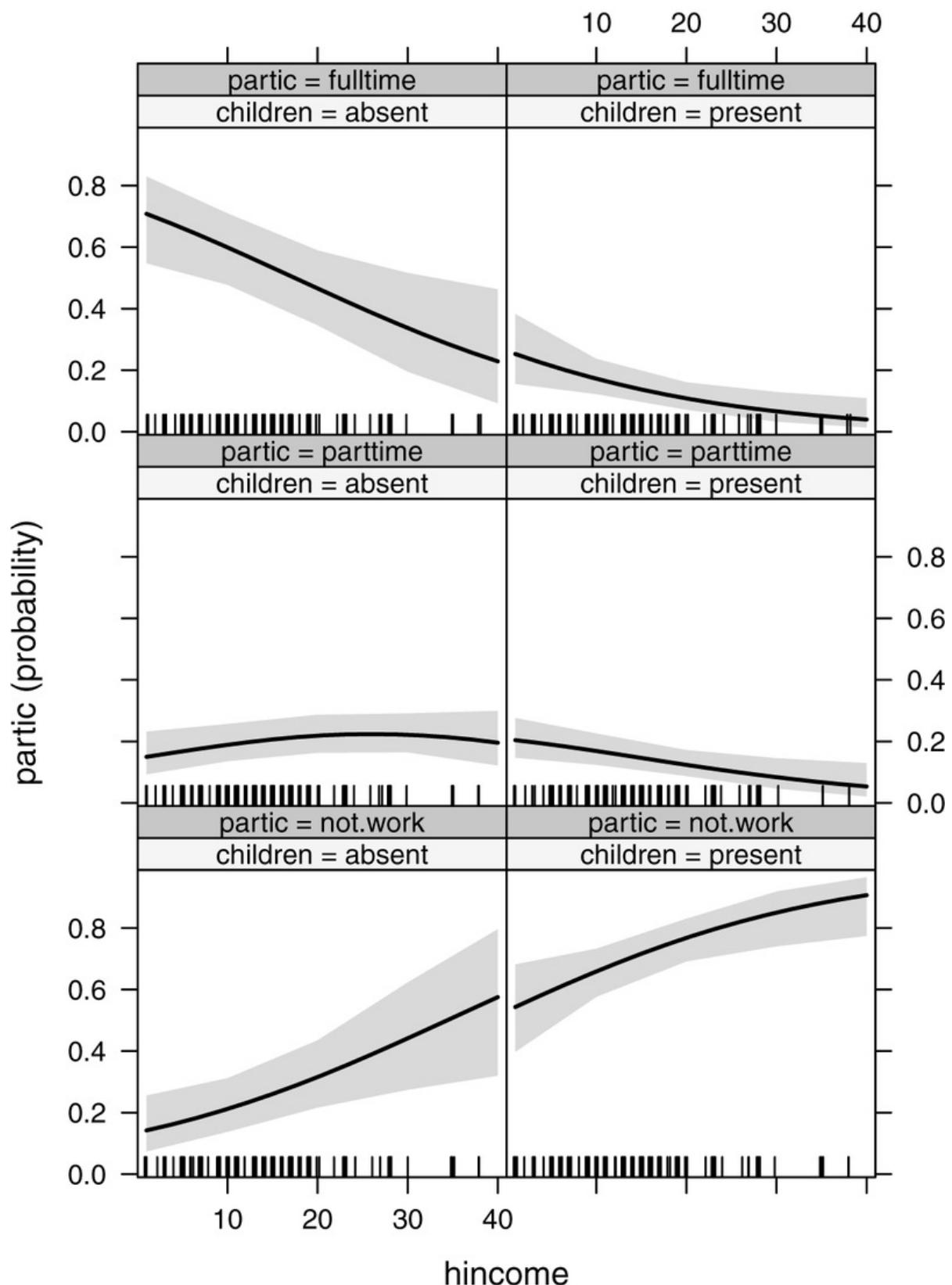
```
[1] 434.88
```

**AIC (mod.polr)**

```
[1] 449.66
```

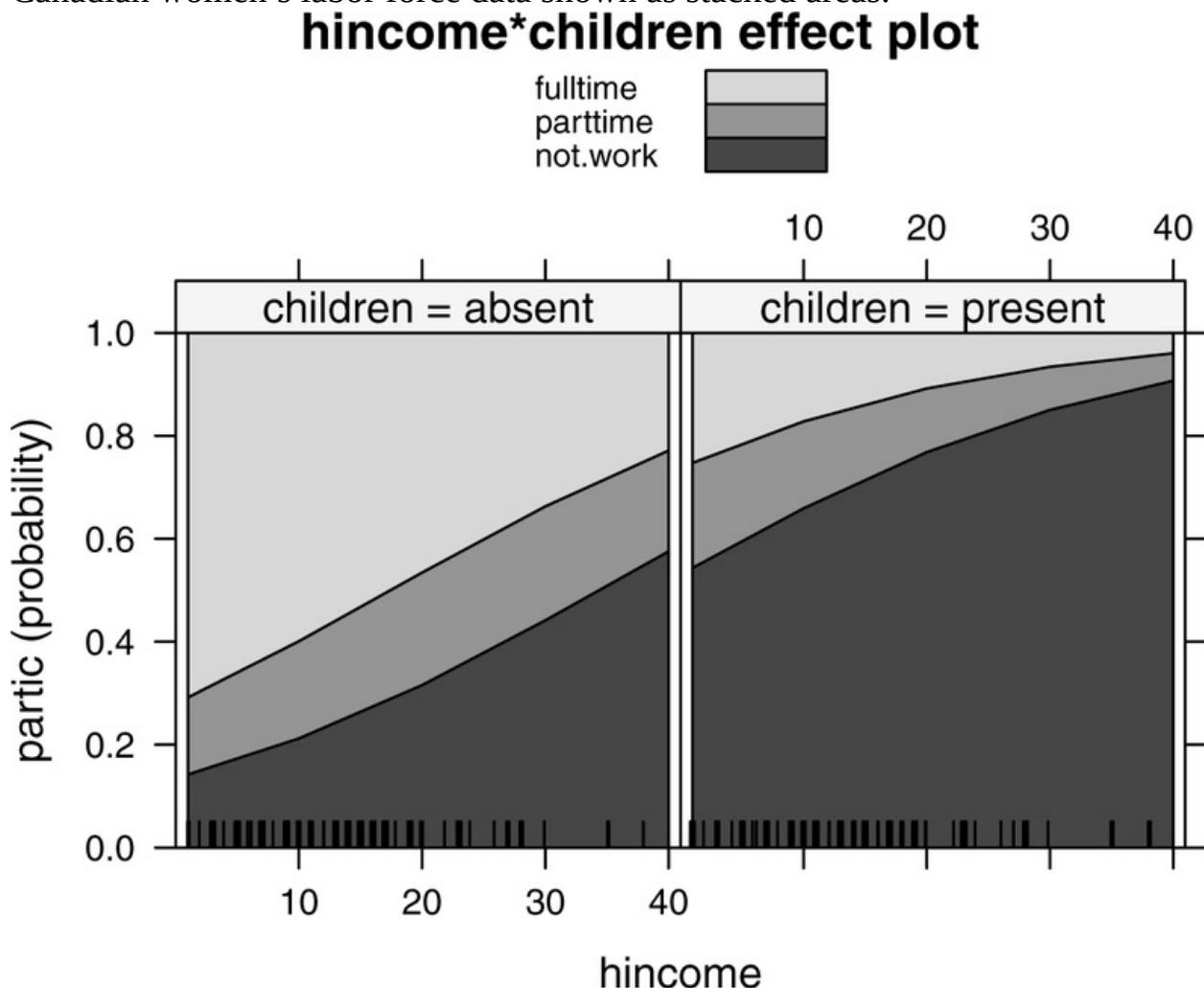
**Figure 6.11** Estimated probability of working full-time, part-time, and not working outside the home by husband's income and presence of children, from the proportional-odds model fit to the Canadian women's labor force data. The bands, showing 95% pointwise confidence envelopes around the fitted probabilities, are based on standard errors computed by the delta method (Fox & Andersen, 2006).

## hincome\*children effect plot



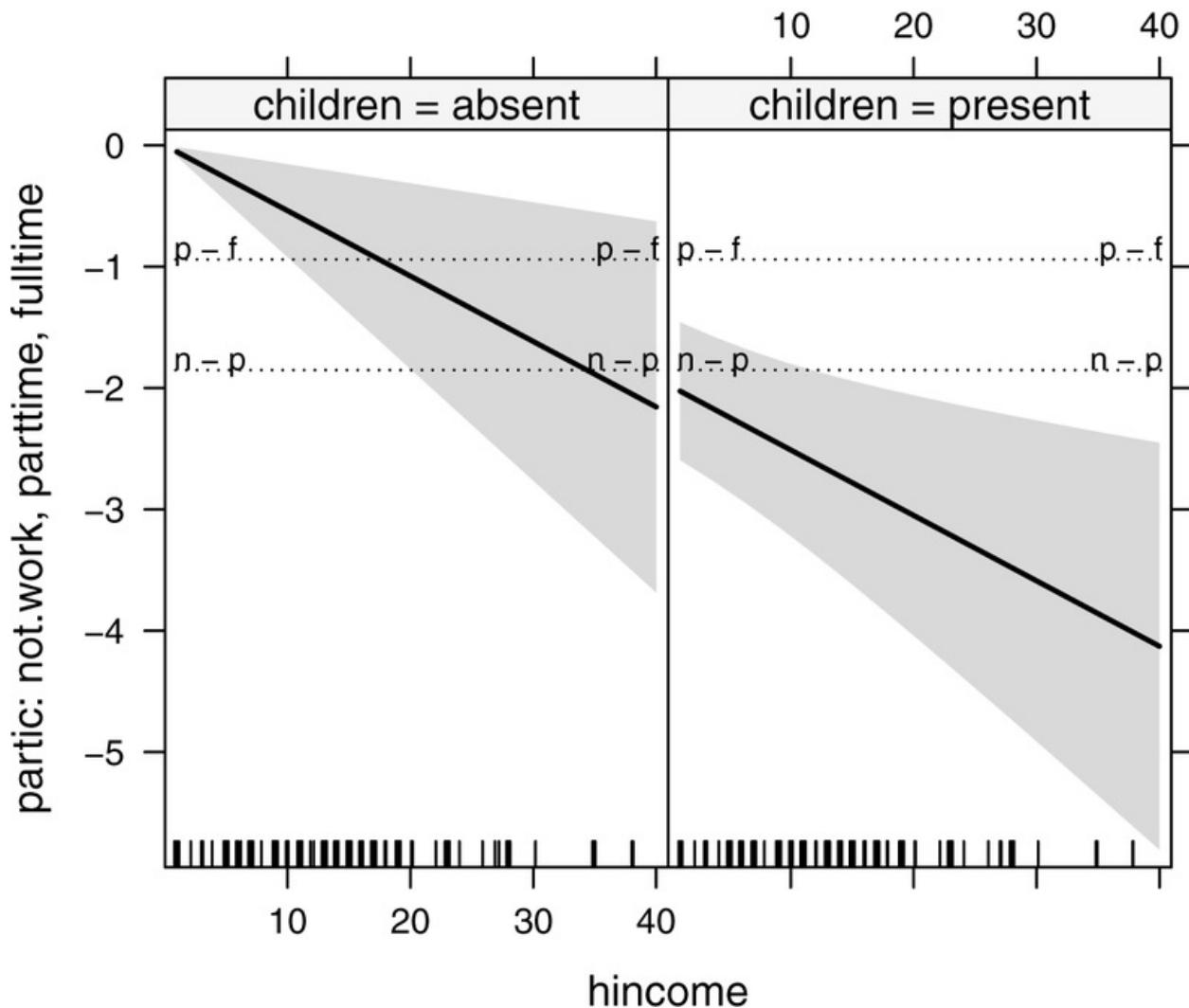
There are several tests of the proportional-odds assumption, or, equivalently, the assumption that the  $k$  regression coefficients in the linear predictor (the “slopes”) are the same for all  $q - 1$  cumulative logits,  $\text{logit}[\Pr(y > 1)]$ , ...,  $\text{logit}[\Pr(y = q)]$ . A straightforward approach is to specify a more general model for the cumulative logits that permits unequal slopes, followed by a likelihood-ratio test comparing the proportional-odds model to its more general alternative, but we do not pursue this possibility here.<sup>24</sup>

**Figure 6.12** Estimated probabilities from the proportional-odds model fit to the Canadian women’s labor force data shown as stacked areas.



**Figure 6.13** The estimated latent response from the proportional-odds model fit to the Canadian women’s labor force data. The dotted horizontal lines are the estimated thresholds between adjacent levels of the observed response, which are abbreviated as n (“not.work”), p (“parttime”), and f (“fulltime”).

## hincome\*children effect plot



[24](#) The unequal-slopes model may be fit by the `clm()` function in the **ordinal** package and the `vglm()`

Another approach, suggested by Brant (1990), is to base a test on separate binary logit models fit to the various cumulative logits, taking into account not just the variances of the estimated coefficients but also their covariances. This test is available from the `poTest()` function in the **car** package:

### **poTest (mod.polr)**

```
Tests for Proportional Odds
polr(formula = partic ~ hincome + children, data = Womenlf)

          b[polr] b[>not.work] b[>parttime] Chisquare df
Overall                               19.60  2
hincome      -0.0539      -0.0423      -0.0987    6.16  1
childrenpresent -1.9720      -1.5756      -2.5631   15.41  1
Pr(>Chisq)
Overall        5.6e-05
hincome         0.013
childrenpresent 8.6e-05
```

The `poTest ()` function provides both an overall test of the proportional-odds assumption and separate tests for each regressor, all of which produce small  $p$ -values for our model, casting doubt on the assumption of proportional odds. The output also displays the estimated coefficients for the binary logit models fit to the two cumulative logits for the three-level response, alongside the estimated coefficients for the proportional-odds model.

It's our experience that the proportional-odds assumption is rarely supported by a hypothesis test. The proportional-odds model can nevertheless provide useful simplification of the data as long as fitted probabilities under the model aren't substantially distorted—which is sadly *not* the case for our example.

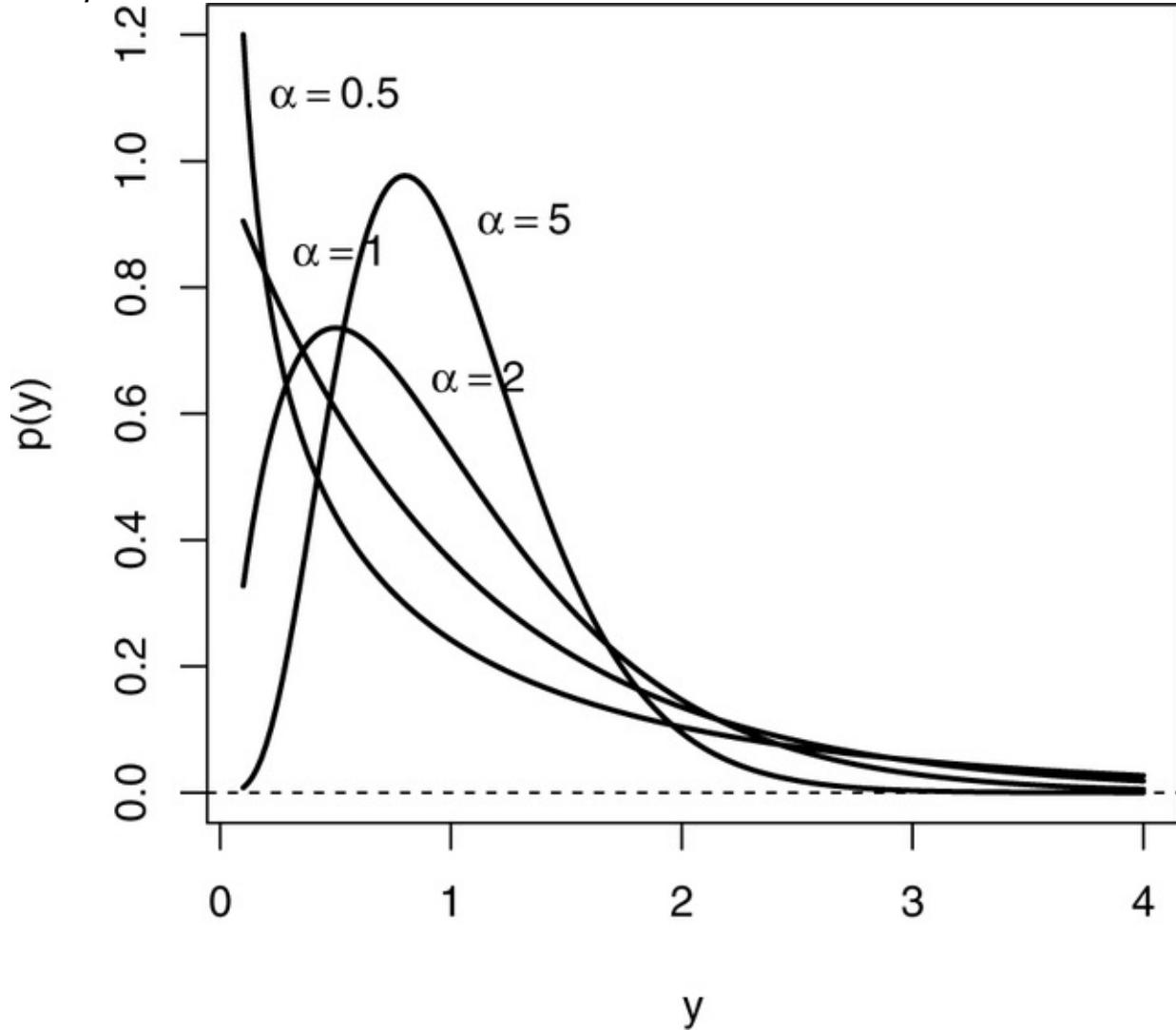
## 6.10 Extensions

### 6.10.1 More on the Anova () Function

The primary use of the `Anova ()` function is to calculate Type II tests of hypotheses about terms in a model with a linear predictor. For a GLM, the `Anova ()` function computes likelihood-ratio chi-square tests by default, but it can optionally calculate Type II Wald chi-square tests or, for models that have an estimated dispersion parameter,  $F$ -tests; the various tests are selected via the `test.statistic` argument. The `Anova ()` function can also perform Type III tests for GLMs, specified by the `type` argument.<sup>25</sup> The default method for the `Anova ()` generic function computes Type II, and optionally Type III, Wald chi-square

tests for many classes of statistical models with linear predictors.

**Figure 6.14** Gamma densities, for various values of the shape parameter,  $\alpha$ , and with  $\mu = 1$  fixed.



[25](#) As in the case of linear models, Type III tests for GLMs require careful formulation of the regressors for the model; see [Section 5.3.4](#).

Likelihood-ratio tests and  $F$ -tests require fitting more than one model to the data, while Wald tests do not. The  $F$ -tests computed by Anova () base the estimated dispersion on Pearson's  $X^2$  by default and therefore are not the same as likelihood-ratio tests even when, as in the binomial and Poisson models, the dispersion is fixed. The Pearson statistic is the sum of squared Pearson residuals from the GLM. For terms with 1  $df$  in an additive model, the Wald  $\chi^2$  statistics

provided by `Anova()` are simply the squares of the corresponding Wald z-scores printed by the `S()` function.

## 6.10.2 Gamma Models

Generalized linear models with errors from a gamma distribution are used much less often than are binomial, Poisson, or normal models. A gamma distribution can be appropriate for a strictly positive response variable.<sup>26</sup> Like the normal distribution, the gamma distribution has two parameters, which can be taken to be a mean parameter  $\mu$  and a positive *shape parameter*  $\alpha$ . Unlike the normal distribution, the shape of the corresponding gamma density function changes with the parameters, as shown in [Figure 6.14](#) for fixed  $\mu = 1$  and varying  $\alpha$ . For  $\alpha \leq 1$ , the density has a pole at the origin, while for  $\alpha > 1$ , it is unimodal but positively skewed. The skewness decreases as  $\alpha$  increases. Because the variance of a gamma distribution is  $\mu^2/\alpha$ , the shape parameter is the inverse of the dispersion parameter  $\phi$  in the usual notation for a GLM. A characteristic of the gamma distribution is that the *coefficient of variation*, the ratio of the standard deviation to the mean, is constant and equal to  $\sqrt{\alpha}$ .

[26](#) The inverse-Gaussian family is also appropriate for continuous positive data, but applications are even more unusual.

An example for which gamma errors may be useful is the `Transact` data set, introduced in [Section 4.2.3](#). Errors in these data are likely to increase with the size of the response, and so the gamma assumption of constant coefficient of variation may be reasonable, as suggested by Cunningham and Heathcote (1989):

```

trans.gamma <- glm(time ~ t1 + t2, family=Gamma(link=identity),
  data=Transact)
S(trans.gamma)

```

Call: `glm(formula = time ~ t1 + t2, family = Gamma(link = identity), data = Transact)`

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 152.951  | 51.800     | 2.95    | 0.0034   |
| t1          | 5.706    | 0.426      | 13.41   | <2e-16   |
| t2          | 2.007    | 0.058      | 34.61   | <2e-16   |

(Dispersion parameter for Gamma family taken to be 0.029387)

Null deviance: 92.603 on 260 degrees of freedom  
 Residual deviance: 7.477 on 258 degrees of freedom

| logLik  | df | AIC    | BIC    |
|---------|----|--------|--------|
| -2156.8 | 4  | 4321.6 | 4335.8 |

Number of Fisher Scoring iterations: 4

The canonical link for the Gamma family is the inverse link (see [Table 6.2](#) on page 274), and if we were to use that link, we would then model the mean for time as  $E(\text{time}|t_1, t_2) = 1/(\beta^0 + \beta^1 t_1 + \beta^2 t_2)$ . Using the identity link instead, the coefficients of the gamma regression model have the same interpretations as the coefficients of the linear model that we fit by least squares in [Section 4.2.3](#): The intercept is the average amount of time required independent of transactions, the coefficient of  $t_1$  is the typical time required for each Type 1 transaction, and the coefficient of  $t_2$  is the typical time required for each Type 2 transaction. The inverse scale loses this simple interpretation of coefficients. Using the identity link with the Gamma family can be problematic because it can lead to negative fitted values for a strictly positive response, a problem that doesn't occur for our example.

The estimate of the shape parameter, which is the inverse of the dispersion parameter in the GLM, is returned by the `gamma.shape()` function in the **MASS** package:

Alpha: 35.0729  
***gamma.shape(trans.gamma)*** SE: 3.0557

The large estimated shape implies that the error distribution is nearly symmetric.

In [Section 5.1.4](#), having fit a linear model to the Transact data, we used the deltaMethod () function to compute an approximate standard error and confidence limits for the ratio of regression coefficients for the predictors t1 and t2. Comparing the results obtained from the delta method to those produced by bootstrapping, we found that we could compute a more accurate delta method standard error when we corrected for nonconstant error variance. Let's repeat this analysis for the gamma regression, in which nonconstant variance is built into the model, as usual setting the seed for R's random number generator to make the computation reproducible:<sup>27</sup>

```
set.seed(12134) # for reproducibility
deltaMethod(trans.gamma, "t1/t2")

      Estimate      SE  2.5 % 97.5 %
t1/t2    2.8427 0.27676 2.3002 3.3851

boot.trans <- Boot(trans.gamma)
ratio <- boot.trans$t[, "t1"]/boot.trans$t[, "t2"]
c("t1/t2"=as.vector(coef(trans.gamma)[2]/coef(trans.gamma)[3]),
  bootSE=sd(ratio), quantile(ratio, c(0.025, 0.975)))

      t1/t2   bootSE     2.5%   97.5%
      2.84267 0.31794 2.27031 3.49586
```

<sup>27</sup> See [Section 10.7](#) for more information about random simulations in R.

The call to as.vector removes the name from the ratio of coefficients, for which we supply the name "t1/t2". The bootstrap standard error and percentile confidence interval for the gamma regression are much closer to those produced by the delta method than was the case for the linear model that we fit by least squares, where the uncorrected delta method standard error was much too small. Although the point estimate of the ratio of coefficients is similar for the linear and gamma models, the realistic standard error of the estimate (e.g., produced by the bootstrap) is much smaller for the gamma model than for the linear model.

### 6.10.3 Quasi-Likelihood Estimation

The likelihood functions for exponential families depend only on the mean and variance of the conditional distribution of  $y$ . The essential idea of *quasi-likelihood estimation* is to use the same maximizing function for any distribution that matches the first two moments (i.e., the mean and variance) of an exponential family distribution. It can be shown (McCullagh & Nelder, 1989, chap. 9) that most of the desirable properties of maximum-likelihood estimates for exponential families—including asymptotic normality, asymptotic unbiasedness, and the usual covariance matrix for the estimates—are shared by estimates from distributions that match the first two moments of an exponential family.

This is a familiar idea: When we apply least-squares regression to a model with nonnormal errors, for example, the resulting estimates are unbiased, are asymptotically normal, and have the usual covariance matrix, as long as the assumptions of linearity, constant error variance, and independence hold. When the errors are nonnormal, the least-squares estimates are not in general maximum-likelihood estimates, but are still maximally efficient among linear unbiased estimators (by the Gauss-Markov theorem), though no longer necessarily among *all* unbiased estimators.

In R, quasi-likelihood estimation for GLMs is achieved by specifying the quasi family generator function, with link and variance as arguments. These arguments default to "identity" and "constant", respectively, a combination that yields linear least-squares estimates. Of course, there is no reason to compute the least-squares estimates in this convoluted manner. There are also the special families `quasipoisson()` and `quasibinomial()` for quasi-likelihood estimation of overdispersed Poisson and binomial GLMs, which we discuss next.

### 6.10.4 Overdispersed Binomial and Poisson Models

The term *overdispersion* means that the observed conditional variance of the response is larger than the variation implied by the distribution used in fitting the model.<sup>28</sup> Overdispersion can be symptomatic of several different problems:

1. Observations on different individuals with the same values of the regressors  $\mathbf{x}$  do not have exactly the same distribution; that is, there are unaccounted-

for individual differences that produce additional variation. This situation is often termed *unmodeled heterogeneity*.

2. Observations may be correlated or clustered, while the specified variance function wrongly assumes uncorrelated data.
3. The specified mean function is wrong, and this misspecification “leaks” into the estimated variance function.

[28](#) It is also possible for binomial or Poisson data to be *underdispersed*, but this is uncommon in applications.

The first two problems are addressed in this section. The third problem will be discussed in [Chapter 8](#).

One approach to overdispersion for a binomial or Poisson GLM is to estimate the dispersion parameter  $\phi$  rather than use the assumed value  $\phi = 1$  for the Poisson or binomial distributions. The usual estimator of the dispersion parameter is , the value of Pearson’s  $X^2$  divided by the residual  $df$ . Estimating the dispersion has no effect on the coefficient estimates, but it inflates all of their standard errors by the factor

## Quasi-Likelihood and Estimating $\phi$

As an example, the model `ornstein.mod` fit to Ornstein’s interlocking-directorate data in [Section 6.5](#) has residual deviance 1,547.1 with 234  $df$ . The large value of the deviance when compared to its degrees of freedom indicates possible overdispersion.[29](#) The estimator of  $\phi$  is

```
(phihat <- sum(residuals(mod.ornstein, type="pearson")^2)/df.residual(mod.ornstein))
```

```
[1] 6.3986
```

[29](#) This is so for both Poisson data and for binomial data with binomial denominators larger than 1—that is, binomial rather than binary data.

Standard errors and Wald tests adjusted for overdispersion may then be obtained by the command



**S(mod.ornstein, dispersion=phihat, brief=TRUE)**

Coefficients:

|              | Estimate | Std. Error | z value | Pr(> z ) |
|--------------|----------|------------|---------|----------|
| (Intercept)  | -0.8394  | 0.1366     | -6.14   | 8.1e-10  |
| log2(assets) | 0.3129   | 0.0118     | 26.58   | < 2e-16  |
| nationOTH    | -0.1070  | 0.0744     | -1.44   | 0.15030  |
| nationUK     | -0.3872  | 0.0895     | -4.33   | 1.5e-05  |
| nationUS     | -0.7724  | 0.0496     | -15.56  | < 2e-16  |
| sectorBNK    | -0.1665  | 0.0958     | -1.74   | 0.08204  |
| sectorCON    | -0.4893  | 0.2132     | -2.29   | 0.02174  |
| sectorFIN    | -0.1116  | 0.0757     | -1.47   | 0.14046  |
| sectorHLD    | -0.0149  | 0.1192     | -0.13   | 0.90051  |
| sectorMAN    | 0.1219   | 0.0761     | 1.60    | 0.10949  |
| sectorMER    | 0.0616   | 0.0867     | 0.71    | 0.47760  |
| sectorMIN    | 0.2498   | 0.0689     | 3.63    | 0.00029  |
| sectorTRN    | 0.1518   | 0.0789     | 1.92    | 0.05445  |
| sectorWOD    | 0.4983   | 0.0756     | 6.59    | 4.4e-11  |

(Dispersion parameter for poisson family taken to be 6.3986)

Null deviance: 3737.0 on 247 degrees of freedom  
Residual deviance: 1547.1 on 234 degrees of freedom

| logLik  | df | AIC    | BIC    |
|---------|----|--------|--------|
| -1222.5 | 14 | 2473.1 | 2522.3 |

Number of Fisher Scoring iterations: 5

The Anova () function in the **car** package uses this estimate of  $\phi$  to get tests if we specify test.statistic="F":

```
Anova (mod.ornstein, test.statistic="F")
```

Analysis of Deviance Table (Type II tests)

Response: interlocks

|              | Sum Sq | Df  | F value | Pr(>F)  |
|--------------|--------|-----|---------|---------|
| log2(assets) | 731    | 1   | 114.28  | < 2e-16 |
| nation       | 276    | 3   | 14.38   | 1.2e-08 |
| sector       | 103    | 9   | 1.78    | 0.072   |
| Residuals    | 1497   | 234 |         |         |

The estimated dispersion, , is substantially greater than 1, producing much larger standard errors and thus smaller test statistics than were obtained from the standard Poisson regression model (fit to the Ornstein data in [Section 6.5](#)). An identical analysis is produced using the argument family=quasipoisson with glm() (as the reader can verify):

```
mod.ornstein.q <- update (mod.ornstein, family=quasipoisson)
```

## Negative-Binomial Regression

An alternative strategy for overdispersed count data is to model between-individual variability. One useful approach leads to the *negative-binomial regression model* for data that might otherwise be modeled with a Poisson distribution. We introduce a *random subject effect*  $S$  that has a different value for each subject<sup>30</sup> and assume that the conditional distribution of the response given the predictors and the subject effect ( $y|\mathbf{x}, S$ ) is a Poisson distribution with mean  $S \times \mu(\mathbf{x})$ . The new feature is that the marginal distribution of  $S$  is a gamma distribution with mean parameter  $\theta$  and shape parameter  $\alpha$ .<sup>31</sup> It then follows (e.g., Venables & Ripley, 2002, [Section 7.4](#)) that  $E(y|\mathbf{x}) = \mu(\mathbf{x})$  and  $\text{Var}(y|\mathbf{x}) = \mu(\mathbf{x}) + \mu(\mathbf{x})^2/\theta$ . The second term in the variance provides the overdispersion relative to the Poisson model. If  $\theta$  is small compared to  $\mu(\mathbf{x})$ , then overdispersion can be substantial. As  $\theta \rightarrow \infty$ , the negative-binomial model approaches the

Poisson.

[30](#) We'll explore the notion of random effects more generally in [Chapter 7](#) on mixed-effects models.

[31](#) In [Section 6.10.2](#), we used the symbol  $\mu$  for the mean of a gamma distribution, but doing so here would confuse the mean of the marginal distribution of the random effect  $S$  with the mean function for the conditional distribution of  $y|\mathbf{x}, S$ . This notation is also consistent with the labeling of output in the `glm.nb()` function.

The negative-binomial distribution only fits into the framework of GLMs when the value of  $\theta$  is assumed known. We can then fit negative-binomial regression models with `glm()` by using the `negative.binomial()` family generator provided by the **MASS** package. The default link is the log link, as for the Poisson family, and therefore the regression coefficients have the same interpretation as in Poisson regression.

For example, setting  $\theta = 1.5$ :

```
mod.ornstein.nb <- update (mod.ornstein, family=negative.binomial (1.5))
```

To select a value for  $\theta$ , we can define a grid of reasonable values and then choose the one that minimizes the AIC:[32](#)

```

thetas <- seq(0.5, 2.5, by=0.5)
aics <- rep(0, 5) # allocate vector
for (i in seq(along=thetas)) {
  aics[i] <- AIC(update(mod.ornstein.nb,
    family=negative.binomial(thetas[i])))
}
names(aics) <- thetas
aics

```

|        |        |        |        |        |
|--------|--------|--------|--------|--------|
| 0.5    | 1      | 1.5    | 2      | 2.5    |
| 1772.9 | 1691.0 | 1673.8 | 1676.2 | 1686.3 |

32 All of these models have the same number of parameters, so the parsimony penalties for the AIC are all the same. We can't compare deviances for the models directly because the scale changes with the value of  $\theta$ .

The minimum AIC is therefore at about  $\theta = 1.5$ , the value we used above:

**brief(mod.ornstein.nb)**

|               | (Intercept) | log2(assets) | nationOTH | nationUK  |           |
|---------------|-------------|--------------|-----------|-----------|-----------|
| Estimate      | -0.827      | 0.3164       | -0.104    | -0.389    |           |
| Std. Error    | 0.372       | 0.0352       | 0.226     | 0.231     |           |
| exp(Estimate) | 0.437       | 1.3722       | 0.901     | 0.678     |           |
|               | nationUS    | sectorBNK    | sectorCON | sectorFIN | sectorHLD |
| Estimate      | -0.788      | -0.411       | -0.762    | -0.104    | -0.213    |
| Std. Error    | 0.129       | 0.371        | 0.445     | 0.247     | 0.343     |
| exp(Estimate) | 0.455       | 0.663        | 0.467     | 0.902     | 0.808     |
|               | sectorMAN   | sectorMER    | sectorMIN | sectorTRN | sectorWOD |
| Estimate      | 0.0766      | 0.0787       | 0.239     | 0.101     | 0.390     |
| Std. Error    | 0.1818      | 0.2276       | 0.184     | 0.243     | 0.228     |
| exp(Estimate) | 1.0796      | 1.0819       | 1.271     | 1.107     | 1.478     |

Residual deviance = 280 on 234 df, Est. dispersion = 0.891

An alternative is to estimate  $\theta$  along with the regression coefficients. The

resulting model, which is not a traditional GLM, can be fit with the `glm.nb()` function in the **MASS** package, for which the default link is again the log link:

```
summary(glm.nb(interlocks ~ log2(assets) + nation + sector,  
               data=Ornstein))
```

Call:

```
glm.nb(formula = interlocks ~ log2(assets) + nation + sector,  
       data = Ornstein, init.theta = 1.639034209, link = log)
```

Deviance Residuals:

| Min    | 1Q     | Median | 3Q    | Max   |
|--------|--------|--------|-------|-------|
| -2.809 | -0.990 | -0.189 | 0.430 | 2.408 |

Coefficients:

|              | Estimate | Std. Error | z value | Pr(> z ) |
|--------------|----------|------------|---------|----------|
| (Intercept)  | -0.8254  | 0.3798     | -2.17   | 0.030    |
| log2(assets) | 0.3162   | 0.0359     | 8.80    | < 2e-16  |
| nationOTH    | -0.1045  | 0.2300     | -0.45   | 0.649    |
| nationUK     | -0.3894  | 0.2357     | -1.65   | 0.099    |
| nationUS     | -0.7882  | 0.1320     | -5.97   | 2.4e-09  |
| sectorBNK    | -0.4085  | 0.3773     | -1.08   | 0.279    |
| sectorCON    | -0.7570  | 0.4571     | -1.66   | 0.098    |
| sectorFIN    | -0.1035  | 0.2518     | -0.41   | 0.681    |
| sectorHLD    | -0.2110  | 0.3498     | -0.60   | 0.546    |
| sectorMAN    | 0.0768   | 0.1860     | 0.41    | 0.680    |
| sectorMER    | 0.0776   | 0.2325     | 0.33    | 0.738    |
| sectorMIN    | 0.2399   | 0.1884     | 1.27    | 0.203    |
| sectorTRN    | 0.1013   | 0.2475     | 0.41    | 0.682    |
| sectorWOD    | 0.3908   | 0.2325     | 1.68    | 0.093    |

(Dispersion parameter for Negative Binomial(1.639) family taken to be 1)

Null deviance: 521.58 on 247 degrees of freedom  
Residual deviance: 296.52 on 234 degrees of freedom  
AIC: 1675

Number of Fisher Scoring iterations: 1

Theta: 1.639  
Std. Err.: 0.192

2 x log-likelihood: -1645.257

The estimate is very close to the value 1.5 that we picked by grid search. We used the standard R summary () function here in preference to S () in the **car** package because summary () shows the estimated dispersion parameter more clearly in the output.

## 6.11 Arguments to glm ()

The glm () function in R takes the following arguments:

```

args ("glm")

function (formula, family = gaussian, data, weights, subset,
  na.action, start = NULL, etastart, mustart, offset,
  control = glm.control(...), model = TRUE,
  method = "glm.fit", x = FALSE, y = TRUE, contrasts = NULL,
  ...)

```

We have already discussed in some detail the use of the formula and family arguments. The data, subset, na.action, and contrasts arguments work as in lm () (see [Section 4.9.1](#)).

We comment briefly on the other arguments to glm ():

### 6.11.1 weights

The weights argument is used to specify *prior weights*, which are a vector of  $n$  positive numbers. Leaving off this argument effectively sets all the prior weights to 1. For Gaussian GLMs, the weights argument is used for weighted-least-squares regression, and this argument can serve a similar purpose for gamma GLMs. For a Poisson GLM, the weights have no obvious elementary use. As we mentioned (in [Section 6.4](#)), the weights argument may also be used to specify binomial denominators (i.e., number of successes plus number of failures) in a binomial GLM.

The IWLS computing algorithm used by glm () ([Section 6.12](#)) produces a set of *working weights*, which are different from the prior weights. The weights () function applied to a fitted "glm" object retrieves the *prior* weights by default; to get the working weights, specify the argument type="working".

### 6.11.2 start, etastart, mustart

These arguments supply start values respectively for the coefficients in the linear predictor, the linear predictor itself, and the vector of means. One of the remarkable features of GLMs is that the automatic algorithm that is used to find starting values is almost always effective, and the user therefore need not provide starting values except in very rare circumstances.

### 6.11.3 offset

An offset is used to allow for a predictor in a GLM that has a fixed coefficient of 1. The most common use for offsets is in Poisson models. Suppose, for example, that for a single individual with regressors  $\mathbf{x}$ , the number of visits made to the doctor in a fixed time period has a Poisson distribution with mean  $\mu(\mathbf{x})$ . We don't observe each individual but rather observe the total number of doctor visits  $y$  among the  $N$  people who share the regressors  $\mathbf{x}$ . From a basic property of the Poisson distribution,  $y|N, \mathbf{x}$  has a Poisson distribution with mean  $N \times \mu(\mathbf{x})$ . If

$$\mu(\mathbf{x}) = N \exp[\eta(\mathbf{x})]$$

we fit a model with the log link,  $\log[\mu(\mathbf{x})] = \eta(\mathbf{x}) + \log N$

and we therefore want to fit a Poisson regression with the linear predictor  $\eta(\mathbf{x}) + \log N$ . This is accomplished by setting the argument `offset=log(N)`.<sup>33</sup>

<sup>33</sup> Alternatively, we can add the term `offset(N)` to the right-hand side of the model formula in a call to `glm()`. Functions in our **effects** package can handle an offset specified via the `offset` argument to `glm()` but not an offset specified by calling the `offset()` function.

Similar situations arise more generally when the response  $y$  is a count (e.g., number of children born in a geographic area in, say, a year) and  $N$  is a measure of size (e.g., number of women of childbearing age in the area). Then using  $\log N$  as an offset in a count regression model with the log link effectively produces a regression model for the *rate*  $y/N$  (e.g., the general fertility rate—children born divided by number of women).

### 6.11.4 control

This argument allows the user to set several technical options, in the form of a list, controlling the IWLS fitting algorithm (described in the next section): `epsilon`, the convergence criterion (which defaults to 0.0001), representing the maximum relative change in the deviance before a solution is declared and iteration stops; `maxit`, the maximum number of iterations (default, 10); and `trace` (default, FALSE), which, if TRUE, causes a record of the IWLS iterations to be printed. These control options can also be specified directly as arguments to `glm`

- 0. The ability to control the IWLS fitting process is useful when convergence problems are encountered.

### 6.11.5 model, method, x, y

As for linear models, these are technical options.

## 6.12 Fitting GLMs by Iterated Weighted Least Squares\*

Maximum-likelihood estimates for generalized linear models in R are obtained by *iterated weighted least squares (IWLS)*, also called *iteratively reweighted least squares (IRLS)*. It occasionally helps to know some of the details.

IWLS proceeds by forming a quadratic local approximation to the log-likelihood function; maximizing this approximate log-likelihood is a linear weighted-least-squares problem. Suppose that the vector  $\beta^{(t)}$  contains the current estimates of the regression parameters of the GLM at iteration  $t$ . Using the subscript  $i$  for the  $i$ th observation, we calculate the current values of the linear predictor, , where is the  $i$ th row of the model matrix  $\mathbf{X}$  of the regressors; the fitted values, ; the variance function, ; the *working response*,<sup>34</sup>

$$z_i^{(t)} = \eta_i^{(t)} + (\gamma_i - \mu_i^{(t)}) \left( \frac{\partial \eta_i}{\partial \mu_i} \right)^{(t)}$$

$$z_i^{(t)} - \eta_i^{(t)} = (\gamma_i - \mu_i^{(t)}) \left( \frac{\partial \eta_i}{\partial \mu_i} \right)^{(t)}$$

<sup>34</sup> The values

are called *working residuals* and play a role in diagnostics for GLMs (see [Section 8.6](#)).

$$w_i^{(t)} = \frac{1}{c_i v_i^{(t)} \left[ \left( \frac{\partial \eta_i}{\partial \mu_i} \right)^{(t)} \right]^2}$$

and the *working weights*,

where the  $c_i$  are fixed constants that depend on the distributional family of the GLM (e.g., for the binomial family, ). Then we perform a weighted-least-squares regression of  $z^{(t)}$  on the xs in the linear predictor, minimizing the weighted sum of squares , obtaining new estimates of the regression parameters,  $\beta^{(t+1)}$ . This process is initiated with suitable starting values,  $\beta^{(0)}$ , and continues until the coefficients stabilize at the maximum-likelihood estimates, .

The estimated asymptotic covariance matrix of is obtained from the last iteration of the IWLS procedure, as  $\widehat{\mathcal{V}}(\widehat{\beta}) = \widehat{\phi}(\mathbf{X}'\mathbf{W}\mathbf{X})^{-1}$

where  $\mathbf{W} = \text{diag } \{w^i\}$ , and (if  $\phi$  is to be estimated from the data) is the Pearson statistic divided by the residual *df* for the model.

Binomial logistic regression provides a relatively simple illustration; we have (after algebraic manipulation):

$$\begin{aligned} \mu_i^{(t)} &= [1 + \exp(-\eta_i^{(t)})]^{-1} \\ v_i^{(t)} &= \mu_i^{(t)}(1 - \mu_i^{(t)}) \\ \left( \frac{\partial \eta_i}{\partial \mu_i} \right)^{(t)} &= \frac{1}{\mu_i^{(t)}(1 - \mu_i^{(t)})} \\ z_i^{(t)} &= \eta_i^{(t)} + (\gamma_i - \mu_i^{(t)})/v_i^{(t)} \\ w_i^{(t)} &= N_i v_i \end{aligned}$$

and  $\phi$  is set to 1.

## 6.13 Complementary Reading and References

- The structure of GLMs, introduced in [Section 6.1](#), is amplified in Fox (2016, Section 15.1) and Weisberg (2014, Section 12.4). The canonical reference for generalized linear models is McCullagh and Nelder (1989).
- Logistic regression, covered in [Sections 6.3–6.4](#), is discussed in Fox (2016, chap. 14) and Weisberg (2014, Sections 12.1–12.3).
- Poisson regression models, and the related loglinear models for contingency tables, are discussed in Fox (2016, Section 15.2). More complete treatments can be found in books by Agresti (2007), Simonoff (2003), and Fienberg (1980), among others.
- The material in [Sections 6.7–6.9](#) on models for polytomous responses is discussed in Fox (2016, Section 14.2).
- Many of these topics are also covered in Long (1997) and in Powers and Xie (2008).

# 7 Fitting Mixed-Effects Models

John Fox and Sanford Weisberg

The linear and generalized linear models of [Chapters 4](#) and [6](#) assume independently sampled observations. In contrast, *mixed-effects models* (or *mixed models*, for short), the subject of the current chapter, are models for dependent data, with observations clustered into groups. Common applications in the social sciences are to *hierarchical data*, where individuals are clustered in higher-level units (such as students within schools), and to *longitudinal data*, where repeated observations are taken on the same individuals (as in a panel survey).

We discuss two widely used R packages in the chapter: **nlme** (Pinheiro & Bates, 2000; Pinheiro, Bates, DebRoy, Sarkar, & R Core Team, 2018), which is part of the standard R distribution, and **lme4** (Bates, Mächler, Bolker, & Walker, 2015), which is available on CRAN.<sup>1</sup> We take up both linear mixed-effects models for conditionally normally distributed responses, with illustrative applications to hierarchical and longitudinal data, and generalized linear mixed-effects models, where the conditional distribution of the response need not be normal.

[1](#) **nlme** stands for **n**onlinear **l**inear **m**ixed **e**ffects, even though the package also includes the **lme** () function for fitting *linear* mixed models. Similarly, **lme4** stands for **l**inear **m**ixed **e**ffects with **S4** classes but also includes functions for fitting *generalized linear* and *nonlinear* mixed models.

- [Section 7.1](#) revisits the linear model, to provide a point of departure for linear mixed-effects models (LMMs), which are described in [Section 7.2](#).
- [Section 7.3](#) introduces generalized linear mixed-effects models (GLMMs).

Mixed models, and their implementation in R, are a broad and complex subject. This chapter merely scratches the surface, introducing the mixed models most frequently used in social science applications and that have strong analogies with the linear and generalized linear models discussed previously.<sup>2</sup>

[2](#) Mixed-effects models are also described in the online appendices to the *R Companion* on Bayesian methods and on nonlinear regression.

## 7.1 Background: The Linear Model Revisited

For comparison with the mixed-effects models discussed in this chapter, we rewrite the standard linear model from [Chapter 4](#) using slightly different but equivalent notation:

$$y_i = \beta_1 x_{1i} + \beta_2 x_{2i} + \cdots + \beta_p x_{pi} + \varepsilon_i$$

$$\varepsilon_i \sim \text{NID}(0, \sigma^2)$$

As before,  $y_i$  is the response;  $x_{1i}, \dots, x_{pi}$  are regressors;  $\beta_1, \dots, \beta_p$ , the population

regression coefficients, are fixed and generally unknown parameters; and  $i = 1, \dots, n$  indexes  $n$  cases. We generally take  $x_{1i} = 1$ , to accommodate an intercept, and interpret the other regressors as fixed or conditionally fixed values. The  $\varepsilon_i$  are specified as independent, normally distributed random errors with zero means and common unknown error variance  $\sigma^2$ . We can think of  $\varepsilon$  as providing a *random effect*, particular to each case, because without it the response  $y$  would be completely determined by the xs. Part of the variation of  $y$  is due to the error variance  $\sigma^2$ , which may be thought of as a *variance component*. Although the assumption of normally distributed errors is stronger than needed for the linear model, distributional assumptions are essential for the mixed-effects models discussed in this chapter.

### 7.1.1 The Linear Model in Matrix Form\*

Some readers will find the matrix formulation of the linear model easier to follow:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \varepsilon$$

$$\varepsilon \sim \mathbf{N}_n(0, \sigma^2 \mathbf{I}_n)$$

where  $\mathbf{y} = (y_1, y_2, \dots, y_n)'$  is the response vector;  $\mathbf{X}$  is the fixed model matrix, with typical row  $\beta = (\beta_1, \beta_2, \dots, \beta_p)'$  is the vector of population regression coefficients;  $\varepsilon = (\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n)'$  is the vector of errors;  $\mathbf{N}_n$  represents the  $n$ -variable multivariate normal distribution; 0 is an  $n \times 1$  vector of zeros; and  $\mathbf{I}_n$  is the order-  $n$  identity matrix.

## 7.2 Linear Mixed-Effects Models

The data for mixed-effects models are more complicated than for the linear model because cases are divided into groups or *clusters*. Suppose that there are  $M$  clusters, each with one or more cases. Cases in the same group are generally not independent; for example, test scores for students in the same school are likely to be correlated because the students in a school share teachers, facilities, and other factors common to the school. Cases in different groups are assumed to be independent, because, for example, students in different schools have different teachers and other properties associated with schools. We use the subscript  $i$ ,  $i = 1, \dots, M$ , to index clusters, and the subscript  $j = 1, \dots, n_i$  to index the observations within a cluster. For hierarchical data, clusters represent higher-level units (such as schools), and observations represent individuals (students within a school).<sup>3</sup> For longitudinal data, clusters typically represent individuals (such as respondents to a panel study), and observations represent repeated

measurements taken on different occasions (such as responses to different waves of a panel study); the occasions need not be the same for different individuals.

[3](#) There may be more than two levels of hierarchy, such as students within classrooms within schools, and the organization of the data need not be strictly nested, as, for example, when students in a high school have multiple teachers. These more complex data patterns can also be handled by mixed-effects models, at the expense of more complicated notation. We consider only two-level, nested data in this chapter.

Following Laird and Ware (1982), the *linear mixed-effects model (LMM)* posits the following structure for the data:

(7.1)

$$y_{ij} = \beta_1 x_{1ij} + \dots + \beta_p x_{pij} + b_{i1} z_{1ij} + \dots + b_{iq} z_{qij} + \varepsilon_{ij}$$

$$b_{ik} \sim N(0, \psi_k^2), \text{Cov}(b_{ik}, b_{ik'}) = \psi_{kk'}$$

$$\varepsilon_{ij} \sim N(0, \sigma^2 \lambda_{ijj}), \text{Cov}(\varepsilon_{ij}, \varepsilon_{ij'}) = \sigma^2 \lambda_{ijj'}$$

where

- $y_{ij}$  is the value of the response variable for the  $j$ th of  $n_i$  cases (e.g., students) in the  $i$ th of  $M$  groups or clusters (e.g., schools). Group sizes, the  $n_i$ , can vary from group to group.
- $\beta_1, \dots, \beta_p$  are the *fixed-effect coefficients*. The fixed-effect coefficients are assumed to be the same for all clusters and so are not indexed by  $i$ .
- $x_{1ij}, \dots, x_{pij}$  are the fixed-effect regressors for observation  $j$  in group  $i$ . The values of the regressors are treated as fixed.
- $b_{i1}, \dots, b_{iq}$  are the *random effects* for group  $i$ . The  $b_{ik}$ ,  $k = 1, \dots, q$ , are random variables, not parameters, and are similar in this respect to the errors  $\varepsilon_{ij}$ .
- Although the random effects differ by groups, they are assumed to be sampled from a common, multivariately normally distributed population, with zero means, variances, and covariances. The variances and covariances are unknown parameters, called *variance and covariance components*, to be estimated from the data along with the fixed-effects coefficients. In some applications, the  $\psi$ s may be parameterized in terms of a relatively small number of fundamental parameters.
- $z_{1ij}, \dots, z_{qij}$  are the random-effect regressors. In the examples in this chapter, and commonly in applications, the  $z$ s are a subset of the  $x$ s, but this is not necessarily the case.

- $\varepsilon_{ij}$  is the error for observation  $j$  in group  $i$ .
- The errors for group  $i$  are assumed to have a multivariate normal distribution with zero means;  $\sigma^2 \lambda_{ijj'}$  is the covariance between errors  $\varepsilon_{ij}$  and  $\varepsilon_{ij'}$  in group  $i$ , and so unlike in the linear model, we may allow the  $\varepsilon$ s to have different variances and to be correlated. In most of the applications in the chapter, however,  $\lambda_{ijj} = 1$  and  $\lambda_{ijj'} = 0$ , corresponding to uncorrelated errors with equal variances, but we consider correlated errors in [Section 7.2.5](#). Because of their large number, the  $\lambda_{ijj'}$  are generally parametrized in terms of a few basic parameters, and their specific form depends upon context.
- This structure for the random effects and errors is purely *within-group*, so random effects and errors in different groups are uncorrelated.

## 7.2.1 Matrix Form of the Linear Mixed-Effects Model\*

The matrix form of the LMM is equivalent to Equation 7.1 but considerably simpler to write down. For group  $i$ ,  $i = 1, \dots, M$ ,

$$(7.2) \quad \begin{aligned} \mathbf{y}_i &= \mathbf{X}_i \boldsymbol{\beta} + \mathbf{Z}_i \mathbf{b}_i + \boldsymbol{\varepsilon}_i \\ \mathbf{b}_i &\sim \mathbf{N}_q(0, \boldsymbol{\Psi}) \\ \boldsymbol{\varepsilon}_i &\sim \mathbf{N}_{n_i}(0, \sigma^2 \boldsymbol{\Lambda}_i) \end{aligned}$$

where

- $\mathbf{y}_i$  is the  $n_i \times 1$  response vector for observations in the  $i$ th group. The  $n_i$  need not all be equal and usually are not equal.
- $\mathbf{X}_i$  is the fixed  $n_i \times p$  model matrix of fixed-effect regressors for cases in group  $i$ .
- $\boldsymbol{\beta}$  is the  $p \times 1$  vector of fixed-effect coefficients, which are the same in every group.
- $\mathbf{Z}_i$  is the  $n_i \times q$  fixed model matrix of regressors for the random effects for observations in group  $i$ .
- $\mathbf{b}_i$  is the  $q \times 1$  vector of random effects for group  $i$ .
- $\boldsymbol{\varepsilon}_i$  is the  $n_i \times 1$  vector of random errors for cases in group  $i$ .
- $\boldsymbol{\Psi}$  is the  $q \times q$  covariance matrix for the random effects. To conform to our previous notation, the diagonal elements are (the random-effect variances)

and the off-diagonal elements are  $\psi_{jj'}$ .

- $\sigma^2 \boldsymbol{\Lambda}_i$  is the  $n_i \times n_i$  covariance matrix for the errors in group  $i$ , often with .

Using Equation 7.2, it is instructive to compute the covariance matrix of  $\mathbf{y}_i$  in terms of the variances and covariances of the random effects and errors (i.e., the variance and covariance components):

$$\begin{aligned}\text{Var}(\mathbf{y}_i) &= \text{Var}(\mathbf{Z}_i \mathbf{b}_i) + \text{Var}(\boldsymbol{\varepsilon}_i) \\ &= \mathbf{Z}_i \boldsymbol{\Psi} \mathbf{Z}'_i + \sigma^2 \boldsymbol{\Lambda}_i\end{aligned}$$

Thus, the mixed-effects model implies that the observations within a cluster generally have correlated responses and that their correlations depend on all the variance and covariance components and on the random-effect regressors  $\mathbf{Z}$ . The simplest case of a mixed-effects linear model has only one random effect, a random group-specific intercept. In this case,  $q = 1$ ,  $\mathbf{b}_i = b_{1i}$ , and the corresponding  $\mathbf{Z}_i = 1$ , a column of ones. If we further assume uncorrelated within-group errors with constant variances, (where  $I$  is the identity matrix), we

get  $\text{Var}(\mathbf{y}_i) = \psi_1^2 11' + \sigma^2 I_{n_i}$

Thus, each element of  $\mathbf{y}_i$  has the same variance, ; each pair of observations in the same cluster have the same covariance, ; and the correlation between any two observations in the same cluster is constant, .

## 7.2.2 An Application to Hierarchical Data

Applications of mixed models to hierarchical data have become common in the social sciences, and nowhere more so than in research on education. The following example is borrowed from Raudenbush and Bryk's influential text on hierarchical linear models (Raudenbush & Bryk, 2002) and also appears in a paper by Singer (1998), who demonstrates how such models can be fit by the MIXED procedure in SAS. In this section, we will show how to model Raudenbush and Bryk's data using the `lme()` function in the **nlme** package and the `lmer()` function in the **lme4** package.

The data for the example, from the 1982 "High School and Beyond" (HSB) survey, are for 7,185 high school students from 160 schools. There are on average  $7,185/160 \approx 45$  students per school. We have two levels of hierarchy,

with schools at the higher or group level and students within schools as the lower or individual level.<sup>4</sup> The data are available in the data sets `MathAchieve` and `MathAchSchool` in the **nlme** package:<sup>5</sup> The first data set pertains to students within schools, with one row for each of the 7,185 students:

<sup>4</sup> We deliberately avoid the terminology “Level 1” and “Level 2” because its use isn’t standardized: In the social sciences, “Level 1” typically references the lower, or individual, level, and “Level 2” the higher, or group, level. In the documentation for the **nlme** package, this terminology is reversed.

<sup>5</sup> Some data sets in the **nlme** package, such as `MathAchieve`, are *grouped-data objects*, which behave like data frames but include extra information that can be used by functions in **nlme**. The extra features of grouped-data objects are not used by the **lme4** package. In this chapter, we simply treat grouped-data objects as data frames.

```
library("car")
library("nlme")
brief(MathAchieve)
```

|   |   | 7185 x 6 nfnGroupedData (7180 rows omitted) | School | Minority | Sex    | SES    | MathAch | MEANSES |
|---|---|---|--------|----------|--------|--------|---------|---------|
|   |   |   | [c]    | [f]      | [f]    | [n]    | [n]     | [n]     |
| 1 | 2 | 1224  | 1224   | No       | Female | -1.528 | 5.876   | -0.428  |
| 2 | 3 | 1224  | 1224   | No       | Female | -0.588 | 19.708  | -0.428  |
| 3 | 4 | 1224  | 1224   | No       | Male   | -0.528 | 20.349  | -0.428  |
| 4 | 5 | ...   | ...    |          |        |        |         |         |
| 5 | 6 | 7184  | 9586   | No       | Female | -0.008 | 16.241  | 0.627   |
| 6 | 7 | 7185  | 9586   | No       | Female | 0.792  | 22.733  | 0.627   |

The first three rows are all for students in School 1224 and the last two for students in School 9586. Although the students in the data set are ordered by schools, ordering cases by groups isn’t required by functions in the **nlme** and **lme4** packages.<sup>6</sup>

[6](#) For longitudinal data, we generally want the repeated observations for each individual to be in temporal order, but it's not necessary that the corresponding rows of the data set be adjacent.

The second data set pertains to the schools into which students are clustered, and there is one row for each of the  $M = 160$  schools:

***brief(MathAchSchool)***

| 160 x 7 data.frame (155 rows omitted) |        |      |          |        |         |         |          |
|---------------------------------------|--------|------|----------|--------|---------|---------|----------|
|                                       | School | Size | Sector   | PRACAD | DISCLIM | HIMINTY | MEANSSES |
|                                       | [f]    | [n]  | [f]      | [n]    | [n]     | [f]     | [n]      |
| 1224                                  | 1224   | 842  | Public   | 0.35   | 1.597   | 0       | -0.428   |
| 1288                                  | 1288   | 1855 | Public   | 0.27   | 0.174   | 0       | 0.128    |
| 1296                                  | 1296   | 1719 | Public   | 0.32   | -0.137  | 1       | -0.420   |
| . . .                                 |        |      |          |        |         |         |          |
| 9550                                  | 9550   | 1532 | Public   | 0.45   | 0.791   | 0       | 0.059    |
| 9586                                  | 9586   | 262  | Catholic | 1.00   | -2.416  | 0       | 0.627    |

In the analysis that follows, we use the following variables:

- School: an identification number for the student's school that appears in both the MathAchieve and MathAchSchool data sets. Students are clustered or grouped within schools.
- SES: the socioeconomic status of each student's family. This is a student-level variable in the MathAchieve data frame, sometimes called an *inner* or *individual-level* variable. The SES scores are *centered* to have a mean of zero, within rounding error, for the 7185 students in the sample. Prior to analyzing the data, we will recenter SES to a mean of zero within each school.
- MathAch: the student's score on a math achievement test, another student-level variable, and the response variable in the models that we consider below.
- Sector: a factor coded "Catholic" or "Public". This is a school-level variable and hence is identical for all students in the same school. A variable of this kind is sometimes called an *outer variable* or a *contextual variable*. Because the Sector variable resides in the school data set, we need to merge the data sets ([Section 2.3.5](#)). Such data management tasks are common in

preparing data for mixed modeling.

- MEANSES: this variable, given in both data sets, is, according to the help pages for the data sets, the “mean SES for the school.” The values of the variable, however, do not correspond to the *sample* means we compute from the student data in the MathAchieve data frame.<sup>7</sup> We ignore MEANSES and recompute the average SES in each school. Mean school SES is a school-level variable produced by aggregating data on the students within each school. This kind of variable is sometimes called a *compositional variable*.

<sup>7</sup> The source of this discrepancy is unclear, but it’s possible that MEANSES was computed for *all* of the students in each school rather than from the *sample* of students in the MathAchSchool data set.

We combine the individual and student-level data in a sequence of steps.

- First we use the group\_by () and summarize () functions along with the pipe (%>%) operator, all from the dplyr package (discussed in [Section 2.3.5](#)), to recompute the mean SES in each school and add it, along with Sector, to the MathAchSchool data frame:

```
library("dplyr")
MathAchieve %>% group_by(School) %>%
  summarize(mean.ses = mean(SES)) -> Temp
Temp <- merge(MathAchSchool, Temp, by="School")
brief(Temp)

160 x 8 data.frame (155 rows and 2 columns omitted)
  School Size   Sector PRACAD . . . MEANSES  mean.ses
      [f]  [n]       [f]     [n]           [n]       [n]
  1    1224  842 Public    0.35      -0.428 -0.434383
  2    1288 1855 Public    0.27      0.128  0.121600
  3    1296 1719 Public    0.32      -0.420 -0.425500
  . . .
  159   9550 1532 Public    0.45      0.059  0.053034
  160   9586  262 Catholic   1.00      0.627  0.621153
```

The object Temp is an intermediate, temporary data set, initially with one

row for each school, a column with the School ID number, and a column for mean.ses, computed by the summarize () function. We use the standard R merge () function to combine Temp with the school-level data in MathAchSchool, putting the resulting merged data frame back in Temp.

- Next, we merge the school-level variables Sector and mean.ses from Temp with the individual-level variables SES and MathAch from MathAchieve to produce the data set HSB, converting the variable names in the data set to lowercase, as is standard in this book:

```
HSB <- merge(Temp[, c("School", "Sector", "mean.ses")],
               MathAchieve[, c("School", "SES", "MathAch")], by="School")
names(HSB) <- tolower(names(HSB))
```

The combined data set has one row for each student, repeating the school-level values in Temp as needed. We retain Temp because we'll make use of it later in this section.

- Finally, we calculate school-centered SES by subtracting school mean SES from each student's SES:

```
HSB$cses <- with(HSB, ses - mean.ses)
brief(HSB)
```

|      | school | sector   | mean.ses | ses    | mathach | cses      |
|------|--------|----------|----------|--------|---------|-----------|
|      | [f]    | [f]      | [n]      | [n]    | [n]     | [n]       |
| 1    | 1224   | Public   | -0.43438 | -1.528 | 5.876   | -1.093617 |
| 2    | 1224   | Public   | -0.43438 | -0.588 | 19.708  | -0.153617 |
| 3    | 1224   | Public   | -0.43438 | -0.528 | 20.349  | -0.093617 |
| ...  |        |          |          |        |         |           |
| 7184 | 9586   | Catholic | 0.62115  | -0.008 | 16.241  | -0.629153 |
| 7185 | 9586   | Catholic | 0.62115  | 0.792  | 22.733  | 0.170847  |

The first three students are in School 1224, and thus the contextual variable sector and compositional variable mean.ses, which are properties of the school, are the same for all of them. Similarly, the last two students are in School 9586 and so share values of sector and mean.ses. The variable cses is students' SES centered to a mean of zero within each school and

consequently is an individual-level variable, differing from student to student in the same school.

Following Raudenbush and Bush, we will ask whether students' math achievement is related to their socioeconomic status, whether this relationship varies systematically by sector, and whether the relationship varies randomly across schools within the same sector.

## Examining the Data

As usual, we start by examining the data. There are too many schools to look at each individually, so we select samples of 20 public and 20 Catholic schools, storing each sample in a data frame:

```
set.seed(12345) # for reproducibility
cat <- with(HSB,
            sample(unique(school[sector == "Catholic"]), 20))
Cat.20 <- HSB[is.element(HSB$school, cat), ]
dim(Cat.20)

[1] 1003      6

pub <- with(HSB,
            sample(unique(school[sector == "Public"]), 20))
Pub.20 <- HSB[is.element(HSB$school, pub), ]
dim(Pub.20)

[1] 854      6
```

Thus, Cat.20 contains the data for 20 randomly selected Catholic schools, with 1,003 students, and Pub.20 contains the data for 20 randomly selected public schools, with 854 students. We explicitly set the seed for R's random-number generator to an arbitrary number so that if you replicate our work, you'll get the same samples of schools.

We use the `xyplot()` function in the `lattice` package (see [Section 9.3.1](#)) to visualize the relationship between math achievement and school-centered SES in

the sampled schools:

```
library("lattice")
xyplot(mathach ~ cses | school, data=Cat.20, main="Catholic",
       panel=function(x, y) {
         panel.points(x, y)
         panel.lmline(x, y, lty=2, lwd=2, col="darkgray")
         panel.loess(x, y, span=1, lwd=2)
       }
)

xyplot(mathach ~ cses | school, data=Pub.20, main="Public",
       panel=function(x, y) {
         panel.points(x, y)
         panel.lmline(x, y, lty=2, lwd=2, col="darkgray")
         panel.loess(x, y, span=1, lwd=2)
       }
)
```

The `xyplot()` function draws a trellis display of scatterplots of math achievement against school-centered socioeconomic status, one scatterplot for each school, as specified by the formula `mathach ~ cses | school`. The school number appears in the strip label above each panel. We create one graph for Catholic schools ([Figure 7.1](#)) and another for public schools ([Figure 7.2](#)). The argument `main` to `xyplot()` supplies the title of the graph. Each panel also displays a least-squares regression line and a loess smooth for the points in the school. Given the modest number of students in each school, we define a customized panel function to set the span for the loess smoother to 1, calling the **lattice** functions `panel.points()`, `panel.lmline()`, and `panel.loess()` to add the points, OLS line, and loess line, respectively, to each panel.

Examining the scatterplots in [Figures 7.1](#) and [7.2](#), there is a weak positive relationship between math achievement and SES in most Catholic schools, although there is variation among schools. In some schools, the slope of the regression line is near zero or even negative. There is also a positive relationship between the two variables for all but one of the public schools, and here the average slope is larger. Considering the moderate number of students in each school, linear regressions appear to provide a reasonable summary of the within-

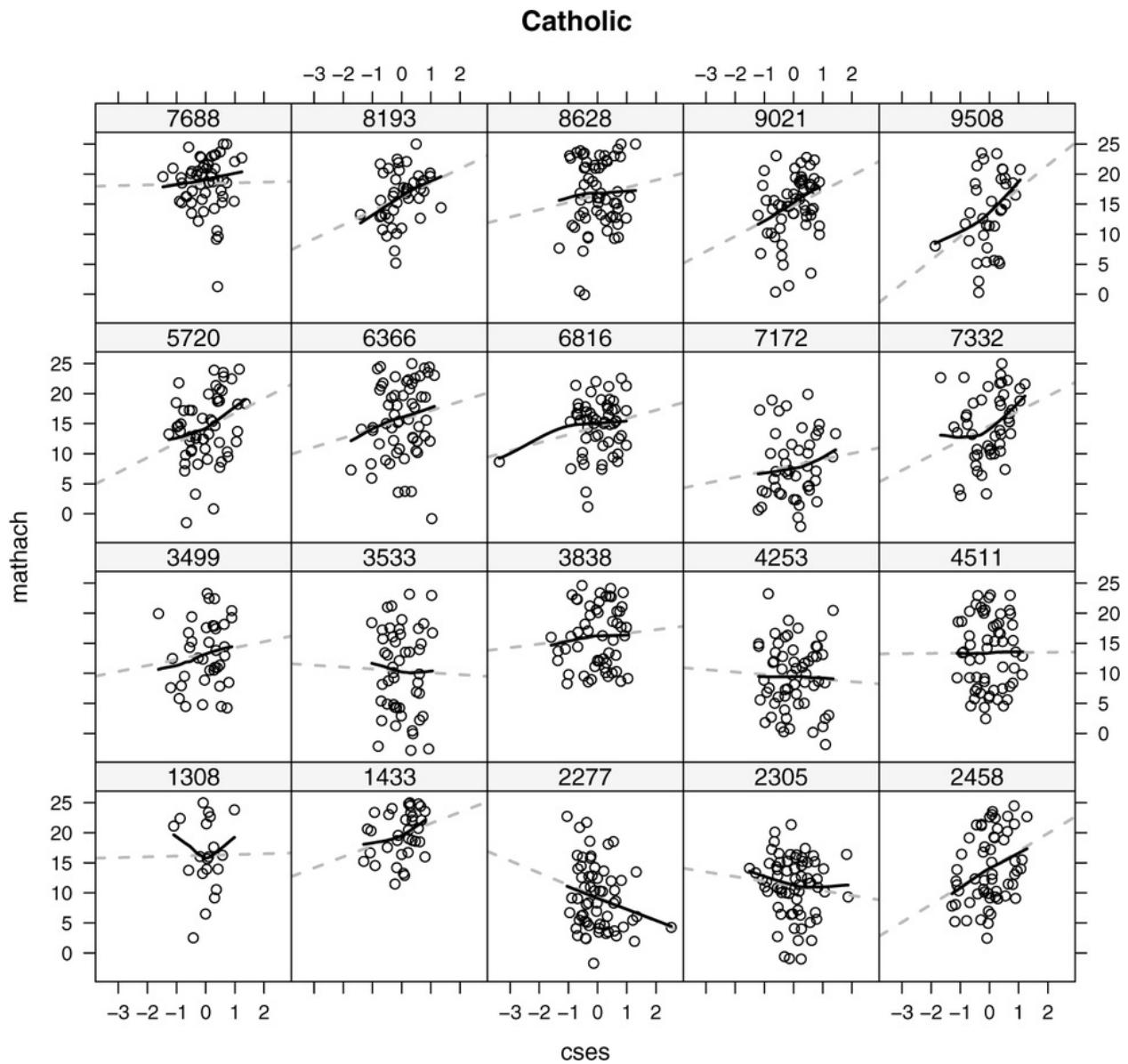
school relationships between math achievement and SES.

## Fitting Separate Regressions Within Schools

The **nlme** package includes the `lmList()` function for fitting a linear model to the cases in each group, returning a list of linear-model objects, which is itself an object of class "lmList".<sup>8</sup> Here, we fit the regression of math achievement scores on centered socioeconomic status for each school, creating separate "lmList" objects for Catholic and public schools:

```
cat.list <- lmList(mathach ~ cses | school,  
                    subset = sector == "Catholic", data=HSB)  
pub.list <- lmList(mathach ~ cses | school,  
                    subset = sector == "Public", data=HSB)
```

**Figure 7.1** Trellis display of math achievement by socioeconomic status for 20 randomly selected Catholic schools. The straight broken gray lines show linear least-squares fits and the solid black lines show loess smooths.



[8](#) A similar function is included in the **lme4** package.

Each of cat.list and pub.list consists of a *list* of linear-model objects; for example, using `brief()` to summarize the regression for the first public school:

```
brief(pub.list[[1]])
```

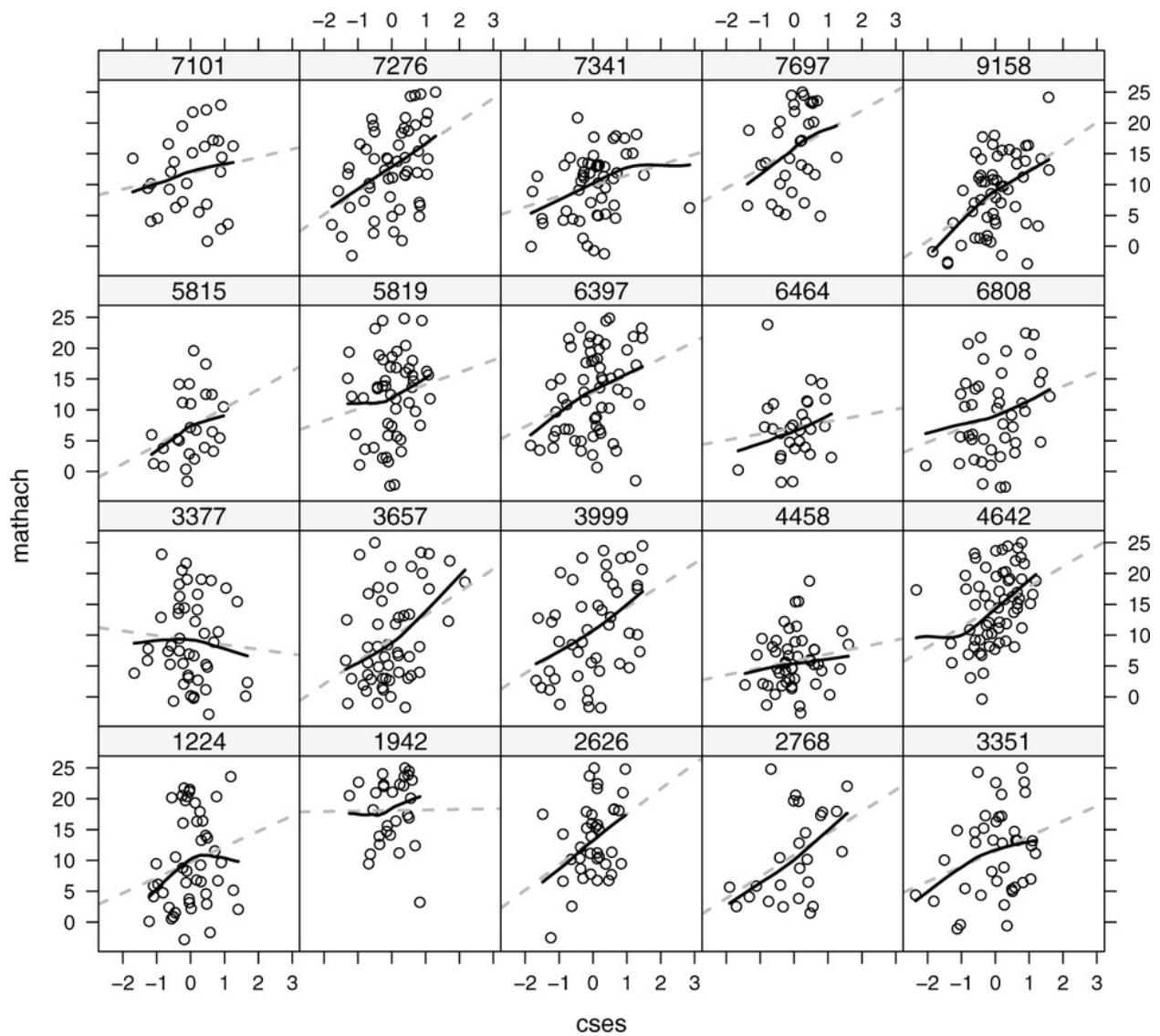
|          | (Intercept) | cses |
|----------|-------------|------|
| Estimate | 9.72        | 2.51 |

Std. Error                  1.10 1.77

Residual SD = 7.51 on 45 df, R-squared = 0.043

**Figure 7.2** Trellis display of math achievement by socioeconomic status for 20 randomly selected public schools.

### Public



```

cat.coef <- coef(cat.list)
head(cat.coef) # first 6 Catholic schools

  (Intercept)      cses
1308       16.256  0.12602
1317       13.178  1.27391
1433       19.719  1.85429
1436       18.112  1.60056
1462       10.496 -0.82881
1477       14.228  1.23061

pub.coef <- coef(pub.list)

```

We then examine all the slope and intercept estimates with parallel boxplots to compare the coefficients for Catholic and public schools, producing [Figure 7.3](#):

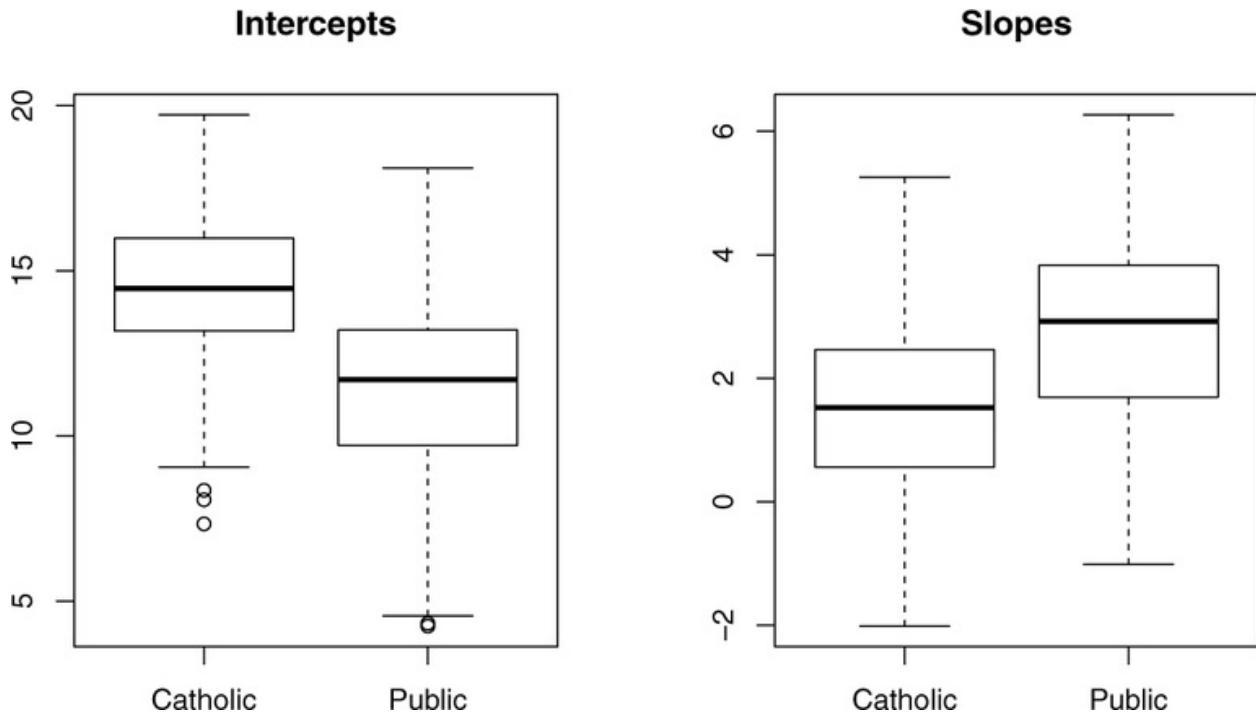
```

old <- par(mfrow=c(1, 2))
boxplot(cat.coef[, 1], pub.coef[, 1], main="Intercepts",
        names=c("Catholic", "Public"))
boxplot(cat.coef[, 2], pub.coef[, 2], main="Slopes",
        names=c("Catholic", "Public"))

par(old) # restore

```

**Figure 7.3** Boxplots of intercepts and slopes for the regressions of math achievement on centered SES in Catholic and public schools.



Setting the plotting parameter `mfrow` to one row and two columns produces the side-by-side pairs of boxplots in [Figure 7.3](#); at the end, we restore `mfrow` to its previous value.

It's apparent from [Figure 7.3](#) that Catholic schools on average have larger intercepts and smaller positive slopes than public school. Because student's SES is centered within schools, the intercepts estimate the average math achievement score in each school, and so average math achievement tends to be higher in Catholic than in public schools. That the average SES slope is lower in Catholic schools means that SES tends to make less of a difference to students' math achievement there than in public schools.

We next examine school-level effects by plotting the within-school slopes and intercepts against the schools' mean SES scores. To do so, we must first add school mean SES to the data sets of coefficients:

```

cat.coef <- merge(cat.coef, Temp[, c("School", "mean.ses")],
                    by.x="row.names", by.y="School")
pub.coef <- merge(pub.coef, Temp[, c("School", "mean.ses")],
                    by.x="row.names", by.y="School")
colnames(pub.coef)[c(1, 2)] <-
  colnames(cat.coef)[c(1, 2)] <- c("school", "intercept")
head(cat.coef)

  school intercept      cses mean.ses
1    1308     16.256  0.12602  0.52800
2    1317     13.178  1.27391  0.34533
3    1433     19.719  1.85429  0.71200
4    1436     18.112  1.60056  0.56291
5    1462     10.496 -0.82881 -0.66940
6    1477     14.228  1.23061  0.15958

```

Recall that the Temp data frame contains the school mean SES values that we computed previously. We use `merge()`, rather than `cbind()`, to add these values to the data frames of coefficients to ensure that rows match up properly by school ID, given in the row names of the coefficients and the variable School in Temp. We change the name of the intercept column to "intercept", which, unlike "(Intercept)", is a valid R name. Finally, we call the `scatterplot()` function from the `car` package to see how the coefficients are related to school mean SES:

```

scatterplot(intercept ~ mean.ses, data=cat.coef, boxplots=FALSE,
            main="Catholic")

scatterplot(cses ~ mean.ses, data=cat.coef, boxplots=FALSE,
            main="Catholic")

scatterplot(intercept ~ mean.ses, data=pub.coef, boxplots=FALSE,
            main="Public")

scatterplot(cses ~ mean.ses, data=pub.coef, boxplots=FALSE,
            main="Public")

```

The resulting scatterplots appear in [Figure 7.4](#).

The intercepts, representing the average levels of math achievement in the

schools, are linearly related to mean.ses in both the Catholic and the public schools, although the slope of the regression line is a bit steeper for public schools:

```
brief(lm(intercept ~ mean.ses, data=cat.coef))
```

|            | (Intercept) | mean.ses |
|------------|-------------|----------|
| Estimate   | 13.431      | 4.826    |
| Std. Error | 0.257       | 0.606    |

Residual SD = 1.99 on 68 df, R-squared = 0.483

```
brief(lm(intercept ~ mean.ses, data=pub.coef))
```

|            | (Intercept) | mean.ses |
|------------|-------------|----------|
| Estimate   | 12.183      | 5.852    |
| Std. Error | 0.195       | 0.483    |

Residual SD = 1.74 on 88 df, R-squared = 0.625

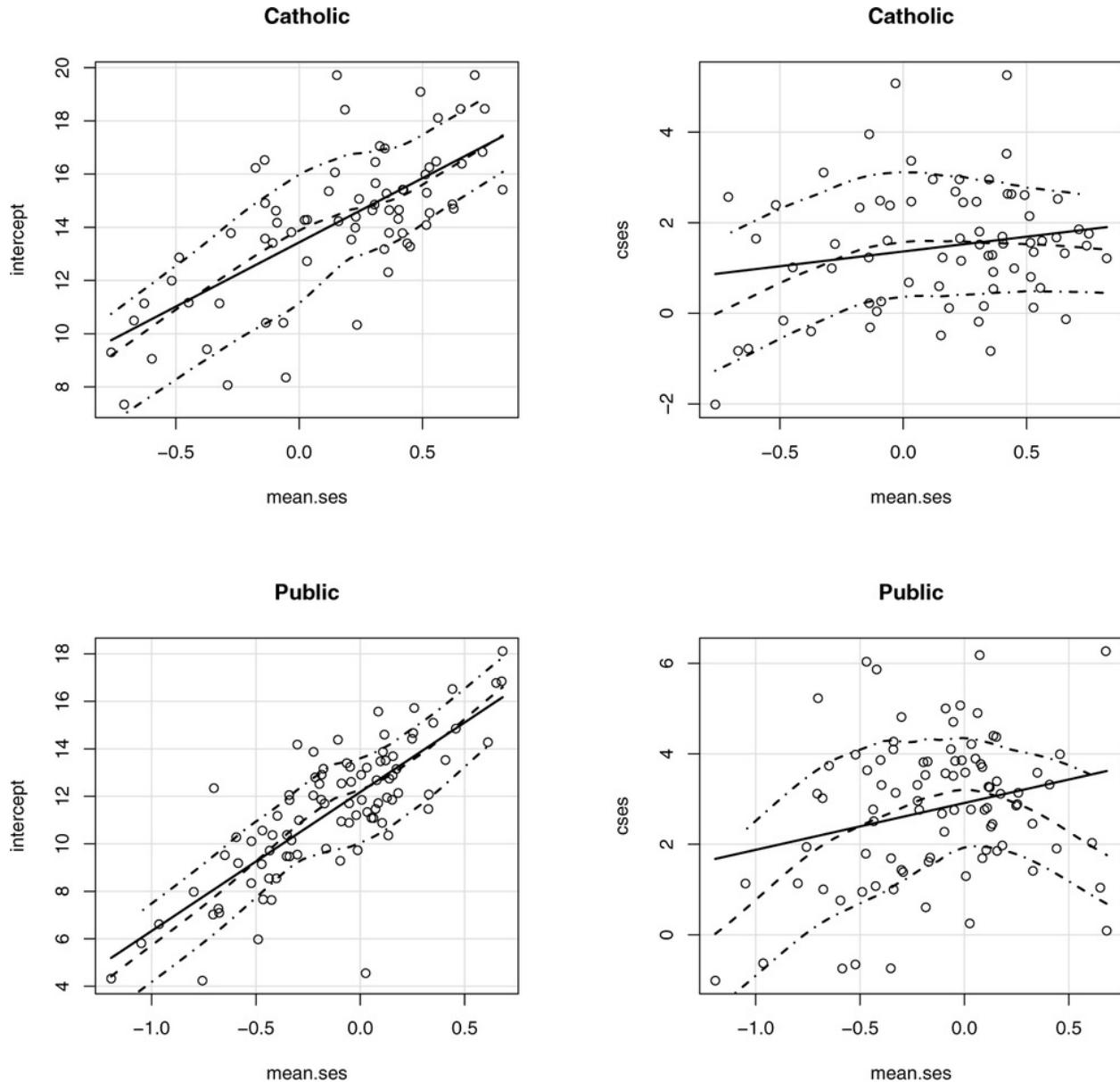
There appears to be a quadratic relationship between the cses slopes and mean.ses for both types of schools. In formulating a mixed model for the HSB data, we will follow Raudenbush and Bryk (2002) and Singer (1998) in ignoring this quadratic relationship. We invite the reader to modify the hierarchical model specified below to take the quadratic relationship into account (cf. Fox, 2016, chap. 23).

## A Hierarchical Linear Model for the High School and Beyond Data

From our initial examinations of the HSB data, we expect that the relationship between students' mathach and their cses, school-centered SES, can be reasonably summarized by a straight line for each school but that each school may require its own slope and intercept. In addition, the collections of slopes and intercepts may differ systematically between Catholic and public schools. The

approach of Raudenbush and Bryk (2002) and Singer (1998) to the HSB data is based on these observations.

**Figure 7.4** Scatterplots of within-school regression coefficients by mean school SES, separately for Catholic schools (top) and public schools (bottom).



- Within schools, we have the regression of math achievement on individual-level cses. Because cses is centered at zero in each school, the intercept is interpretable at the adjusted mean math achievement in a school, and the slope is interpreted as the average change in math achievement associated with a one-unit increase in SES. The individual-level regression equation for student  $j$  in school  $i$  is

$$(7.3) \quad \text{mathach}_{ij} = \alpha_{0i} + \alpha_{1i} \text{cses}_{ij} + \varepsilon_{ij}$$

an intercept and slope for each school and the common error variance. This analysis treats the  $\alpha$ s as fixed-effect regression coefficients.

- Second, at the school level, and also following Raudenbush and Bryk, as well as Singer, we entertain the possibility that the school intercepts and slopes depend upon sector and upon the average level of SES in the schools, now treating the  $\alpha$ s as random variables, each with an error component, denoted by  $u$ :

$$(7.4) \quad \begin{aligned} \alpha_{0i} &= \gamma_{00} + \gamma_{01}\text{mean.ses}_i + \gamma_{02}\text{sector}_i + u_{0i} \\ \alpha_{1i} &= \gamma_{10} + \gamma_{11}\text{mean.ses}_i + \gamma_{12}\text{sector}_i + u_{1i} \end{aligned}$$

This kind of formulation is sometimes called a *coefficients-as-outcomes* model.

Substituting the school-level Equations 7.4 into the individual-level Equation 7.3 produces

$$\begin{aligned} \mathbf{mathach}_{ij} &= \gamma_{00} + \gamma_{01}\text{mean.ses}_i + \gamma_{02}\text{sector}_i + u_{0i} \\ &\quad + (\gamma_{10} + \gamma_{11}\text{mean.ses}_i + \gamma_{12}\text{sector}_i + u_{1i})\mathbf{cses}_{ij} + \varepsilon_{ij} \end{aligned}$$

Then, multiplying out and rearranging terms,

$$\begin{aligned} \mathbf{mathach}_{ij} &= \gamma_{00} + \gamma_{01}\text{mean.ses}_i + \gamma_{02}\text{sector}_i + \gamma_{10}\mathbf{cses}_{ij} \\ &\quad + \gamma_{11}\text{mean.ses}_i\mathbf{cses}_{ij} + \gamma_{12}\text{sector}_i\mathbf{cses}_{ij} \\ &\quad + u_{0i} + u_{1i}\mathbf{cses}_{ij} + \varepsilon_{ij} \end{aligned}$$

Here, the  $\gamma$ s are fixed-effect coefficients, while the  $u$ s and the individual-level errors  $\varepsilon_{ij}$  are random effects. This specification greatly decreases the number of parameters that need to be estimated, as there are only six  $\gamma$ -parameters, plus two variances for the random  $u_{0i}$  and  $u_{1i}$ , one parameter for their covariance, and a variance for the  $\varepsilon$ s, or 10 parameters in total.

Finally, we rewrite the model in the Laird-Ware notation of the LMM (Equation 7.1):

$$(7.5)$$

$$\begin{aligned}
\text{mathach}_{ij} = & \beta_1 + \beta_2 \text{mean.ses}_i + \beta_3 \text{cses}_{ij} + \beta_4 \text{sector}_i \\
& + \beta_5 \text{mean.ses}_i \text{cses}_{ij} + \beta_6 \text{cses}_{ij} \text{sector}_i \\
& + b_{i1} + b_{i2} \text{cses}_{ij} + \varepsilon_{ij}
\end{aligned}$$

The change is purely notational, using  $\beta$ s for fixed effects and  $b$ s for random effects, and reordering the terms slightly to correspond to the order in which they will appear when we fit the model using the `lme()` and `lmer()` functions. We place no constraints on the covariance matrix of the random effects, so , and , and the individual-level errors are independent within schools, with constant variance, .

Even though the individual-level *errors* are assumed to be independent with constant variances, values of the *response variable* in the same school are heteroscedastic and correlated by virtue of their shared random effects:

$$\text{Var}(\text{mathach}_{ij}) = \sigma^2 + \psi_1^2 + \text{cses}_{ij}^2 \psi_2^2 + 2\text{cses}_{ij} \psi_{12}$$

$$\begin{aligned}
\text{Cov}(\text{mathach}_{ij}, \text{mathach}_{ij'}) = & \psi_1^2 + \text{cses}_{ij} \text{cses}_{ij'} \psi_2^2 \\
& + (\text{cses}_{ij} + \text{cses}_{ij'}) \psi_{12}
\end{aligned}$$

Cases in different groups are uncorrelated.

## Fitting the LMM With `lme()`

LMMs can be fit with the `lme()` function in the **nlme** package or with the `lmer()` function in the **Lme4** package. The two functions have different syntax, strengths, and limitations, but for the current example, either can be used. We illustrate `lme()` first. Specifying the fixed effects in the call to `lme()` is identical to specifying a linear model in a call to `lm()` (see [Chapter 4](#)). Random effects are specified via the `random` argument to `lme()`, which takes a one-sided R model formula.

Before fitting a mixed model to the HSB data, we reorder the levels of the factor `sector` so that the contrast for `sector` uses the values zero for public schools and 1 for Catholic schools, in conformity with the coding employed by Raudenbush and Bryk (2002) and by Singer (1998):<sup>9</sup>

```
HSB$sector <- factor(HSB$sector, levels=c ("Public", "Catholic"))
```

The LMM in Equation 7.5 is specified as follows:

```
hsb.lme.1 <- lme (mathach ~ mean.ses*cses + sector*cses, random = ~
cses | school, data=HSB)
```

[9](#) As in a linear model, reordering the levels of the factor changes the values of some of the fixed-effects coefficient estimates but does not change other aspects of the fitted model.

The formula for the random effects includes only the term for centered SES. As in a linear-model formula a random intercept is implied unless it is explicitly excluded by specifying -1 in the random formula. By default, lme () fits the model by *restricted maximum likelihood (REML)*, which in effect corrects the maximum-likelihood estimator for degrees of freedom (see the complementary readings in [Section 7.4](#)).

**S (hsb.lme.1)**

```
Linear mixed model fit by REML, Data: HSB
```

Fixed Effects:

```
Formula: mathach ~ mean.ses * cses + sector * cses
```

|             | Estimate | Std.Error | df   | t value | Pr(> t ) |
|-------------|----------|-----------|------|---------|----------|
| (Intercept) | 12.128   | 0.199     | 7022 | 60.85   | < 2e-16  |

|                    |        |       |      |       |         |
|--------------------|--------|-------|------|-------|---------|
| mean.ses           | 5.333  | 0.369 | 157  | 14.45 | < 2e-16 |
| cse                | 2.945  | 0.156 | 7022 | 18.93 | < 2e-16 |
| sectorCatholic     | 1.227  | 0.306 | 157  | 4.00  | 9.6e-05 |
| mean.ses:cse       | 1.039  | 0.299 | 7022 | 3.48  | 0.00051 |
| cse:sectorCatholic | -1.643 | 0.240 | 7022 | -6.85 | 7.9e-12 |

Random effects:

Formula: ~cse | school

Structure: General positive-definite, Log-Cholesky parametrization

StdDev Corr

|             |         |        |
|-------------|---------|--------|
| (Intercept) | 1.54265 | (Intr) |
| cse         | 0.31784 | 0.391  |
| Residual    | 6.05980 |        |

Number of Observations: 7185

Number of Groups: 160

| logLik | df | AIC   | BIC   |
|--------|----|-------|-------|
| -23252 | 10 | 46524 | 46592 |

The output from the S function for "lme" objects consists of several elements:

- The method of estimation (here, REML) and the data set to which the model is fit
- A table of fixed effects, similar to output from lm ()

To interpret the coefficients in this table, all of which are associated with very small  $p$ -values,<sup>10</sup> refer to the hierarchical form of the model given in Equations 7.3 and 7.4, and to the Laird-Ware form of the LMM in Equation 7.5.

<sup>10</sup> See [Section 7.2.3](#) for more careful hypothesis tests of fixed-effects coefficients in LMMs.

The fixed-effect intercept coefficient, , represents an estimate of the average level of math achievement in public schools (which is the baseline level of the dummy regressor for sector) adjusted for SES.

Likewise, the coefficient labeled `sectorCatholic`, , represents the difference between the average adjusted level of math achievement in Catholic schools and public schools.

The coefficient for `cse`, , is the estimated average slope for SES in public schools, while the coefficient labeled `cse:sectorCatholic`, , gives the difference in average slopes between Catholic and public schools. As we noted in our exploration of the data, the average level of math achievement is higher in Catholic than in public schools, and the average slope relating math achievement to students' SES is larger in public than in Catholic schools.

Given the parametrization of the model, the coefficient for `mean.ses`, , represents the slope of the regression of schools' average level of math achievement on their average level of SES.

The coefficient for the interaction `meanses:cse`, , gives the average change in the within-school SES slope associated with a one-unit increment in the school's mean SES.

- The panel in the output below the fixed effects displays estimates of the variance and covariance parameters for the random effects, in the form of standard deviations and correlations. The term labeled Residual is the estimate of  $\sigma$ . Thus, , , , and .
- The number of observations (students) and the number of groups (schools) appear next.
- The final panel shows the maximized (restricted) log-likelihood under the model, the degrees of freedom (number of parameters) for the model, and two measures of model fit, the *Akaike Information Criterion* (or *AIC*) and the *Bayesian Information Criterion* (or *BIC*), which are sometimes used for model selection (see, e.g., Weisberg, 2014, [Section 10.2.1](#) or Fox, 2016, Section 22.1.1).

The model is sufficiently simple, despite the interactions, to interpret the fixed effects from the estimated coefficients, but even here it is likely easier to visualize the model in effect plots. The **effects** package has methods for mixed models fit by functions in the **nlme** and **lme4** packages. We use the `predictorEffects()` function to produce a customized plot ([Figure 7.5](#)) of the

predictor effect of cses, which interacts in the model with both sector and mean.ses:

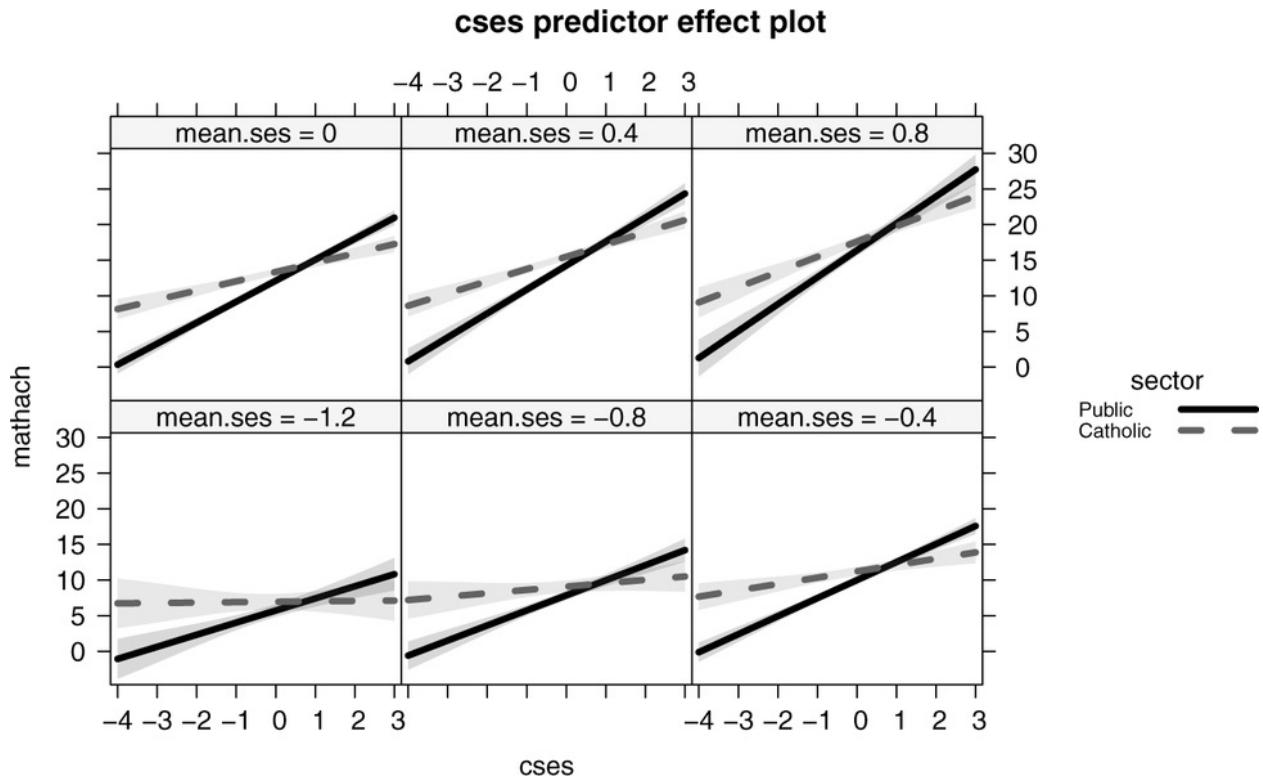
```
library("effects")
plot(predictorEffects(hsb.lme.1, ~ cses,
                      xlevels=list(mean.ses=
                                   round(seq(-1.2, 0.8, length=6), 1))),
      lines=list(multiline=TRUE, lwd=4),
      confint=list(style="bands"),
      axes=list(x=list(rug=FALSE)),
      lattice=list(key.args=list(space="right", columns=1)))
```

In constructing the graph, we use the xlevels argument to predictorEffects () to set mean.ses to six equally spaced values over its approximate range. We specify the lines argument to plot () to obtain a multiline graph, with lines for Catholic and public schools in each panel; the confint argument to include pointwise 95% confidence bands around the fitted lines; the axes argument to suppress the marginal rug-plot on the horizontal axes of the panels; and the lattice argument to position the key to the right of the graph. It is clear from the effect plot that the impact of students' SES on math achievement increases as the mean SES in their school rises and is greater at fixed levels of mean SES in public schools than in Catholic schools.

## Modeling Variance and Covariance Components

The model we fit to the HSB data assumes that each school has its own intercept  $b_{1i}$  and slope  $b_{2i}$  random effects sampled from a bivariate normal distribution with zero means, variances and , and covariance . Testing if these *variance and covariance components* are zero can be of interest in some problems. We can test hypotheses about the variances and covariances of random effects by deleting random-effects terms from the model. Tests are based on the change in the log of the maximized restricted likelihood, calculating log-likelihood-ratio statistics. When LMMs are fit by REML, we must be careful, however, to compare models that are identical in their fixed effects.

**Figure 7.5** Predictor effect display for centered SES in the LMM hsb.lme.1 fit to the High School and Beyond data.



We will first fit an alternative model that assumes that the variance component for the slopes is zero. This is equivalent to specifying a constant correlation between pairs of observations in the same group:

```
hsb.lme.2 <- update(hsb.lme.1,
  random = ~ 1 | school) # omitting random effect of cses
anova(hsb.lme.1, hsb.lme.2)
```

| Model     | df   | AIC   | BIC   | logLik | Test   | L.Ratio | p-value |
|-----------|------|-------|-------|--------|--------|---------|---------|
| hsb.lme.1 | 1 10 | 46524 | 46592 | -23252 |        |         |         |
| hsb.lme.2 | 2 8  | 46521 | 46576 | -23252 | 1 vs 2 | 1.1241  | 0.57    |

We can also fit a model that retains random slopes but includes no variance component for the intercepts, which implies that schools within sectors have the same average adjusted levels of math achievement but that the relationship between students' math achievement and their SES can vary from school to school:

```

hsb.lme.3 <- update(hsb.lme.1,
  random = ~ cses - 1 | school) # omitting random intercept
anova(hsb.lme.1, hsb.lme.3)

      Model df   AIC   BIC logLik   Test L.Ratio p-value
hsb.lme.1     1 10 46524 46592 -23252
hsb.lme.3     2  8 46740 46795 -23362 1 vs 2   220.56  <.0001

```

Each of these likelihood-ratio tests is on two degrees of freedom, because excluding one of the random effects removes not only its variance from the model but also its covariance with the other random effect. The large  $p$ -value in the first of these tests suggests no evidence against the constant-correlation model, or equivalently that , and thus that a random slope for each school is not required. In contrast, the small  $p$ -value for the second test suggests that , and even accounting for differences due to sector and mean school SES, average math achievement varies from school to school.

A more careful formulation of these tests takes account of the fact that each null hypothesis places a variance (but not the covariance) component on a boundary of the parameter space—that is, a variance of zero. Consequently, the null distribution of the likelihood-ratio test statistic is not simply chi-square with 2 degrees of freedom but rather a mixture of chi-square distributions.<sup>11</sup> Fortunately, it is reasonably simple to compute the corrected  $p$ -value:

```

pvalCorrected <- function(chisq, df) {
  (pchisq(chisq, df, lower.tail=FALSE) +
    pchisq(chisq, df - 1, lower.tail=FALSE)) / 2
}
pvalCorrected(1.1241, df=2)
[1] 0.42954
pvalCorrected(220.56, df=2)
[1] 6.7236e-49

```

<sup>11</sup> See the complementary readings in [Section 7.4](#) for discussion of this point.

Here, the corrected  $p$ -values are similar to the uncorrected ones.

Model hsb.lme.2, fit above, omits the random effects for cses; the resulting fixed-effects estimates and their standard errors are nearly identical to those for the initial model, hsb.lme.1, which includes these random effects:

```
compareCoefs(hsb.lme.1, hsb.lme.2)
```

Calls:

```
1: lme.formula(fixed = mathach ~ mean.ses * cses + sector *  
   cses, data = HSB, random = ~cses | school)  
2: lme.formula(fixed = mathach ~ mean.ses * cses + sector *  
   cses, data = HSB, random = ~1 | school)
```

|                     | Model 1        | Model 2        |
|---------------------|----------------|----------------|
| (Intercept)         | 12.128         | 12.128         |
| mean.ses            | 0.199<br>5.333 | 0.199<br>5.337 |
| SE                  | 0.369          | 0.369          |
|                     |                |                |
| cses                | 2.945          | 2.942          |
| SE                  | 0.156          | 0.151          |
|                     |                |                |
| sectorCatholic      | 1.227          | 1.225          |
| SE                  | 0.306          | 0.306          |
|                     |                |                |
| mean.ses:cses       | 1.039          | 1.044          |
| SE                  | 0.299          | 0.291          |
|                     |                |                |
| cses:sectorCatholic | -1.643         | -1.642         |
| SE                  | 0.240          | 0.233          |

Estimates of fixed effects are often insensitive to precise specification of the random effects, as long as the maximized likelihoods under the alternative

specifications are similar. Consequently, the conclusions based on model hsb.lme.1 from the effect plot in [Figure 7.5](#) apply equally to model hsb.lme.2.

## Fitting the LMM With lmer ()

We can perform the same analysis with lmer () in the **lme4** package. For example, to fit the initial hierarchical model considered in the previous section:

```
library("lme4")
hsb.lmer.1 <- lmer(mathach ~ mean.ses*cses + sector*cses
+ (cses | school), data=HSB)
S(hsb.lmer.1)

Linear mixed model fit by REML
Call: lmer(formula = mathach ~ mean.ses * cses + sector * cses
+ (cses | school), data = HSB)

Estimates of Fixed Effects:
                                         Estimate Std. Error z value Pr(>|z|)
(Intercept)                      12.128     0.199   60.86  < 2e-16
mean.ses                         5.333     0.369   14.45  < 2e-16
cses                            2.945     0.156   18.93  < 2e-16
sectorCatholic                   1.227     0.306    4.00  6.2e-05
mean.ses:cses                    1.039     0.299    3.48  0.00051
cses:sectorCatholic              -1.643     0.240   -6.85  7.3e-12

Estimates of Random Effects (Covariance Components):
Groups      Name          Std.Dev. Corr
school      (Intercept)  1.543
             cses         0.318    0.39
Residual               6.060

logLik       df    AIC    BIC
-23252      10  46524  46592
```

The estimates of the fixed effects and variance/covariance components are the same as those obtained from lme () (see page 350), but the specification of the

model is slightly different: Rather than using a random argument, as in `lme()`, the random effects in `lmer()` are given directly in the model formula, enclosed in parentheses.

In most instances, the choice between `lme()` and `lmer()` is unimportant, but there are some models that can be fit with one of these functions but not with the other. For example, `lme()` provides for modeling within-group correlation structures that specify the  $\Psi$  matrix using only a few parameters, a feature not available with `lmer()`. On the other hand, `lmer()` can accommodate *crossed* (as opposed to *nested*) random effects, while `lme()` cannot. For example, if we were interested in teacher effects on students' achievement, each student in a high school has several teachers, and so students would not be strictly nested within teachers.

A subtle difference between the `S()` output for models fit by `lme()` and `lmer()` is that the former includes *p*-values and degrees of freedom for Wald *t*-tests of the estimated coefficients, while the latter includes *p*-values for asymptotic Wald *z*-tests and no degrees of freedom. Both kinds of Wald tests can be inaccurate, an issue that we address in [Section 7.2.3](#).

As in the previous section, we can remove the random slopes from the model, comparing the resulting model to the initial model by a likelihood-ratio test:

```

hsb.lmer.2 <- lmer(mathach ~ mean.ses*cses + sector*cses
+ (1 | school), data=HSB) # omitting random effect of cses
anova(hsb.lmer.1, hsb.lmer.2)

refitting model(s) with ML (instead of REML)

Data: HSB
Models:
hsb.lmer.2: mathach ~ mean.ses * cses +
sector * cses + (1 | school)
hsb.lmer.1: mathach ~ mean.ses * cses +
sector * cses + (cses | school)
      Df   AIC   BIC logLik deviance Chisq Chi Df
hsb.lmer.2  8 46513 46568 -23249     46497
hsb.lmer.1 10 46516 46585 -23248     46496      1       2
Pr(>Chisq)
hsb.lmer.2
hsb.lmer.1      0.61

```

Out of an abundance of caution, `anova()` refits the models using ML rather than REML, because LR tests of models fit by REML that differ in their *fixed* effects are inappropriate. In our case, however, the models compared have identical fixed effects and differ only in the *random* effects. A likelihood-ratio test is appropriate even if the models are fit by REML. We can obtain this test by specifying the argument

`refit=FALSE:`

```

anova(hsb.lmer.1, hsb.lmer.2, refit=FALSE)

Data: HSB
Models:
hsb.lmer.2: mathach ~ mean.ses * cses +
  sector * cses + (1 | school)
hsb.lmer.1: mathach ~ mean.ses * cses +
  sector * cses + (cses | school)
      Df   AIC   BIC logLik deviance Chisq Chi Df
hsb.lmer.2  8 46521 46576 -23252      46505
hsb.lmer.1 10 46524 46592 -23252      46504  1.12      2
Pr(>Chisq)
hsb.lmer.2
hsb.lmer.1      0.57

```

The results are identical to those using lme ()�

### 7.2.3 Wald Tests for Linear Mixed-Effects Models

Likelihood-ratio-type tests for fixed effects in mixed models fit by REML are generally inappropriate, and a number of approximate methods for testing in this situation have been proposed. One approach to obtaining more accurate inferences in LMMs fit by REML is to adjust the estimated covariance matrix of the fixed effects to reduce the typically downward bias of the coefficient standard errors, as proposed by Kenward and Roger (1997), and to adjust degrees of freedom for Wald  $t$ - and  $F$ -tests, applying a method introduced by Satterthwaite (1946). These adjustments are available for linear mixed models fit by lmer () in the Anova () and linearHypothesis () functions in the **car** package, using infrastructure from the **pbkrtest** package (Halekoh & Højsgaard, 2014). For example:

```

> system.time(print(Anova(hsb.lmer.2, test="F")))
Analysis of Deviance Table
(Type II Wald F tests with Kenward-Roger df)

Response: mathach
          F Df Df.res   Pr(>F)
mean.ses    209.150  1  155.7 < 2.2e-16
cses        409.387  1 7023.1 < 2.2e-16
sector      15.999  1  154.3 9.803e-05
mean.ses:cses 12.878  1 7023.1 0.0003348
cses:sector  49.633  1 7023.1 2.030e-12

      user  system elapsed
749.87   15.82 770.59

```

We called the print () function explicitly in this command so that we could both time the computation and see the output produced by Anova ().

In this case, with many schools and a moderate number of students within each school, the KR tests are essentially the same as Wald chi-square tests using the naïvely computed covariance matrix for the fixed effects:

### **Anova (hsb.lmer.2)**

```

Analysis of Deviance Table (Type II Wald chisquare tests)

Response: mathach
          Chisq Df Pr(>Chisq)
mean.ses    209.2  1    < 2e-16
cses        409.4  1    < 2e-16
sector      16.0   1    6.3e-05
mean.ses:cses 12.9   1    0.00033
cses:sector  49.6   1    1.9e-12

```

Computing adjusted coefficient variances (and covariances) for this example is prohibitively time-consuming, taking more than 12 minutes!

## 7.2.4 Examining the Random Effects: Computing BLUPs

The model `hsb.lmer.2` includes a random coefficient for the intercept,  $b_{i1}$ , for each school. A natural question to ask in this problem is which schools have the largest intercepts, as these are the schools with the largest adjusted values of math achievement, and which schools have the smallest intercepts, as these are the schools that have the lowest adjusted math achievement. The linear mixed model for the HSB data is given by Equation 7.5 (on page 349), and so the intercept for the  $i$ th school is  $\beta_1 + b_{i1}$  for public schools and  $\beta_1 + \beta_4 + b_{i1}$  for Catholic schools. In either case,  $b_{i1}$  represents the adjusted difference between the mean math achievement for the  $i$ th school and the mean for that school's sector (because the student-level covariate `cse`s is centered to a mean of zero within each school and the school-level covariate `mean.ses` is centered to zero across the whole data set).

Estimates for the fixed-effects intercept  $\beta_1$  and for the sector coefficient  $\beta_4$  appear in the `S()` output, shown previously, and can also be retrieved directly from the fitted model by the command

```
fixef(hsb.lmer.2)
```

|                | (Intercept) | mean.ses     | cse                |
|----------------|-------------|--------------|--------------------|
|                | 12.1282     | 5.3367       | 2.9421             |
| sectorCatholic |             | mean.ses:cse | cse:sectorCatholic |
|                | 1.2245      | 1.0444       | -1.6422            |

The  $b_{i1}$  are random variables that are not directly *estimated* from the fitted model. Rather, **lme4** computes the mode of the conditional distribution of each  $b_{i1}$  given the fitted model as *predictions* of the  $b^{i1}$ . Here are the first six of them, computed for model `hsb.lmer.2`, which fits random intercepts but not random slopes:

```
school.effects <- ranef(hsb.lmer.2)$school
head(school.effects)
```

|      | (Intercept) |
|------|-------------|
| 1224 | -0.071165   |
| 1288 | 0.453116    |
| 1296 | -1.679816   |
| 1308 | 0.047918    |
| 1317 | -1.525924   |
| 1358 | -0.538952   |

Such “estimated” random effects (with “estimated” in quotes because the random effect coefficients are random variables, not parameters) are often called *best linear unbiased predictors* or *BLUPs*.

The ranef () function in the **lme4** package is complicated because it has to accommodate complex models with more than one level of hierarchy and with one or more random effects per level in the hierarchy. The command ranef (hsb.lmer.2)\$school returns the predicted random effects for schools as a matrix with one column. Recall that because the random intercepts have an expectation of zero, the predicted random effects are deviations from the fixed-effect intercept. To obtain the predicted intercept for each school, we add the BLUP for the intercept to the fixed-effect estimate of the intercept and, for public schools, the dummy-regressor fixed-effect coefficient for sector:

```

school.effects$sector <- MathAchSchool$Sector
school.effects$intercept <- school.effects$"Intercept" +
  ifelse(school.effects$sector == "Public",
         fixef(hsb.lmer.2)[1], sum(fixef(hsb.lmer.2)[c(1, 4)]))
head(school.effects)

  (Intercept) sector intercept
1224   -0.071165  Public    12.057
1288    0.453116  Public    12.581
1296   -1.679816  Public    10.448
1308    0.047918 Catholic   13.401
1317   -1.525924 Catholic   11.827
1358   -0.538952  Public    11.589

```

The intercept for each school computed in this manner represents average mathach in the school adjusted for school mean.ses (but not for sector). Here, then, are the lowest- and highest-performing schools:

```

school.effects <-
  school.effects[order(school.effects$intercept), ]
head(school.effects)

  (Intercept) sector intercept
8367     -3.6626  Public    8.4656
6990     -2.7383  Public    9.3899
8854     -2.5948  Public    9.5334
4523     -3.5463 Catholic   9.8064
3705     -3.1797 Catholic  10.1730
9397     -1.8720  Public   10.2562

tail(school.effects)

  (Intercept) sector intercept
2655      3.0528  Public   15.181
9198      2.0769 Catholic  15.430
8193      2.8104 Catholic  16.163

```

|      |        |          |        |
|------|--------|----------|--------|
| 8628 | 3.1280 | Catholic | 16.481 |
| 7688 | 3.1692 | Catholic | 16.522 |
| 3427 | 4.2148 | Catholic | 17.568 |

The first command sorts the schools in order of their intercepts. Thus, four of the six lowest performers are public schools, and five of the six highest performers are Catholic schools.

## 7.2.5 An Application to Longitudinal Data

*Longitudinal data* arise when the same units of observation—frequently subjects in an experiment or quasi-experiment, or respondents to a panel survey—are measured repeatedly on different occasions. Observations on the same subject are typically correlated, while observations on different subjects are usually taken to be independent. Mixed-effects models for longitudinal data are very similar to mixed models for hierarchical data, but with subjects treated as the grouping factor rather than groups of individuals.

For an illustration of longitudinal data, we draw on research described by Davis, Blackmore, Katzman, and Fox (2005) on the exercise histories of a patient group of 138 teenage girls hospitalized for eating disorders and of 93 comparable control subjects.<sup>12</sup> At the time of data collection, the girls and their parents were interviewed, and based on the interviews, an estimate of the average number of hours per week of exercise was constructed at each subject's current age, which varied from girl to girl, and in most cases at 2-year intervals in the past, starting at age 8. Thus, the response variable, exercise, was measured at several different ages for each girl, with most subjects measured four or five times and a few measured two or three times. We are interested in modeling change in exercise with age and particularly in examining the potential difference in typical exercise trajectories between patients and control subjects. The data for the study are in the data frame `Blackmore` (named after the researcher who collected the data) in the **carData** package:

<sup>12</sup> These data were generously made available to us by Elizabeth Blackmore and Caroline Davis of York University.

```
brief(Blackmore, rows=c(5, 5))
```

```
945 x 4 data.frame (935 rows omitted)
  subject    age exercise   group
      [f]    [n]      [n]     [f]
1       100  8.00      2.71 patient
2       100 10.00      1.94 patient
3       100 12.00      2.36 patient
4       100 14.00      1.54 patient
5       100 15.92      8.63 patient
. . .
768      286  8.00      1.10 control
769      286 10.00      1.10 control
770      286 12.00      0.35 control
771      286 14.00      0.40 control
772      286 17.00      0.29 control
```

The first column of the data set contains a subject identifier, and there are repeated rows for each subject, one at each age of observation. The observations are ordered by subject, and in the output above, the first five rows are for Subject 100, who is a patient, and the last five are for Subject 286, who is a control subject. Subject 100 was observed at ages 8, 10, 12, 14, and 15.92 years and Subject 286 at ages 8, 10, 12, 14, and 17; the last observation is at the age of hospitalization for the patient and the age at the interview for the control. Mixed-modeling functions in both the **nlme** and **lme4** packages expect longitudinal data in this format—that is, with repeated rows for each subject—but, as in the case of hierarchical data, they don’t require that the rows for each subject be adjacent in the data set.

## Examining the Data

We begin by looking at the distribution of exercise by group and age. Because ages of observation aren’t identical for different subjects, we group age into

fourths by dividing at the quartiles, creating the new factor agegroup:

```
Blackmore$agegroup <- with(Blackmore,
  cut(age, quantile(age, c(0, 0.25, 0.5, 0.75, 1)),
  include.lowest=TRUE))
xtabs(~ group + agegroup, data=Blackmore)

      agegroup
group      [8,10] (10,12] (12,14] (14,17.9]
control      185       60       58       56
patient      275      118      83      110
```

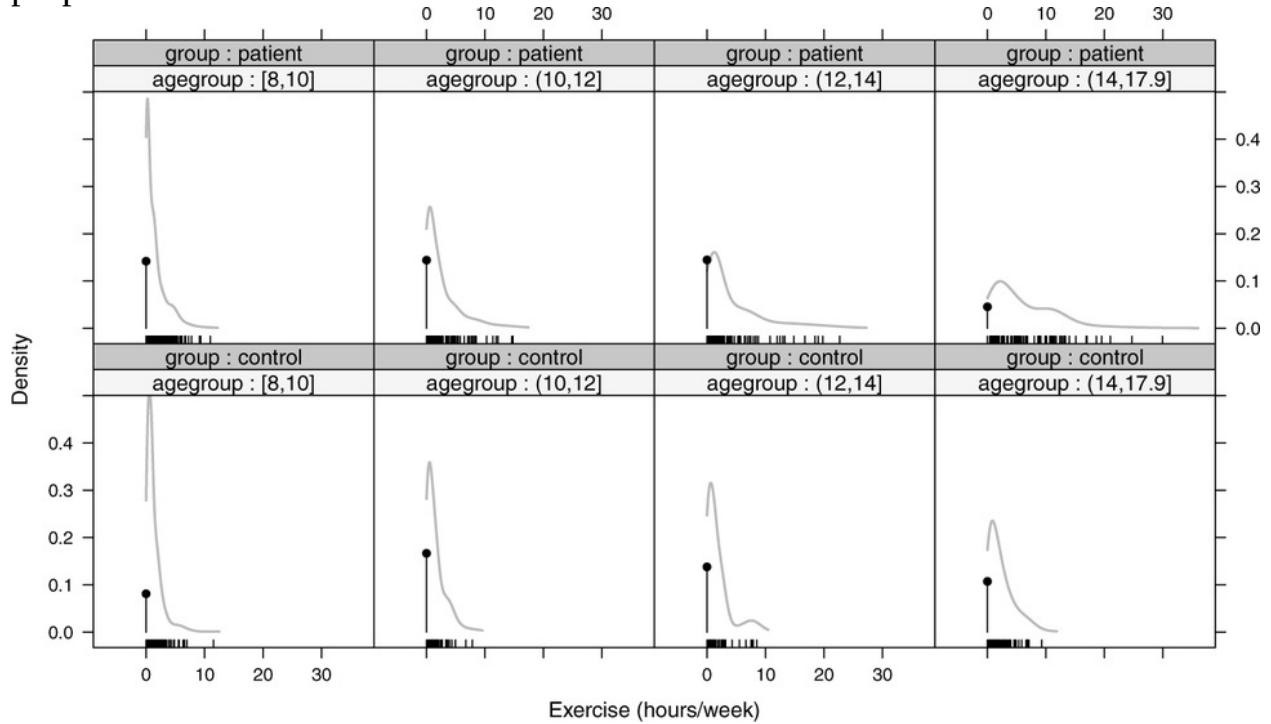
We then draw a density plot for each combination of group and agegroup. Because there are many observations with zero exercise, we also draw spikes representing the proportion of zeros (see [Figure 7.6](#)):

```
densityplot(~ exercise | agegroup + group,
  panel = function(x) {
    res <- adaptiveKernel(x, from=0)
    llines(res$x, res$y, lwd=2, col="darkgray")
    panel.rug(x)
    p <- sum(x == 0)/length(x)
    llines(c(0, 0), c(0, p))
    lpoints(0, p, pch=16)
  },
  strip=strip.custom(strip.names=c(TRUE, TRUE)),
  data=Blackmore, xlab="Exercise (hours/week)")
```

We use the densityplot () function in the **lattice** package to draw the graph.<sup>13</sup> The one-sided formula argument, ~ exercise | agegroup + group, plots the distribution of exercise for combinations of values of the factors agegroup and group.

<sup>13</sup> See [Section 9.3.1](#) for more information on the **lattice** package.

**Figure 7.6** Density plots of weekly hours of exercise by group (patient or control) and agegroup. The spike at the left of each panel shows the proportion of zeros. Because proportions aren't densities, the density estimates and the proportions are not on the same scale.



The work of constructing each panel of the graph in [Figure 7.6](#) is performed by an anonymous panel function, whose single argument, `x`, receives the values of exercise for each panel. The panel function calls `adaptiveKernel()` from the `car` package to compute the density estimate; `adaptiveKernel()` returns a list with elements `$x` and `$y`, which are then used as arguments to the `llines()` function from the `lattice` package to plot the density curve in the panel. Similarly, `panel.rug()` displays the rug-plot at the bottom of each panel, showing the locations of the exercise values in the panel, and `llines()` and `lpoints()` produce the spike at zero exercise showing the proportion of zeros.

The exercise values within each combination of agegroup and group are skewed to the right, with mostly small values. The skewness decreases with age, especially for the patient group. The proportion of zero values of exercise ranges from about 0.05 to about 0.17. Because of the skewness in exercise we expect that the fit of models can be improved by transforming exercise toward normality, and because of the presence of zeros in exercise, we will use the "bcnPower" family with the `powerTransform()` function in `car` ([Section 3.4.2](#)) in preference to an ordinary power transformation.<sup>14</sup>

[14](#) Because all zeros transform into the same value, it would be best to take the zeros explicitly into account in modeling the data. We won't pursue this complication here, but mixed models of this form can be fit with the **glmmTMB** package (Brooks et al., 2017). To get a sense of what's involved, albeit in a simpler context, see our discussion of the zero-inflated Poisson regression model in [Section 10.6.1](#).

The `powerTransform()` function requires that we first use `lmer()` to fit a *target model* for the untransformed response:

```
blackmore.mod.1 <- lmer (exercise ~ age*group + (age|subject),
data=Blackmore)
```

We'll explain the form of the model, in particular the specification of fixed and random effects, later in this section, after we determine how best to transform `exercise`:

```
pwr.black <- powerTransform(blackmore.mod.1, family="bcnPower")
summary(pwr.black)

bcn - Box-Cox Power transformation to Normality
allowing for negative values, lmer fit

Estimated power, lambda
      Est.Power Std.Err. Wald Lower Bound Wald Upper Bound
[1,]    0.2333   0.0189          0.1963        0.2703

Location gamma was fixed at its lower bound
      Est.gamma Std.Err. Wald Lower Bound Wald Upper Bound
[1,]       0.1         NA            NA            NA

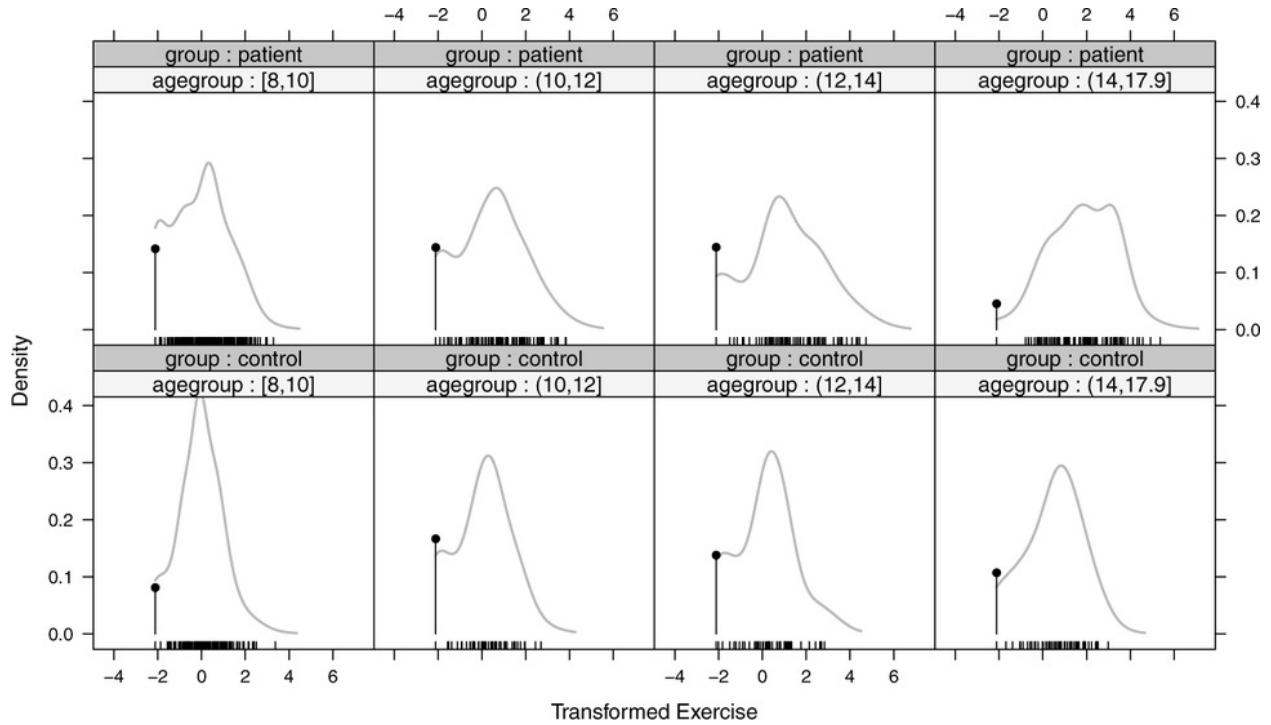
Likelihood ratio tests about transformation parameters
              LRT df pval
LR test, lambda = (0) 88.49042 1    0
LR test, lambda = (1) 1458.57220 1    0
```

This is a slow calculation because it entails refitting the mixed model multiple

times for different values of the transformation parameters, requiring about a minute on our computer to find estimates of the transformation parameters and an additional 30 seconds to compute the summary () output. As it happens, the estimated location parameter in this problem is very close to its minimum possible value of zero, and it is therefore set to  $\gamma = 0.1$ . The power parameter  $\lambda$  is then estimated as if  $\gamma$  were known to be 0.1. The estimated power is , very close to the “nice” value of 0.25, which is contained in the Wald confidence interval for  $\lambda$ . We use the bcnPower () transformation with  $(\lambda, \gamma) = (0.25, 0.1)$ , creating the transformed response tran.exercise, and redrawing the density plots to check on our work:

```
Blackmore$tran.exercise <- bcnPower(Blackmore$exercise,
                                      lambda=0.25, gamma=0.1)
tzero <- min(Blackmore$tran.exercise) # transformed 0
densityplot(~ tran.exercise | agegroup + group,
            panel = function(x) {
              res <- adaptiveKernel(x, from=tzero)
              llines(res$x, res$y, lwd=2, col="darkgray")
              panel.rug(x)
              p <- sum(x == tzero)/length(x)
              llines(c(tzero, tzero), c(0, p))
              lpoints(tzero, p, pch=16)
            },
            strip=strip.custom(strip.names=c(TRUE, TRUE)), data=Blackmore,
            xlab="Transformed Exercise")
```

**Figure 7.7** Transformed exercise by group (patient or control) and agegroup. The spike at the left of each panel shows the transformed values corresponding to zeros on the original exercise scale.



We see in [Figure 7.7](#) that apart from the spikes for the original zeros, we succeeded in making the distributions of `tran.exercise` nearly symmetric for most combinations of group and agegroup.

The density plots ignore subject effects, and so we now turn to analyzing the relationship between `tran.exercise` and age separately for each subject. As in the High School and Beyond example, we will display results only for 20 subjects from each of the patient and control groups, plotting `tran.exercise` against age for each subject:

```

set.seed(12345) # for reproducibility
pat.sample <- with(Blackmore,
                     sample(unique(subject[group == "patient"]), 20))
con.sample <- with(Blackmore,
                     sample(unique(subject[group == "control"]), 20))
print(xyplot(tran.exercise ~ age|subject, data=Blackmore,
             subset=(subject %in% con.sample),
             type=c("p", "r", "g"),
             ylab="Transformed minutes of exercise",
             main="Control Subjects",
             ylim=1.2*range(Blackmore$tran.exercise),
             layout=c(5, 4), aspect=1.0),
             position=c(0, 0, 0.5, 1), more=TRUE)
print(xyplot(tran.exercise ~ age|subject, data=Blackmore,
             subset=(subject %in% pat.sample),
             type=c("p", "r", "g"),
             ylab="Transformed minutes of exercise",
             main="Patients",
             ylim=1.2*range(Blackmore$tran.exercise),
             layout=c(5, 4), aspect=1.0),
             position=c(0.5, 0, 1, 1))

```

We again use the `xyplot()` function from the **lattice** package, specifying several arguments to customize the graph: We include a title with `main`, set the vertical axis label with `ylab`, establish the limits of the vertical axis with `ylim` to be the same in both sets of plots, and control the layout and aspect ratio of the panels. The `type` argument to `xyplot()` is specified in lieu of a panel function and displays points ("p"), the fitted least-squares regression line ("r"), and a background grid ("g") in each panel. We call the `print()` function explicitly to display the "trellis" objects produced by `xyplot()` to specify the position of each graph in the overall plot; see help ("`print.trellis`"). The plots appear in [Figure 7.8](#).<sup>15</sup>

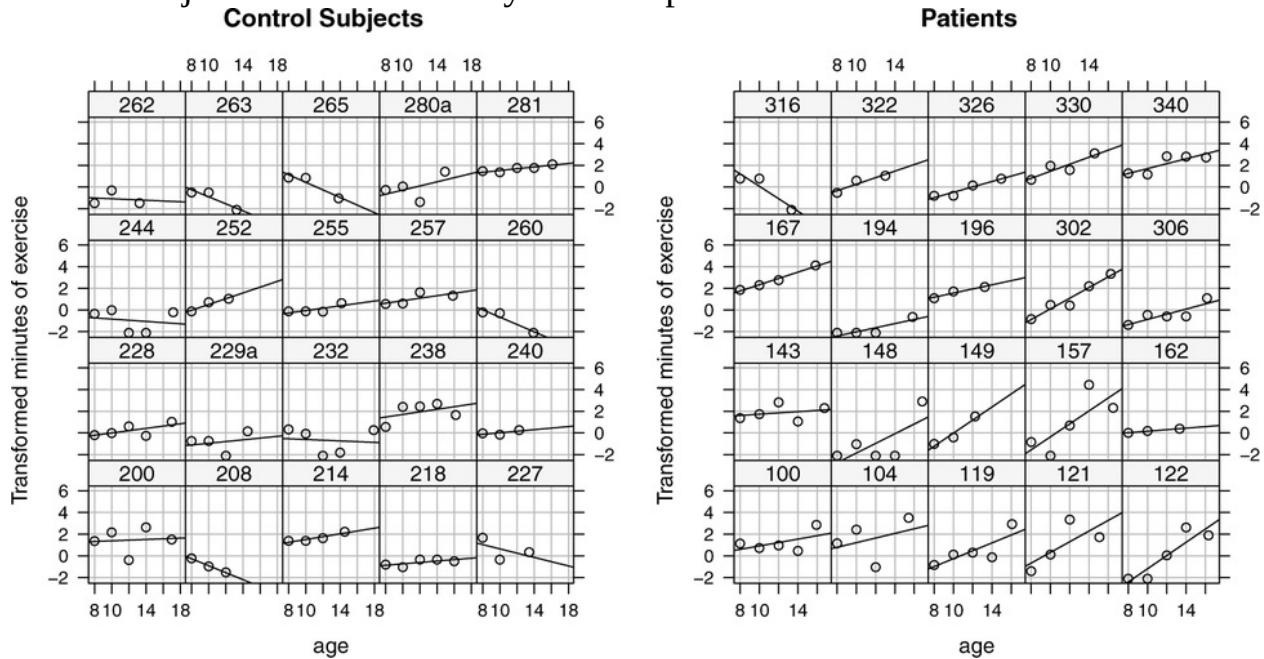
`type=c ("p", "r", "g"),`

```

ylab="Transformed minutes of exercise", main="Patients",
ylim=1.2*range(Blackmore$tran.exercise), layout=c(5, 4), aspect=1.0),
position=c(0.5, 0, 1, 1))

```

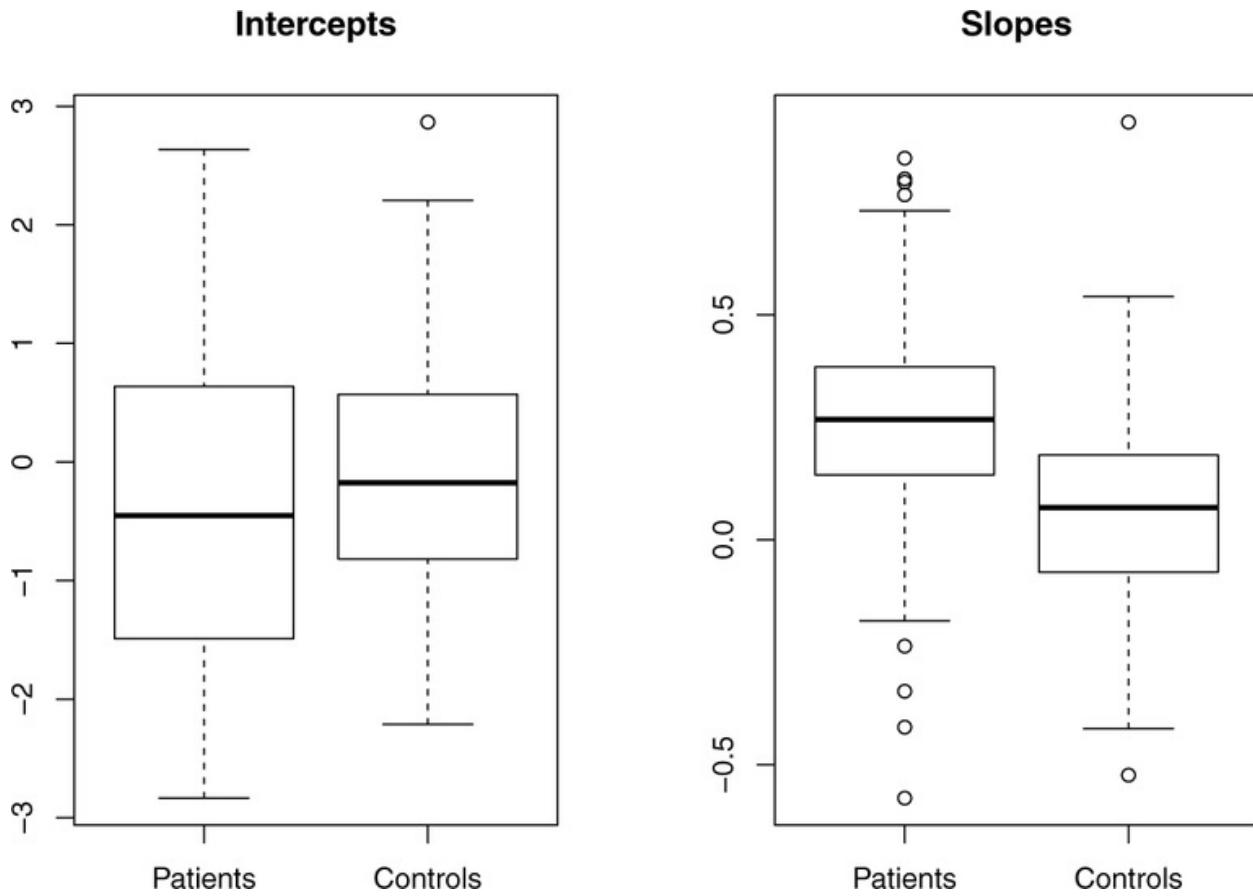
**Figure 7.8** Transformed minutes of exercise by age for 20 randomly selected control subjects and 20 randomly selected patients.



[15](#) Compare this graph and the commands that created it with the Trellis plots that we produced for the High School and Beyond data in [Figures 7.1](#) (page 344) and [7.2](#) (page 345).

There are no more than five data points for each subject, and in many instances, there is no strong within-subject pattern. Nevertheless, it appears as if the general level of exercise is higher among the patients than among the controls. As well, the trend for exercise to increase with age appears stronger and more consistent for the patients than for the controls. We follow up these impressions by constructing side-by-side boxplots for the within-subject intercepts and slopes for the regression of `tran.exercise` on age estimated directly by least squares (much as we did for the schools in the HSB data in [Figure 7.3](#) on page 346):

**Figure 7.9** Coefficients for the within-subject regressions of transformed exercise on age minus 8 years, for patients and control subjects.



```

pat.list <- lmList(tran.exercise ~ I(age - 8) | subject,
                     subset = group=="patient", data=Blackmore)
con.list <- lmList(tran.exercise ~ I(age - 8) | subject,
                     subset = group=="control", data=Blackmore)
pat.coef <- coef(pat.list)
con.coef <- coef(con.list)
old <- par(mfrow=c(1, 2))
boxplot(pat.coef[, 1], con.coef[, 1], main="Intercepts",
        names=c("Patients", "Controls"))
boxplot(pat.coef[, 2], con.coef[, 2], main="Slopes",
        names=c("Patients", "Controls"))
par(old)

```

Boxplots of the within-subjects regression coefficients are shown in [Figure 7.9](#). We changed the origin of age to 8 years, which is the initial age of observation for each subject, so the intercept represents level of transformed exercise at the start of the study.<sup>16</sup> As expected, there is a great deal of variation in both the

intercepts and the slopes. The median intercepts are similar for patients and controls, but there is somewhat more variation among patients. The slopes are noticeably higher on average for patients than for controls, for whom the median slope is close to zero.

[16](#) Because subtraction on the right-hand side of a model formula denotes removing a term from the model, we protected the arithmetic within a call to the `I()` function: See the discussion of linear-model formulas in [Section 4.9.1](#).

## Building a Mixed-Effects Model for the Blackmore Data

Our preliminary examination of the Blackmore data suggests a mixed-effects model that permits each subject to have her own intercept and slope. We build the model in two steps:

- First, consider the possibility that each group of subjects, patients and controls, has a common slope and intercept but that there are no differences in slopes and intercepts among subjects in the same group. Because group is a factor with two levels, let  $g_i$  be a dummy regressor equal to zero for subjects in the control group and 1 for subjects in the patient group. We then have the linear model

$$(7.6) \quad y_{ij} = (\beta_1 + \beta_2 g_i) + (\beta_3 + \beta_4 g_i)x_{ij} + \varepsilon_{ij}$$

where  $y_{ij}^{ij}$  is the response, transformed minutes of exercise, for the  $i$ th of  $M$  subjects measured on the  $j$ th occasion,  $j = 1, \dots, n_i$ ;  $g_i$  is the value of the group dummy regressor for the  $i$ th subject;  $x_{ij}$  is the  $i$ th subject's age on occasion  $j$ ; and  $\varepsilon_{ij}$  is a random error. The population regression line for control subjects is  $\beta_1 + \beta_3 x$  and for patients  $(\beta_1 + \beta_2) + (\beta_3 + \beta_4)x$ .

- We then allow subjects to differ in their individual slopes and intercepts, specifying these individual differences as random deviations from the population slopes and intercepts for the two groups. Let  $b_{1i}$  be the random intercept component and  $b_{2i}$  the random slope component for subject  $i$ . This is, then, a special case of the linear mixed model developed in [Section 7.2](#). From Equation 7.1 (page 337), we have , , and .

Combining the fixed and random effects produces the Laird-Ware form of the linear mixed model:

$$(7.7) \quad y_{ij} = (\beta_1 + \beta_2 g_i + b_{1i}) + (\beta_2 + \beta_3 g_i + b_{2i}) x_{ij} + \varepsilon_{ij}$$

Conditional on the values of  $b_{1i}$  and  $b_{2i}$ , the subject-specific intercept is () and the subject-specific slope is ().

The linear mixed model in Equation 7.7 can be fit to the Blackmore data by either lmer () or lme (). Using lmer ():

```
blackmore.mod.2.lmer <- lmer(tran.exercise ~ I(age - 8)*group +
  (I(age - 8) | subject), data=Blackmore)
S(blackmore.mod.2.lmer)
```

```
Linear mixed model fit by REML
Call: lmer(formula = tran.exercise ~ I(age - 8) * group +
  (I(age - 8) | subject), data = Blackmore)
```

Estimates of Fixed Effects:

|                         | Estimate | Std. Error | z value | Pr(> z ) |
|-------------------------|----------|------------|---------|----------|
| (Intercept)             | -0.1719  | 0.1271     | -1.35   | 0.1762   |
| I(age - 8)              | 0.0602   | 0.0233     | 2.58    | 0.0098   |
| grouppatient            | -0.2260  | 0.1639     | -1.38   | 0.1680   |
| I(age - 8):grouppatient | 0.2013   | 0.0293     | 6.86    | 6.9e-12  |

Estimates of Random Effects (Covariance Components):

| Groups | Name | Std.Dev. | Corr |
|--------|------|----------|------|
|--------|------|----------|------|

|          |             |       |       |
|----------|-------------|-------|-------|
| subject  | (Intercept) | 0.998 |       |
|          | I(age - 8)  | 0.133 | -0.07 |
| Residual |             | 0.879 |       |

Number of obs: 945, groups: subject, 231

| logLik  | df | AIC    | BIC    |
|---------|----|--------|--------|
| -1500.8 | 8  | 3017.7 | 3056.5 |

**Anova (blackmore.mod.2.lmer, test.statistic="F")**

Analysis of Deviance Table  
 (Type II Wald F tests with Kenward-Roger df)

Response: tran.exercise

|                  | F      | Df | Df.res | Pr(>F)  |
|------------------|--------|----|--------|---------|
| I(age - 8)       | 174.88 | 1  | 210    | < 2e-16 |
| group            | 1.82   | 1  | 230    | 0.18    |
| I(age - 8):group | 46.92  | 1  | 221    | 7.2e-11 |

This is the same as the model blackmore.mod.1 that we fit previously to determine a transformation of exercise, except that the response is changed to tran.exercise, transformed minutes of exercise, and we subtract 8 from age so that the intercept represents level of exercise at the start of the study. As usual, the intercept is included by default and need not be specified explicitly in the formula for the random effects. To fit the model *without* a random intercept, we would specify (I (age - 8) - 1 | subject).

As before, we can fit the same model by lme ():

```

blackmore.mod.2.lme <- lme(tran.exercise ~ I(age - 8)*group,
  random = ~ 1 + I(age - 8) | subject, data=Blackmore)
fixef(blackmore.mod.2.lme)

(Intercept)           I(age - 8)
-0.171854          0.060198
grouppatient I(age - 8):grouppatient
-0.226000          0.201274

```

The naïve Wald z-tests for the fixed effects and the Kenward-Roger F-tests in the Anova () output produce similar inferences. We start as usual with the test for the interactions, and in this case, the interaction of age with group has a very small *p*-value, reflecting a much steeper average trend in the patient group. Because of the parametrization of the model, the coefficient for group, the difference in fixed-effect intercepts between the groups, represents the difference in average level of exercise for the two groups at the start of the study; this difference is small, with a large *p*-value. The coefficient for I (age - 8), the estimated fixed-effect slope for the control group, is also associated with a small *p*-value, suggesting that the average level of exercise for control subjects increases with age. Similarly, the intercept is the average level of transformed exercise for the control group at age 8.

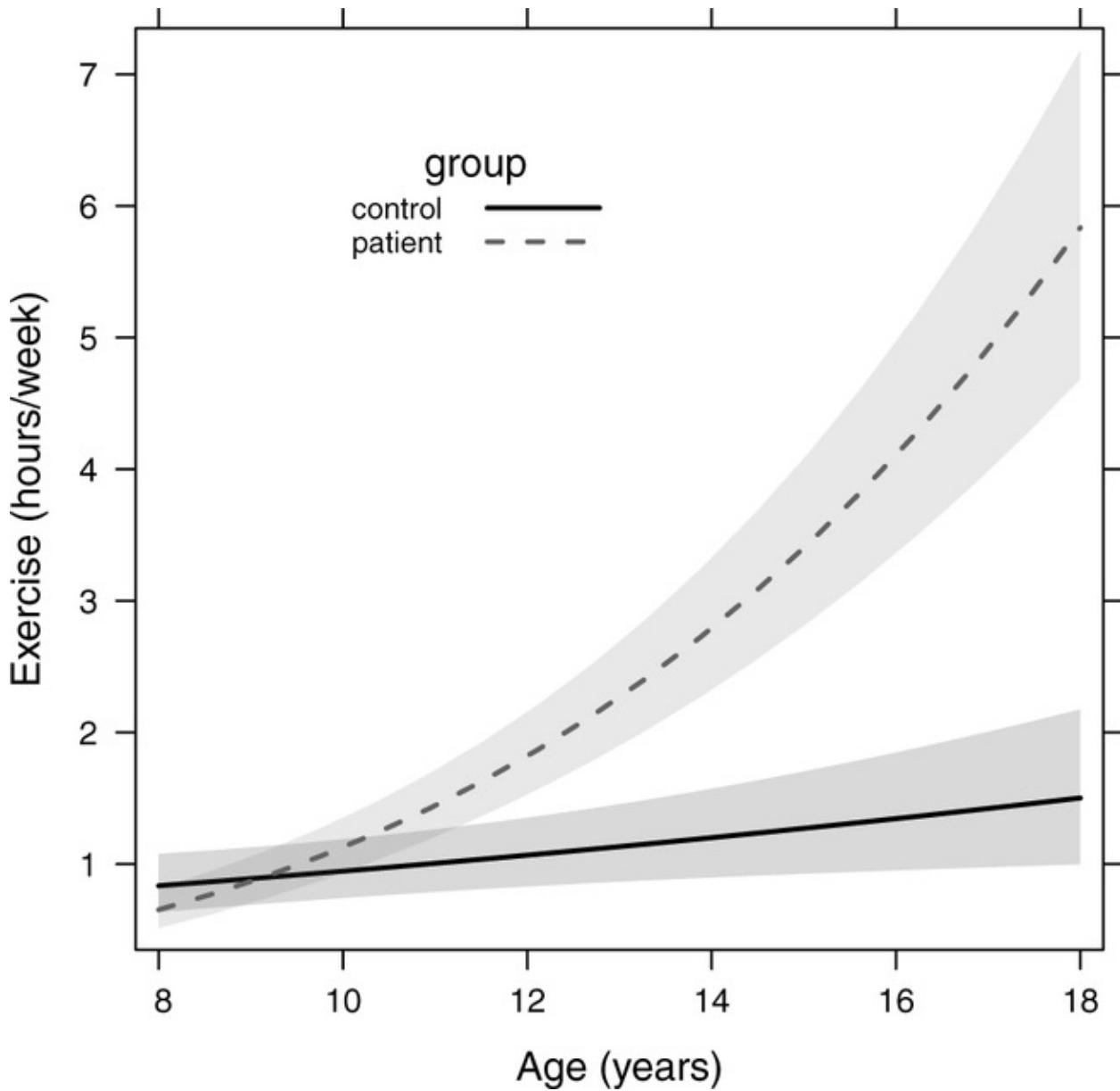
Each of the fixed-effect coefficients has a simple interpretation but in the unfamiliar fourth-root scale using the bcnPower () family of transformations. We can more effectively visualize the fitted model using the Effect () function in the **effects** package, “untransforming” the response on the vertical axis. The result is in [Figure 7.10](#):

```

bm.eff <- Effect(c("age", "group"), blackmore.mod.2.lmer,
  xlevels=list(age=seq(8, 18, by=2)),
  transformation=list(
    link=function(x) bcnPower(x, lambda=0.25, gamma=0.1),
    inverse=function(z)
      bcnPowerInverse(z, lambda=0.25, gamma=0.1)))
plot(bm.eff,
  axes=list(y=list(type="response"), x=list(rug=FALSE)),
  lines=list(multiline=TRUE, lwd=2),
  confint=list(style="bands"),
  lattice=list(key.args=list(x = 0.20, y = 0.75,
    corner = c(0, 0), padding.text = 1.25)),
  xlab="Age (years)", ylab="Exercise (hours/week)", main=""))

```

**Figure 7.10** Effect plot for the interaction of age and group in the mixed-effects model fit to the Blackmore data.



Because this command is complicated, we divide it into two parts:

1. We use the `Effect()` function to specify the predictors in the effect, `age` and `group`. The arguments to `Effect()` include `xlevels`, to indicate where we want the predictor `age` to be evaluated, and `transformation`, to tell the function how to undo the transformation of exercise. `Effect()` returns an object of class "eff", which we save in the R variable `bm.eff`. In specifying the transformation argument list, `link` is an anonymous function for the transformation employed, and `inverse` is an anonymous function that reverses the transformation. The functions `bcnPower()` and

`bcnPowerInverse()` are from the **car** package.<sup>17</sup>

2. We then plot () the object returned by `Effect()`, specifying several arguments. The `axes` argument is a list of lists; we use the `y` element of the list, which is itself a list, to indicate that the plot should be on the scale of the response variable, `exercise`, rather than on the transformed scale, which is the default. We specify the `x` element to omit the default rug-plot showing the marginal distribution of age on the horizontal axis. Similarly, the `confint` argument is a list that controls how confidence intervals are displayed, in this case as confidence bands, and the `lattice` argument controls the placement of the key at the upper left of the graph. A complete list of arguments is given by `help("plot.eff")`.

<sup>17</sup> To take another example, if we had used the transformation `log.exercise <- log(exerise + 0.1)` as the response, we would have set `transform = list(link=function(x) log(x + 0.1), inverse=function(z) (exp(z) - 0.1))`, where the names of the arguments `x` and `z` for the anonymous functions are arbitrary.

It is apparent from [Figure 7.10](#) that the two groups of subjects have similar average levels of exercise at age 8, but that thereafter the level of exercise increases relatively rapidly for the patient group compared to the controls.

We turn next to the random effects and test whether random intercepts and slopes are necessary, omitting each in turn from the model and calculating a likelihood-ratio statistic contrasting the reduced model with the original model:<sup>18</sup>

```

blackmore.mod.3.lmer <- lmer(tran.exercise ~ I(age - 8)*group +
  (1 | subject), data=Blackmore) # no random slopes
anova(blackmore.mod.2.lmer, blackmore.mod.3.lmer, refit=FALSE)

Data: Blackmore
Models:
blackmore.mod.3.lmer: tran.exercise ~ I(age - 8) * group +
  (1 | subject)
blackmore.mod.2.lmer: tran.exercise ~ I(age - 8) * group +
  (I(age - 8) | subject)
      Df AIC BIC logLik deviance Chisq Chi Df
blackmore.mod.3.lmer  6 3049 3078  -1518      3037
blackmore.mod.2.lmer  8 3018 3056  -1501      3002   35.1      2
Pr(>Chisq)

blackmore.mod.3.lmer
blackmore.mod.2.lmer    2.3e-08

pvalCorrected(35.1, 2)
[1] 1.3509e-08

```

[18](#) Recall that we need the argument `refit=FALSE` to prevent `anova()` from refitting the models by maximum likelihood (rather than by REML) prior to computing the likelihood-ratio test.

The null hypothesis for the test is that each subject has her own random intercept but all subjects in the same group have the same slope versus the alternative that each subject has both a random intercept and a random slope. The small *p*-value argues against the null hypothesis of only random intercepts. Similarly,

```

blackmore.mod.4.lmer <- lmer(tran.exercise ~ I(age - 8) * group +
  (age - 1 | subject), Blackmore) # no random intercepts
anova(blackmore.mod.2.lmer, blackmore.mod.4.lmer, refit=FALSE)

Data: Blackmore
Models:
blackmore.mod.4.lmer: tran.exercise ~ I(age - 8) * group +
  (age - 1 | subject)
blackmore.mod.2.lmer: tran.exercise ~ I(age - 8) * group +
  (I(age - 8) | subject)
      Df AIC BIC logLik deviance Chisq Chi Df
blackmore.mod.4.lmer 6 3042 3071 -1515      3030
blackmore.mod.2.lmer 8 3018 3056 -1501      3002   28.5      2
Pr(>Chisq)

blackmore.mod.4.lmer
blackmore.mod.2.lmer      6.5e-07

pvalCorrected(28.5, 2)

[1] 3.7065e-07

```

Here the null hypothesis is that we need only subject-specific slopes, and the small  $p$ -value suggests strong evidence against the null hypothesis. Thus, both subject-specific slopes and intercepts appear to be required. In both cases, we use our `pval-Corrected()` function (introduced on page 354) to compute correct  $p$ -values with random-effect variances in the null hypotheses set to zero.

## 7.2.6 Modeling the Errors

The examples developed in [Sections 7.2.2](#) and [7.2.5](#), respectively for the HSB and Blackmore data, both assume that the errors  $\varepsilon_{ij}$  are independent with constant variance. Among the many generalizations of mixed models is relaxation of this assumption, accommodating more elaborate models for the within-group errors that permit nonconstant error variance, autocorrelation of the errors, or both. While generalization of the random effects is theoretically appealing for some problems, relaxing the assumptions concerning the variances and covariances of the  $\varepsilon_{ij}$  rarely has much influence on estimates of the fixed effects, which are typically of central interest.

The most general assumption is simply that and , where the  $\lambda^i$ s are arbitrary

variances and covariances for the observations in the  $i$ th group, to be estimated from the data.<sup>19</sup> This unrestricted specification is rarely feasible because the number of free parameters grows with the square of the number of observations in each group, making estimation impossible. It is much more common to constrain the error variances and covariances so that they depend on only a handful of parameters. In both longitudinal and spatial applications of mixed models, error covariances are typically made to depend on the “distance” (in time or space) between the corresponding observations. A common example for longitudinal data is autocorrelated errors, where the correlation between two errors for the same subject depends on the time difference between them and on one or more parameters to be estimated from the data.

<sup>19</sup>\* In matrix terms,  $\text{Var}(\varepsilon_i) = \boldsymbol{\Lambda}_i$  for a completely general positive-definite-matrix  $\boldsymbol{\Lambda}_i$ .

Complex specifications for the errors in a mixed model, including for spatial data, are discussed in Pinheiro and Bates (2000, [Section 5.3](#)) and implemented in the lme () function, but not in lmer (). Most of the error correlation structures provided by lme () for longitudinal data are appropriate only for observations that are equally spaced in time. An exception is the corCAR1 () function, which permits us to fit a *continuous first-order autoregressive process* in the errors. Suppose that  $\varepsilon_{ij}$  and  $\varepsilon_{ij'}$  are errors for subject  $i$  separated by  $t$  units of time, where  $t$  need not be an integer; then, according to the continuous first-order autoregressive model, the correlation between these two errors is  $\rho(s) = \phi_{|t|}$  where  $0 \leq \phi < 1$ , and the autoregressive parameter  $\phi$  is to be estimated from the data along with the fixed effects and variance and covariance components.

A continuous first-order autoregressive model for the errors seems a reasonable specification for the time-ordered Blackmore data discussed in [Section 7.2.5](#). We modify the model blackmore.mod.2.lme that we previously fit (page 368), adding autocorrelated errors to the model:

```
blackmore.mod.5.lme <- update (blackmore.mod.2.lme, correlation =
corCAR1(form = ~ I (age - 8) | subject))
```

The autocorrelation structure of the errors is given in the correlation argument to lme (); the form argument to corCAR1 () is a one-sided formula defining the

time dimension for the continuous first-order autoregressive process.

Unfortunately, lme () is unable to fit this model to the data (as the reader can verify): We have too few repeated observations per subject to estimate  $\phi$  along with the variance and covariance components, which also imply correlations among the responses  $y^{ij}$  for each subject. It is, however, possible to estimate models that include autocorrelated errors along with *either* random intercepts or random slopes, but not *both*:

```
blackmore.mod.6.lme <- update(blackmore.mod.2.lme,
  random = ~ 1|subject, # random intercepts
  correlation = corCAR1(form = ~ I(age - 8) | subject))
blackmore.mod.7.lme <- update(blackmore.mod.2.lme,
  random = ~ I(age - 8) - 1 | subject, # random slopes
  correlation = corCAR1(form = ~ I(age - 8) | subject))
```

The models blackmore.mod.2.lme (with random slopes and intercepts, but independent errors), blackmore.mod.6.lme (with random intercepts and auto-correlated errors), and blackmore.mod.7.lme (with random slopes and auto-correlated errors) aren't properly nested for likelihood-ratio tests. We can, however, compare the models using the AIC or BIC, both of which strongly prefer blackmore.mod.6.lme:

**AIC**(*blackmore.mod.2.lme*, *blackmore.mod.6.lme*,  
*blackmore.mod.7.lme*)

|                            | df | AIC    |
|----------------------------|----|--------|
| <i>blackmore.mod.2.lme</i> | 8  | 3017.7 |
| <i>blackmore.mod.6.lme</i> | 7  | 3001.6 |
| <i>blackmore.mod.7.lme</i> | 7  | 3012.0 |

**BIC**(*blackmore.mod.2.lme*, *blackmore.mod.6.lme*,  
*blackmore.mod.7.lme*)

|                            | df | BIC    |
|----------------------------|----|--------|
| <i>blackmore.mod.2.lme</i> | 8  | 3056.4 |
| <i>blackmore.mod.6.lme</i> | 7  | 3035.5 |
| <i>blackmore.mod.7.lme</i> | 7  | 3045.9 |

All three models produce similar estimates of the fixed effects and their standard errors:

```
compareCoefs(blackmore.mod.2.lme, blackmore.mod.6.lme,
blackmore.mod.7.lme)
```

Calls:

```
1: lme.formula(fixed = tran.exercise ~ I(age - 8) * group,
   data = Blackmore, random = ~1 + I(age - 8) | subject)
2: lme.formula(fixed = tran.exercise ~ I(age - 8) * group,
   data = Blackmore, random = ~1 | subject, correlation =
corCAR1(form = ~I(age - 8) | subject))
3: lme.formula(fixed = tran.exercise ~ I(age - 8) * group,
   data = Blackmore, random = ~I(age - 8) - 1 | subject,
correlation = corCAR1(form = ~I(age - 8) | subject))
```

|                         | Model 1 | Model 2 | Model 3 |
|-------------------------|---------|---------|---------|
| (Intercept)             | -0.172  | -0.196  | -0.203  |
| SE                      | 0.127   | 0.143   | 0.137   |
| <br>                    |         |         |         |
| I(age - 8)              | 0.0602  | 0.0668  | 0.0668  |
| SE                      | 0.0233  | 0.0231  | 0.0279  |
| <br>                    |         |         |         |
| grouppatient            | -0.226  | -0.157  | -0.134  |
| SE                      | 0.164   | 0.184   | 0.177   |
| <br>                    |         |         |         |
| I(age - 8):grouppatient | 0.2013  | 0.1894  | 0.1926  |
| SE                      | 0.0293  | 0.0290  | 0.0352  |

## 7.2.7 Sandwich Standard Errors for Least-Squares Estimates

A partial alternative to mixed-effects models is to estimate a fixed-effects regression model by least squares and then to correct coefficient standard errors for dependencies among the regression model errors induced by clustering. This strategy is an extension of the sandwich standard errors introduced in [Section 5.1.2](#) for dealing with nonconstant error variance. The appropriate adjustment must estimate and account for covariances between observations in the same cluster. For the constant correlation structure implied by many mixed-effects

models, the `vcovCL()` function in the **sandwich** package (Zeileis, 2004) can be used in place of the `hccm()` function employed previously to adjust for nonconstant error variance.

For an example, we return to the High School and Beyond data set (in the HSB data frame) that we examined in [Section 7.2.2](#). One linear mixed-effects model that we fit to the HSB data regressed students' math achievement (`mathach`) on school mean SES (`mean.ses`) crossed with the individual students' school-centered SES (`cse`) and school-centered SES crossed with the factor sector ("Public" or "Catholic"). Our model also included random `cse` intercepts and slopes within schools.<sup>20</sup> The results (for model `hsb.lme.1`) are shown on page 350. The analogous model fit by OLS regression is

```
hsb.lm <- lm (mathach ~ mean.ses*cse + sector*cse, data=HSB)
```

[20](#) Because the mixed model includes not only random intercepts but also random slopes, it doesn't imply the constant-correlation structure assumed by `vcovCL()`. Nevertheless, as we're about to see, we get very similar results for the mixed model and for OLS regression with cluster-corrected standard errors.

Let us compare coefficient estimates and their standard errors for these two models, using both conventional and sandwich estimates of the standard errors for model `hsb.lm`:

```

library("sandwich") # for vcovCL()
compareCoefs(hsb.lme.1, hsb.lm, hsb.lm,
  vcov.=list(vcov, vcov, vcovCL(hsb.lm, cluster=HSB$school)))

Warning in compareCoefs(hsb.lme.1, hsb.lm, hsb.lm, vcov. =
list(vcov, vcov, : models to be compared are of different classes

Standard errors computed by list(vcov, vcov, vcovCL(hsb.lm,
  cluster = HSB$school))

Call:
1: lme.formula(fixed = mathach ~ mean.ses * cses + sector *
  cses, data = HSB, random = ~cses | school)
2: lm(formula = mathach ~ mean.ses * cses + sector * cses,
  data = HSB)
3: lm(formula = mathach ~ mean.ses * cses + sector * cses,
  data = HSB)

      Model 1 Model 2 Model 3
(Intercept)    12.128   12.116   12.116
SE             0.199     0.107     0.170

mean.ses       5.333    5.164    5.164
SE             0.369     0.191     0.335

cses          2.945    2.942    2.942
SE             0.156     0.156     0.148

sectorCatholic 1.227    1.281    1.281
SE             0.306     0.158     0.300
mean.ses:cses           1.039    1.043    1.043
SE                  0.299    0.300    0.334

cses:sectorCatholic -1.642   -1.642   -1.642
SE                 0.240    0.240    0.238

```

The optional `vcov`. argument to `compareCoefs()` allows us to control how the coefficient standard errors are computed for each of the models to be compared, with `vcov()` returning the usual coefficient covariance matrices for both the mixed model and the model fit by OLS, while `vcovCL()` computes the cluster-corrected coefficient covariance matrix for the OLS estimates. The cluster

argument to `vcovCL()` specifies the cluster—in this case, the school—to which each observation belongs. The warning message in the output simply reflects the fact that `hsb.lme.1` is of class "lme" while `hsb.lm` is of class "lm".

The estimated fixed-effects regression coefficients from the mixed model and the OLS estimates are very similar for this example. The conventional standard errors for the OLS estimates of the coefficients of the compositional predictor `mean.ses` and the contextual predictor `sector`, however, are much smaller than the corresponding standard errors for the fixed-effects coefficients in the mixed model. Finally, the sandwich standard errors for the OLS estimates, which take clustering into account, are very similar to the standard errors for the fixed-effects estimates.

## 7.3 Generalized Linear Mixed Models

*Generalized linear mixed models (GLMMs)* bear the same relationship to linear mixed models that generalized linear models bear to linear models (cf. [Chapters 4](#) and [6](#)). GLMMs add random effects to the linear predictor of a GLM and express the expected value of the response conditional on the random effects: The link function  $g$  is the same as in generalized linear models. For a GLMM with two levels of hierarchy, the conditional distribution of  $y_{ij}$ , the response for case  $j$  in group  $i$ , given the random effects, is (most straightforwardly) a member of an exponential family, with mean  $\mu_{ij}$ , variance

$$\text{Var}(y_{ij}) = \phi V(\mu_{ij}) \lambda_{ij}$$

and covariances

$$\text{Cov}(y_{ij}, y_{ij'}) = \phi \sqrt{V(\mu_{ij})} \sqrt{V(\mu_{ij'})} \lambda_{ijj'}$$

where  $\phi$  is a dispersion parameter and the function  $V(\mu_{ij})$  depends on the distributional family to which  $y$  belongs (see [Table 6.2](#) on page 274). For example, in the binomial and Poisson families, the dispersion is fixed to  $\phi = 1$ , and in the Gaussian family,  $V(\mu) = 1$ . Alternatively, for quasi-likelihood estimation,  $V(\mu)$  can be given directly, as in generalized linear models.

The GLMM may be written as

$$\eta_{ij} = \beta_1 + \beta_2 x_{2ij} + \cdots + \beta_p x_{pij} + b_{1i} z_{1ij} + \cdots + b_{qi} z_{qij}$$

$$g(\mu_{ij}) = g[\mathbb{E}(y_{ij}|b_{1i}, \dots, b_{qi})] = \eta_{ij}$$

$$b_{ki} \sim N(0, \psi_k^2), \text{Cov}(b_{ki}, b_{k'i}) = \psi_{kk'}$$

are independent for  $i \neq i'$

$$\text{Var}(y_{ij}) = \phi V(\mu_{ij}) \lambda_{ij}$$

$$\text{Cov}(y_{ij}, y_{ij'}) = \phi \sqrt{V(\mu_{ij})} \sqrt{V(\mu_{ij'})} \lambda_{ijj'}$$

$y_{ij}, y_{ij'}$  are independent for  $i \neq i'$

are independent for  $i \neq i'$

where  $\eta_{ij}$  is the linear predictor for case  $j$  in cluster  $i$ ; the fixed-effect coefficients ( $\beta$ s), random-effect coefficients ( $bs$ ), fixed-effect regressors ( $xs$ ), and random-effect regressors ( $zs$ ) are defined as in the LMM ([Section 7.2](#)).<sup>21</sup>

<sup>21</sup> Like the `lmer()` function, the `glmer()` function in the **lme4** package that we will use to fit GLMMs is somewhat more restrictive, setting  $\lambda_{ij} = 1$  and  $\lambda_{ij'} = 0$ .

### 7.3.1 Matrix Form of the GLMM\*

In matrix form, the GLMM is

$$\boldsymbol{\eta}_i = \mathbf{X}_i \boldsymbol{\beta} + \mathbf{Z}_i \mathbf{b}_i$$

$$g(\boldsymbol{\mu}_i) = g[\mathbb{E}(\mathbf{y}_i|\mathbf{b}_i)] = \boldsymbol{\eta}_i$$

$$\mathbf{b}_i \sim \mathbf{N}_q(0, \boldsymbol{\Psi})$$

are independent for  $i \neq i'$

$$\mathbb{E}(\mathbf{y}_i|\mathbf{b}_i) = \boldsymbol{\mu}_i$$

$$\text{Var}(\mathbf{y}_i|\mathbf{b}_i) = \phi V^{1/2}(\boldsymbol{\mu}_i) \boldsymbol{\Lambda}_i V^{1/2}(\boldsymbol{\mu}_i)$$

are independent for  $i \neq i'$

where

- $\mathbf{y}_i$  is the  $n_i \times 1$  response vector for cases in the  $i$ th of  $m$  groups;
- $\boldsymbol{\mu}_i$  is the  $n_i \times 1$  expectation vector for the response, conditional on the random effects;
- $\boldsymbol{\eta}_i$  is the  $n_i \times 1$  linear predictor for the elements of the response vector;
- $g(\cdot)$  is the link function, transforming the conditional expected response to the linear predictor;
- $\mathbf{X}_i$  is the  $n_i \times p$  model matrix for the fixed effects of cases in group  $i$ ;
- $\boldsymbol{\beta}$  is the  $p \times 1$  vector of fixed-effect coefficients;
- $\mathbf{Z}_i$  is the  $n_i \times q$  model matrix for the random effects of cases in group  $i$ ;
- $\mathbf{b}_i$  is the  $q \times 1$  vector of random-effect coefficients for group  $i$ ;
- $\boldsymbol{\Psi}$  is the  $q \times q$  covariance matrix of the random effects;
- $\boldsymbol{\Lambda}_i$  is  $n_i \times n_i$  and expresses the dependence structure for the conditional distribution of the response within each group—for example, if the cases are sampled independently in each group, ;<sup>22</sup>
- ; and
- $\phi$  is the dispersion parameter.

<sup>22</sup> As mentioned, this restriction is imposed by the `glmer()` function in the **lme4** package. See footnote 21.

### 7.3.2 Example: Minneapolis Police Stops

In [Sections 2.3.5](#) and [2.3.6](#), we introduced data on stops of individuals conducted by the Minneapolis police during 2017, merging data on the individual stops, from the `MplsStops` data set, with demographic data on Minneapolis neighborhoods, from the `MplsDemo` data set; both data sets are in the **carData** package. We perform a similar operation here, adding the demographic data to the data on individual stops:

```

Mpls <- merge(MplsStops, MplsDemo, by="neighborhood")
nrow(Mpls)

[1] 49620

names(Mpls)

[1] "neighborhood"      "idNum"           "date"
[4] "problem"            "MDC"              "citationIssued"
[7] "personSearch"       "vehicleSearch"    "preRace"
[10] "race"                "gender"            "lat"
[13] "long"                 "policePrecinct" "population"
[16] "white"                "black"             "foreignBorn"
[19] "hhIncome"              "poverty"          "collegeGrad"

```

There are nearly 50,000 stops that occurred in neighborhoods included in the MplsDemo data set; three neighborhoods in the MplsStops data set have no housing and therefore no demographic data. The stops for these unmatched neighborhoods were dropped rather than merged (see [Section 2.3.6](#)). We focus on non- traffic stops, for which problem == "suspicious", and on the stop-level variables race, gender, and personSearch (whether or not the person stopped was searched) and the single neighborhood-level variable black (the proportion of the population that is black according to the 2015 American Community Survey).

A peculiarity of these data is that the variables race, gender, and person- Search were collected only for police stops recorded on a squad-car computer, indicated by MDC == "MDC", while other types of stops, conducted by police on foot, bicycle, or horseback, are coded as MDC == "other". We retain only "suspicious" stops recorded in a squad car:

```

Mpls <- subset(Mpls,
  subset = problem == "suspicious" & MDC == "MDC",
  select=c("neighborhood", "race", "gender", "black",
  "personSearch"))
summary(Mpls)

```

|                  | neighborhood |                 | race         |
|------------------|--------------|-----------------|--------------|
| Downtown         | West : 1823  | Unknown         | : 8197       |
| Whittier         | : 1160       | Black           | : 5049       |
| East Phillips    | : 810        | White           | : 3475       |
| Lyndale          | : 762        | Native American | : 1168       |
| Jordan           | : 674        | East African    | : 639        |
| Midtown Phillips | : 655        | Latino          | : 522        |
| (Other)          | : 13817      | (Other)         | : 651        |
|                  | gender       | black           | personSearch |
| Female : 2700    | Min.         | : 0.004         | NO : 16705   |
| Male : 11110     | 1st Qu.      | : 0.122         | YES: 2996    |
| Unknown: 5845    | Median       | : 0.211         |              |
| NA's : 46        | Mean         | : 0.230         |              |
|                  | 3rd Qu.      | : 0.315         |              |
|                  | Max.         | : 0.656         |              |

The subset argument selects cases based on both problem and MDC. For simplicity, we further reduce the data set to include only individuals for whom race == "White", "Black", or "Native American" and those for whom gender == "Female" or "Male". There are relatively few individuals of other races, and we choose to treat cases for which gender == "Unknown" or race == "Unknown" as missing data.<sup>23</sup>

```

Mpls$race <- Recode(Mpls$race,
  ' "White" = "White"; "Black" = "Black";
  "Native American" = "Native American"; else=NA ')
Mpls$gender[Mpls$gender == "Unknown"] <- NA
Mpls$gender <- droplevels(Mpls$gender)
Mpls <- na.omit(Mpls)
Mpls$neighborhood <- droplevels(Mpls$neighborhood)
Mpls$race <- factor(Mpls$race,
  levels=c("White", "Black", "Native American"))
summary(Mpls)

```

|               | neighborhood  | race                 | gender      |
|---------------|---------------|----------------------|-------------|
| Downtown      | West :1407    | White :3457          | Female:2087 |
| Whittier      | : 554         | Black :5007          | Male :7525  |
| East Phillips | : 466         | Native American:1148 |             |
| Lyndale       | : 447         |                      |             |
| Hawthorne     | : 377         |                      |             |
| Midtown       | Phillips: 327 |                      |             |
| (Other)       |               | :6034                |             |
|               | black         | personSearch         |             |
| Min.          | :0.004        | NO :7070             |             |
| 1st Qu.       | :0.135        | YES:2542             |             |
| Median        | :0.211        |                      |             |
| Mean          | :0.233        |                      |             |
| 3rd Qu.       | :0.297        |                      |             |
| Max.          | :0.656        |                      |             |

**nrow(Mpls)**

[1] 9612

[23](#) We are implicitly assuming the reasons for using the code "Unknown" for either of these variables is independent of the response variable personSearch. If,

for example, race="Unknown" is much more frequent when person searches are not carried out, then the rate of searches will be overestimated by treating "Unknown" as missing. An online appendix to the *R Companion* discusses multiple imputation of missing data.

Our strategy in subsetting the data is to recode factor levels to be deleted as NA and then to delete cases with NAs by na.omit(). The droplevels() function eliminates the now-empty "Unknown" level from the factor gender and the three levels of the factor neighborhood with no data. Although it is, as usual, inessential to do so, we reorder the levels of the factor race to simplify interpretation. The data set has consequently been reduced to 9,612 non-traffic stops in neighborhoods with housing, conducted from a squad car, of white, black, or Native American individuals whose gender was recorded as female or male.

We're interested in how the decision to search individuals is potentially influenced by their race and gender. We can formulate a preliminary answer to this question by ignoring neighborhood effects, cross-classifying personSearch by race and gender:

```
ftable(round(100 *
prop.table(xtabs(~ race + gender + personSearch, data=Mpls),
margin=c(1, 2)),
1))
```

|                 |        | personSearch |      |
|-----------------|--------|--------------|------|
|                 |        | NO           | YES  |
| race            | gender |              |      |
| White           | Female | 84.2         | 15.8 |
|                 | Male   | 81.7         | 18.3 |
| Black           | Female | 78.7         | 21.3 |
|                 | Male   | 66.9         | 33.1 |
| Native American | Female | 73.0         | 27.0 |
|                 | Male   | 64.5         | 35.5 |

We use the xtabs() function to cross-classify the three factors, the prop.table() function to calculate proportions within margins one and two of the table (i.e., race and gender), and ftable() to "flatten" the three-way table for printing. Multiplying by 100 and rounding produces percentages to one decimal place.

Within race, males are more likely than females to be searched, with the difference between the genders greater for blacks and Native Americans than for whites. Within gender, Native Americans and blacks are more likely than whites to be searched, with the difference between the races generally greater for males than for females, although female Native Americans are searched at a relatively high rate. This pattern suggests an interaction between race and gender in determining whether or not a stopped individual is searched.

Before formulating a mixed-effects logistic-regression model for the binary response personSearch, let's consider how much data we have within neighborhoods by looking at selected quantiles of the distribution of the number of observations per neighborhood:

```
summary(as.vector(xtabs(~ neighborhood, data=Mpls)))
```

| Min. | 1st Qu. | Median | Mean  | 3rd Qu. | Max.   |
|------|---------|--------|-------|---------|--------|
| 5.0  | 25.2    | 47.5   | 114.4 | 142.5   | 1407.0 |

```
summary(as.vector(xtabs(~ neighborhood + gender + race,  
data=Mpls)))
```

| Min. | 1st Qu. | Median | Mean | 3rd Qu. | Max.  |
|------|---------|--------|------|---------|-------|
| 0.0  | 1.0     | 6.0    | 19.1 | 17.0    | 828.0 |

Each call to xtabs () returns a table of counts—in the first instance, of the number of stops in each neighborhood and in the second, for each neighborhood, gender, and race combination. Each table is changed to a vector of counts for which the summary () function returns a few summary statistics. The counts vary from 5 to 1,407 within neighborhoods, and from zero to 828 in neighborhood×gender×race cells.

We consequently opt for a simple random-effects structure for our mixed model, with a random intercept potentially varying by neighborhood. We include in the fixed part of the model the main effects and an interaction for the individual-level factors race and gender, along with a main effect for the neighborhood-level contextual predictor black:<sup>24</sup>

```
mod.mpls <- glmer (personSearch ~ race*gender + black
```

`+ (1 | neighborhood), data=Mpls, family=binomial)`

24 In Figure 9.23 (page 475), we draw a map of the number of police stops (including both traffic and non–traffic stops) in the various Minneapolis neighborhoods. It’s clear from this map that adjacent neighborhoods tend to have similar numbers of stops, suggesting that it might make sense to take geographic autocorrelation into account in modeling the data. That, however, is beyond the capabilities of `glmer()`, which assumes that  $\lambda_{ijj'} = 0$  in the GLMM (Equation 7.8).

Specification of the fixed and random effects (the latter in parentheses) is the same as for `lmer()`. We add the `family` argument to fit a binomial GLMM with the default logit link.

### **Anova (`mod.mpls`)**

Analysis of Deviance Table (Type II Wald chisquare tests)

Response: personSearch

|             | Chisq  | Df | Pr (>Chisq) |
|-------------|--------|----|-------------|
| race        | 109.60 | 2  | < 2e-16     |
| gender      | 57.79  | 1  | 2.9e-14     |
| black       | 14.97  | 1  | 0.00011     |
| race:gender | 9.22   | 2  | 0.00994     |

### **S (`mod.mpls`)**

Generalized linear mixed model fit by ML  
 Call: glmer(formula = personSearch ~ race \* gender + black + (1  
 | neighborhood), data = Mpls, family = binomial)

Estimates of Fixed Effects:

|                                | Estimate | Std. Error | z value |
|--------------------------------|----------|------------|---------|
| (Intercept)                    | -2.041   | 0.128      | -15.92  |
| raceBlack                      | 0.152    | 0.132      | 1.15    |
| raceNative American            | 0.599    | 0.148      | 4.05    |
| genderMale                     | 0.231    | 0.111      | 2.08    |
| black                          | 1.358    | 0.351      | 3.87    |
| raceBlack:genderMale           | 0.437    | 0.145      | 3.02    |
| raceNative American:genderMale | 0.213    | 0.175      | 1.22    |
|                                | Pr(> z ) |            |         |
| (Intercept)                    | < 2e-16  |            |         |
| raceBlack                      | 0.25113  |            |         |
| raceNative American            | 5e-05    |            |         |
| genderMale                     | 0.03774  |            |         |
| black                          | 0.00011  |            |         |
| raceBlack:genderMale           | 0.00251  |            |         |
| raceNative American:genderMale | 0.22330  |            |         |

Exponentiated Fixed Effects and Confidence Bounds:

|                                | Estimate | 2.5 %   | 97.5 %  |
|--------------------------------|----------|---------|---------|
| (Intercept)                    | 0.12986  | 0.10099 | 0.16697 |
| raceBlack                      | 1.16364  | 0.89828 | 1.50738 |
| raceNative American            | 1.81981  | 1.36249 | 2.43062 |
| genderMale                     | 1.25967  | 1.01316 | 1.56615 |
| black                          | 3.88890  | 1.95458 | 7.73747 |
| raceBlack:genderMale           | 1.54785  | 1.16597 | 2.05481 |
| raceNative American:genderMale | 1.23686  | 0.87849 | 1.74142 |

Estimates of Random Effects (Covariance Components):

| Groups       | Name        | Std.Dev. |
|--------------|-------------|----------|
| neighborhood | (Intercept) | 0.349    |

Number of obs: 9612, groups: neighborhood, 84

| logLik  | df | AIC     | BIC     |
|---------|----|---------|---------|
| -5305.9 | 8  | 10627.8 | 10685.2 |

The Wald tests reported by the Anova () function have small  $p$ -values for the race  $\times$  gender interaction and for the proportion black in the neighborhood.

The exponentiated coefficient for black, , suggests that, fixing race and gender, the odds of being searched would be four times higher in an entirely black neighborhood than in an entirely white neighborhood.<sup>25</sup>

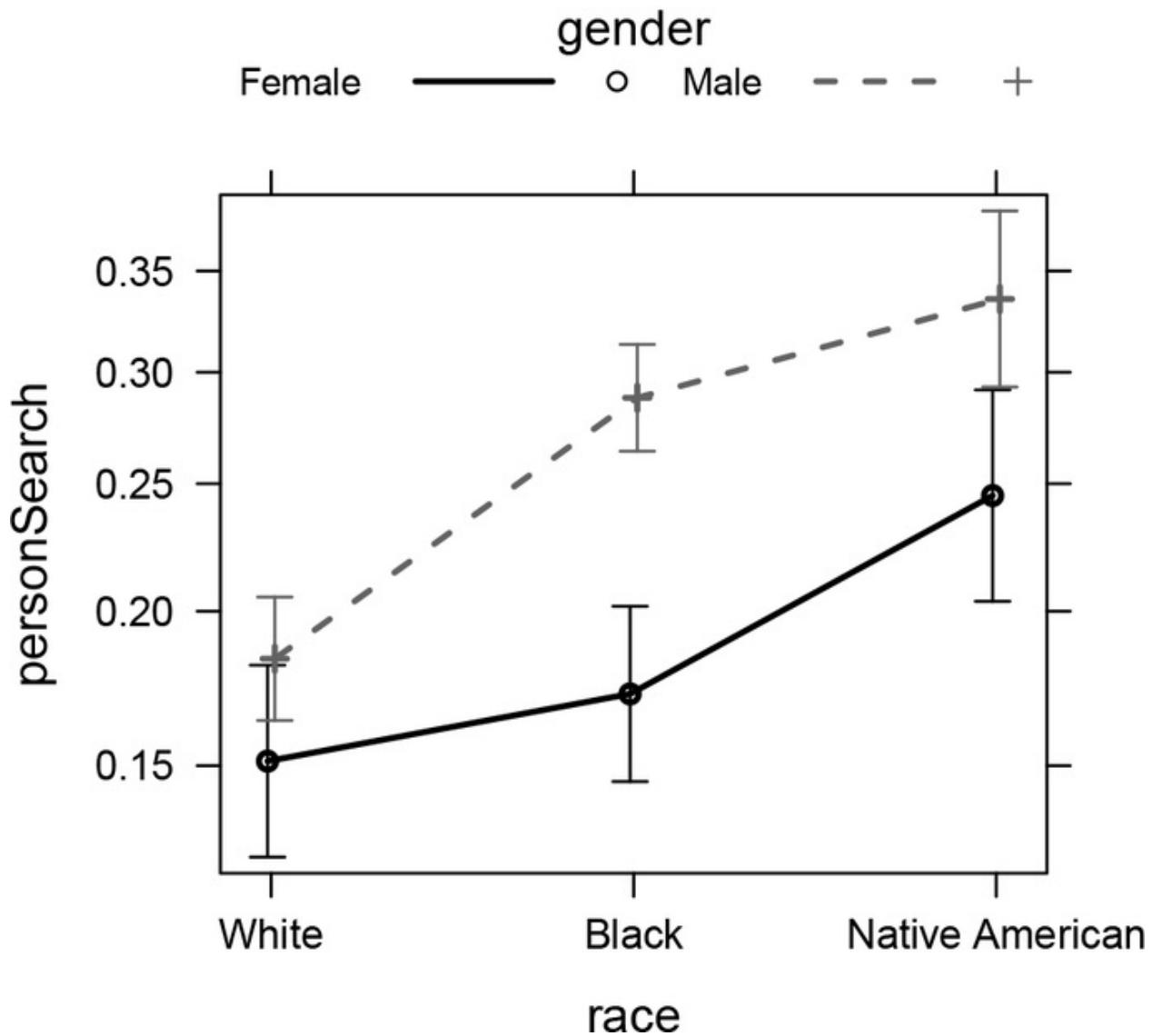
<sup>25</sup> This extrapolates beyond the range of the data—the minimum proportion black in a neighborhood is close to zero but the maximum is approximately 0.7. The confidence limits for the multiplicative effect of black are wide, ranging from approximately a factor of 2 to a factor of 8.

We turn to an effect display ([Figure 7.11](#)) to interpret the interaction between race and gender:

```
plot(Effect(c("race", "gender"), mod.mpls), lines=list  
      (multiline=TRUE), confint=list(style="bars"))
```

**Figure 7.11** Effect plot for the race by gender interaction in the binomial GLMM fit to the Minneapolis police stops data.

## race\*gender effect plot



The general pattern we observed in the contingency table that we constructed disregarding neighborhoods generally holds up. Fixing the proportion black in a neighborhood to its average value, Native Americans and blacks are more likely than whites to be searched, with the difference between the races somewhat greater for males than for females; males are more likely than females to be searched, with the difference between the genders larger for blacks and Native Americans than for whites.

## 7.4 Complementary Reading

- Much of the material in this appendix is adapted from Fox (2016, chaps. 23 and 24).
- A very brief treatment of mixed models may be found in Weisberg (2014, [Section 7.4](#)).
- Snijders and Bosker (2012) and Raudenbush and Bryk (2002) are two accessible books that emphasize hierarchical linear and, to a lesser extent, generalized linear models.
- Gelman and Hill (2007) develop mixed models in the more general context of regression analysis; these authors also discuss Bayesian approaches to mixed models.
- Stroup (2013) presents a more formal and comprehensive development of generalized linear mixed models, treating other regression models, such as linear models, generalized linear models, and linear mixed-effects models, as special cases (and emphasizing SAS software for fitting these models).

# 8 Regression Diagnostics for Linear, Generalized Linear, and Mixed-Effects Models

John Fox and Sanford Weisberg

*Regression diagnostics* are methods for determining whether a fitted regression model adequately represents the data. In this chapter, we present methods for linear, generalized linear, and mixed-effects models, but many of the methods described here are also appropriate for other regression models. Because most of the methods for diagnosing problems in linear models fit by least squares extend naturally to generalized linear and mixed-effects models, we deal at greater length with linear-model diagnostics.

Regression diagnostics address the adequacy of a statistical model *after* it has been fit to the data. Careful thought about the problem at hand along with examination of the data (as in [Chapter 3](#)) *prior to* specifying a preliminary model often avoids problems at a later stage. Careful preliminary work doesn't, however, guarantee the adequacy of a regression model, and the practice of statistical modeling is therefore often one of iterative refinement. The methods described in this chapter can help you to reformulate a regression model to represent the data more accurately.

Linear models make strong and sometimes unrealistic assumptions about the structure of the data. When assumptions are violated, estimates and predictions can behave badly and may even completely misrepresent the data. The same is true of other parametric regression models. Regression diagnostics can reveal problems and often point the way toward solutions.

All of the methods discussed in this chapter either are available in standard R functions or are implemented in the **car** and **effects** packages. A few functions that were once in earlier versions of the **car** package are now a standard part of R.

An original and continuing goal of the **car** package is to make regression diagnostics readily available in R. It is our experience that diagnostic methods are much more likely to be used when they are *convenient*. For example, added-variable plots, described in [Section 8.3.3](#), are constructed by regressing a particular regressor and the response on all the other regressors in the model, computing the residuals from these auxiliary regressions, and plotting one set of residuals against the other. This is not hard to do in R, although the steps are somewhat more complicated when there are factors, interactions, or polynomial or regression-spline terms in the model. The `avPlots()` function in the **car**

package constructs all the added-variable plots for a linear or generalized linear model and adds such enhancements as a least-squares line and point identification.

- [Section 8.1](#) describes various kinds of residuals in linear models.
- [Section 8.2](#) introduces basic scatterplots of residuals, along with related plots that are used to assess the fit of a model to data.
- Subsequent sections are specialized to particular problems, describing methods for diagnosis and at least touching on possible remedies. [Section 8.3](#) introduces methods for detecting unusual data, including outliers, high-leverage points, and influential cases.
- [Section 8.4](#) returns to the topic of transforming the response and predictors (discussed previously in [Section 3.4](#)) to correct problems such as nonnormally distributed errors and nonlinearity.
- [Section 8.5](#) deals with nonconstant error variance.
- [Sections 8.6](#) and [8.7](#) respectively describe the extension of diagnostic methods to generalized linear models, such as logistic and Poisson regression, and to mixed-effects models.
- Diagnosing collinearity in regression models is the subject of [Section 8.8](#).
- Although extensive, this chapter isn't intended to be encyclopedic. We focus on methods that are most frequently used in practice and that we believe to be most generally useful. There are consequently several diagnostic functions in the `car` package that aren't covered in the chapter. [Section 8.9](#) briefly outlines these functions.

## 8.1 Residuals

Residuals of one sort or another are the basis of most diagnostic methods. Suppose that we specify and fit a linear model assuming constant error variance  $\sigma^2$ . The *ordinary residuals* are given by the differences between the responses and the fitted values,

(8.1)

$$e_i = y_i - \hat{y}_i, \quad i = 1, \dots, n$$

In ordinary-least-squares (OLS) regression, the residual sum of squares is equal to . If the regression model includes an intercept, then . The ordinary residuals are uncorrelated with the fitted values or indeed any linear combination of the regressors, including the regressors themselves, and so patterns in plots of ordinary residuals versus linear combinations of the regressors can occur only if one or more assumptions of the model are inappropriate.

If the regression model is correct, then the ordinary residuals are random variables with mean zero and with variance given by

$$(8.2) \quad \text{Var}(e_i) = \sigma^2(1 - h_i)$$

The quantity  $h^i$  is called a *leverage* or *hat-value*. In linear models with fixed predictors,  $h^i$  is a nonrandom value constrained to be between zero and 1, depending on the location of the predictors for a particular case relative to the other cases.<sup>1</sup> Let  $\mathbf{x}_i$  represent the vector of regressors for the  $i$ th of  $n$  cases.<sup>2</sup> Large values of  $h_i$  correspond to cases with relatively unusual  $\mathbf{x}_i$ -values, whereas small  $h_i$  corresponds to cases close to the center of the predictor data (see [Section 8.3.2](#)).

<sup>1</sup> In a model with an intercept, the minimum hat-value is  $1/n$ .

<sup>2</sup> We assume here the  $x_{0i} = 1$  is the constant regressor for a model with an intercept; if there is no intercept, then  $x_0$  is simply omitted.

Ordinary residuals for cases with large  $h_i$  have smaller variance. To correct for the nonconstant variance of the residuals, we can divide them by an estimate of their standard deviation. Letting  $\hat{\sigma}$  represent the estimate of  $\sigma^2$  in a model with an intercept and  $k$  other regressors, the *standardized residuals* are

$$(8.3) \quad e_{Si} = \frac{e_i}{\hat{\sigma}\sqrt{1 - h_i}}$$

While the  $e_{Si}$  have constant variance, they are no longer uncorrelated with the fitted values or linear combinations of the predictors, so using standardized residuals in plots is not an obvious improvement.

*Studentized residuals* are given by

$$(8.4) \quad e_{Ti} = \frac{e_i}{\hat{\sigma}_{(-i)}\sqrt{1 - h_i}}$$

where  $\hat{\sigma}_{(-i)}$  is the estimate of  $\sigma^2$  computed from the regression without the  $i$ th case. Like the standardized residuals, the Studentized residuals have constant variance. In addition, if the original errors are normally distributed, then  $e_{Ti}$  follows a  $t$ -distribution with  $n - k - 2$  degrees of freedom and can be used to test for outliers (see [Section 8.3](#)). One can show that in OLS linear regression,

$$(8.5) \quad \hat{\sigma}_{(-i)}^2 = \frac{\hat{\sigma}^2(n - k - 1 - e_{Si}^2)}{n - k - 2}$$

and so computing the Studentized residuals doesn't really require refitting the regression without the  $i$ th case.

If the model is fit by weighted-least-squares (WLS) regression with known positive weights  $w_i$ , then the ordinary residuals are replaced by the *Pearson residuals*,

$$(8.6) \quad e_{Pi} = \sqrt{w_i} e_i$$

The residual sum of squares is in WLS estimation. If we construe OLS regression to have implicit weights of  $w_i = 1$  for all  $i$ , then Equation 8.1 is simply a special case of Equation 8.6, and we will generally use the term *Pearson residuals* to cover both of these cases. The standardized and Studentized residuals are unaffected by weights because the weights cancel in the numerator and denominator of their formulas.

The generic R function `residuals()` can compute various kinds of residuals. The default for a linear model is to return the ordinary residuals even if weights are present. Setting the argument `type="pearson"` (with a lowercase "p") returns the Pearson residuals, which produces correctly weighted residuals if weights are present and the ordinary residuals if there are no weights. Pearson residuals are the default when `residuals()` is used with a generalized linear model. The functions `rstandard()` and `rstudent()` return the standardized and Studentized residuals, respectively. The function `hatvalues()` returns the hat-values.

## 8.2 Basic Diagnostic Plots

The **car** package includes a number of functions that produce plots of residuals and related quantities. The variety of plots reflects the fact that no one diagnostic graph is appropriate for all purposes.

### 8.2.1 Plotting Residuals

Plots of residuals versus fitted values and versus each of the regressors in turn are the most basic diagnostic graphs. If a linear model is correctly specified, then the Pearson residuals are independent of the fitted values and also of the regressors or the predictors on which they are based, and these graphs should be *null plots*, with no systematic features, in the sense that the conditional distribution of the residuals that are plotted on the vertical axis should not change with the fitted values, a regressor, or a predictor on the horizontal axis. The presence of systematic features generally implies a failure of one or more assumptions of the model. Of interest in these plots are nonlinear patterns, changes in variation across the graph, and isolated points.

Plotting residuals is useful for revealing problems but less useful for determining the exact nature of the problem. Consequently, we will present other diagnostic graphs to suggest improvements to a model.

Consider, for example, a modification of the model used in [Section 4.2.2](#) for the Canadian occupational-prestige data:

```
library("car")
Prestige$type <- factor(Prestige$type,
  levels=c("bc", "wc", "prof"))
prestige.mod.2 <- lm(prestige ~ education + income + type,
  data=Prestige)
brief(prestige.mod.2)
            (Intercept) education      income typewc typeprof
Estimate       -0.623      3.673 0.001013   -2.74      6.04
Std. Error      5.228      0.641 0.000221    2.51      3.87

Residual SD = 7.09 on 93 df, R-squared = 0.835
```

In [Section 4.2.2](#), we replaced the predictor income by the regressor log (income). Here we naïvely use income without transformation, in part to demonstrate what happens when a predictor needs transformation. For convenience, we also reorder the levels of the factor type.

The standard residual plots for this model are given by the `residualPlots()` function in the `car` package:

```
residualPlots(prestige.mod.2)
```

|            | Test stat | Pr(> Test stat ) |
|------------|-----------|------------------|
| education  | -0.68     | 0.4959           |
| income     | -2.89     | 0.0049           |
| type       |           |                  |
| Tukey test | -2.61     | 0.0090           |

This command produces the four graphs in [Figure 8.1](#) with the Pearson residuals on the vertical axis. The horizontal axis in the top row is for the numeric regressors education and income. The first graph in the second row shows boxplots of the residuals for the various levels of the factor type. The final graph has the fitted values on the horizontal axis. The broken line in each panel is the horizontal line through  $e_P = 0$ ; as explained below, the solid line is a quadratic fit to the points in the plot.

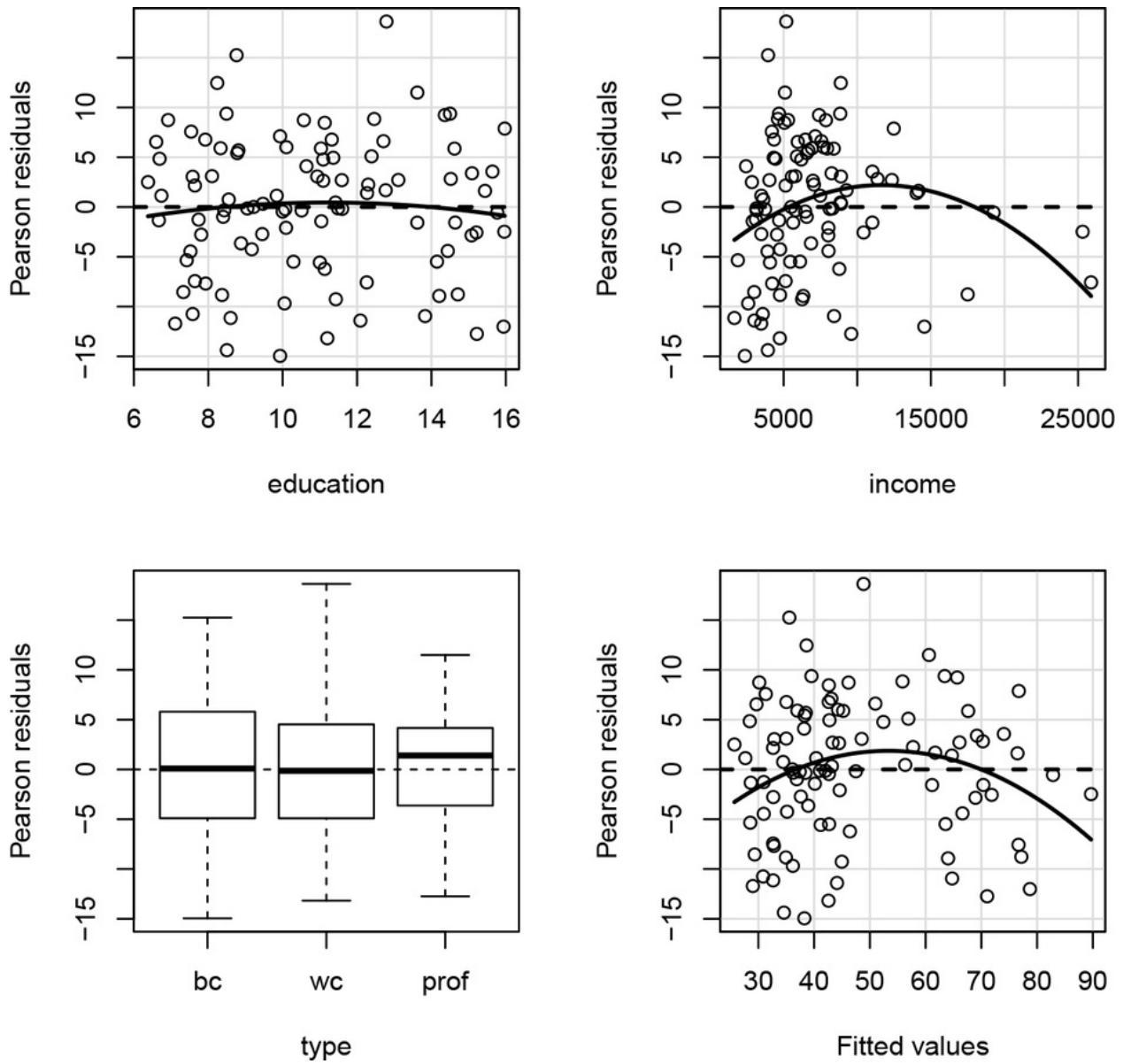
The most common diagnostic graph in linear regression is the plot of residuals versus the fitted values, shown at the bottom right of [Figure 8.1](#). The plot has a curved general pattern, suggesting that the model we fit is not adequate to describe the data. The plot of residuals versus education at the top left, however, resembles a null plot, in which no particular pattern is apparent. A null plot is consistent with an adequate model, but as is the case here, one null plot is insufficient to provide evidence of an adequate model, and indeed one nonnull plot is enough to suggest that the specified model does not match the data. The plot of residuals versus income at the top right is also curved, as might have been anticipated in light of our preliminary examination of the data in [Section 3.3.2](#). The residual plot for a factor like type, at the bottom left, is a set of boxplots of the residuals at the various levels of the factor. In a null plot, the boxes should all have about the same center and inter-quartile distance, as is more or less the case here.

To help examine these residual plots, a *lack-of-fit test* is computed for each numeric regressor and a curve is added to the corresponding graph. The lack-of-

fit test for education, for example, is the  $t$ -test for the regressor ( $\text{education}^2$ )<sup>2</sup> added to the model, for which the corresponding  $p$ -value rounds to .50, indicating no lack of fit of this type. For income, the lack-of-fit test produces the  $p$ -value .005, clearly confirming the nonlinear pattern visible in the graph. The lines shown on the plot are the fitted quadratic regressions of the Pearson residuals on the numeric regressors.

For the plot of residuals versus fitted values, the test, called *Tukey's test for non-additivity* (Tukey, 1949), is obtained by adding the squares of the fitted values to the model and refitting. The  $p$ -value for Tukey's test is obtained by comparing the test statistic to the standard-normal distribution. The test confirms the visible impression of curvature in the residual plot, further reinforcing the conclusion that the fitted model is not adequate.

**Figure 8.1** Basic residual plots for the regression of prestige on education, income, and type in the Prestige data set.



The `residualPlots()` function shares many arguments with other graphics functions in the **car** package; see `help("residualPlots")` for details. All arguments beyond the first are optional. The `id` argument controls point identification; for example, setting `id=list(n=3)` would automatically identify the three cases with the largest absolute residuals (see [Section 3.5](#)). There are additional arguments to control the layout of the plots and the type of residual plotted; setting `type="rstudent"`, for example, would plot Studentized residuals rather than Pearson residuals. Setting `smooth=TRUE`, `quadratic=FALSE` would display a loess smooth rather than a quadratic curve on each plot, although the test statistics always correspond to fitting quadratics.

If you want only the plot of residuals against fitted values, you can use

***residualPlots (prestige.mod.2, ~ 1, fitted=TRUE)***

whereas the plot against education only can be obtained with

***residualPlots (prestige.mod.2, ~ education, fitted=FALSE)***

The second argument to residualPlots (), and to other functions in the **car** package that can produce graphs with several panels, is a *one-sided formula* that specifies the regressors against which to plot residuals. The formula `~ .` is the default, to plot against *all* the regressors; `~ 1` plots against *none* of the regressors and in the current context produces a plot against fitted values only; `~ . - income` plots against all regressors but income. Because the fitted values are not part of the formula that defined the model, there is a separate fitted argument, with default TRUE to include a plot of residuals against fitted values and set to FALSE to exclude it.

## 8.2.2 Marginal-Model Plots

A variation on the basic residual plot is the *marginal-model plot*, proposed by R. D. Cook and Weisberg (1997) and implemented in the marginalModelPlots () function:

***marginalModelPlots (prestige.mod.2)***

The plots shown in [Figure 8.2](#) all have the response variable, in this case prestige, on the vertical axis, while the horizontal axis is given in turn by each of the numeric regressors in the model and the fitted values. No plot is produced for the factor predictor type. The plots of the response versus individual regressors display the conditional distribution of the response given each regressor, ignoring the other regressors; these are *marginal* plots in the sense that they show the marginal relationship between the response and each regressor. The plot versus fitted values is a little different, in that it displays the conditional

distribution of the response given the fit of the model.

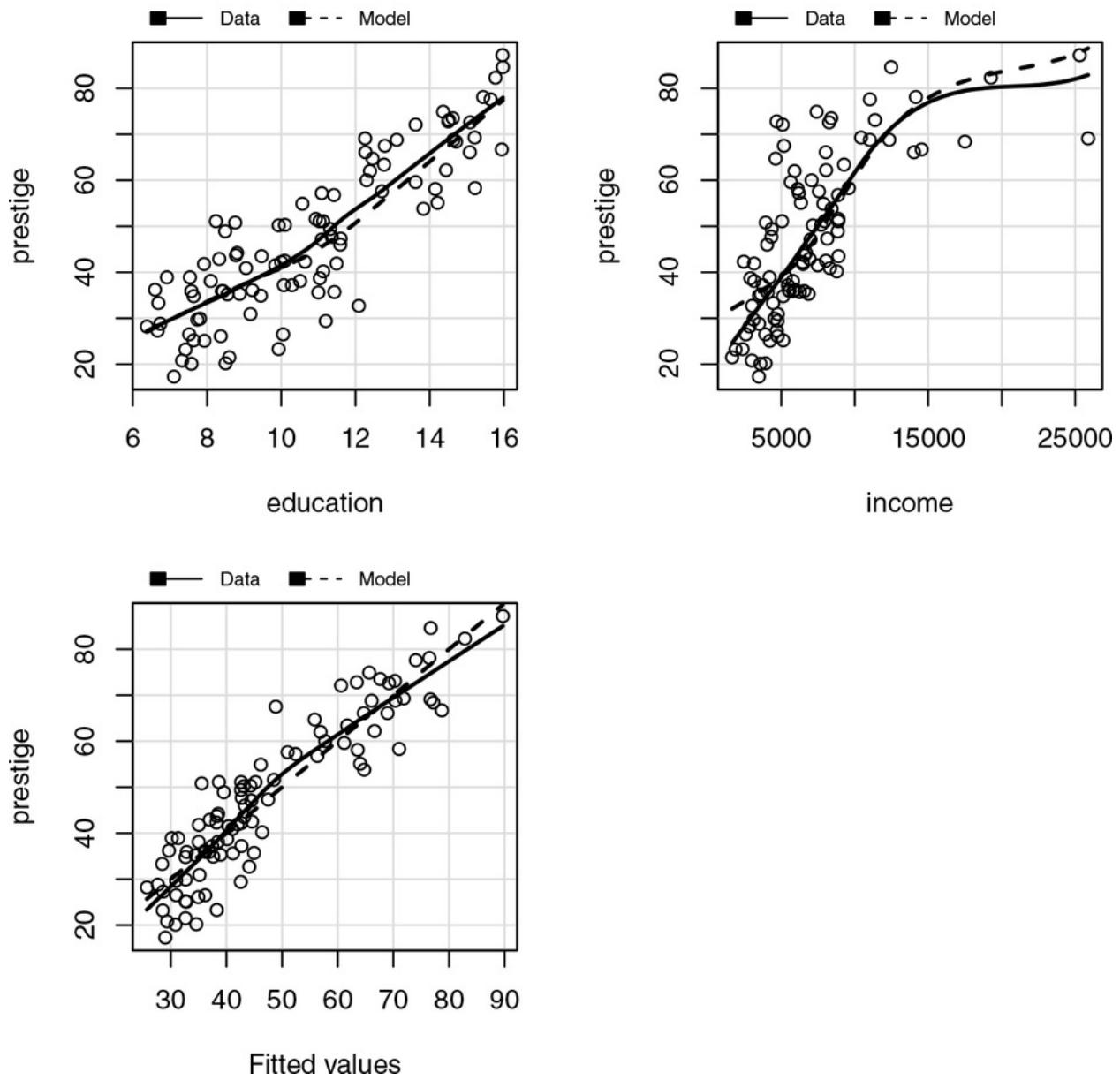
We can estimate a regression function for each of the marginal plots by fitting a smoother to the points in the plot. The `marginalModelPlots()` function uses a loess smooth, as shown by the solid line on the plot.

Now imagine a second graph that replaces the vertical axis by the fitted values from the model. If the model is appropriate for the data, then, under fairly mild conditions, the smooth fit to this second plot should also estimate the conditional expectation of the response given the regressor on the horizontal axis. The second smooth is also drawn on the marginal-model plot, as a dashed line. If the model fits the data well, then the two smooths should match on each of the marginal-model plots; if any pair of smooths fails to match, then we have evidence that the model does not fit the data well.

An interesting feature of the marginal-model plots in [Figure 8.2](#) is that even though the model that we fit to the `Prestige` data specifies linear *partial* relationships between prestige and each of education and income, it is able to reproduce nonlinear *marginal* relationships for these two regressors. Indeed, the model, as represented by the dashed lines, does a fairly good job of matching the marginal relationships represented by the solid lines, although the systematic failures discovered in the residual plots are discernable here as well.

**Figure 8.2** Marginal-model plots for the regression of prestige on education, income, and type in the `Prestige` data set.

## Marginal-Model Plots



Marginal-model plots can be used with any fitting or modeling method that produces fitted values, and so they can be applied to some problems where the definition of residuals is unclear. In particular, marginal-model plots work well with generalized linear models.

The `marginalModelPlots()` function has an `SD` argument, which, if set to `TRUE`, adds estimated standard-deviation lines to the graph. The plots can therefore be used to check both the regression function, as illustrated here, and assumptions about variance. Other arguments to the `marginalModelPlots()` function are

similar to those for `residualPlots()`.

### 8.2.3 Added-Variable Plots

The marginal-model plots of the last section display the *marginal* relationships between the response and each regressor, *ignoring* other regressors in the model. In contrast, *added-variable plots*, also called *partial-regression plots*, display the *partial* relationship between the response and a regressor, *adjusted for* all the other regressors.

Suppose that we have a regression problem with response  $y$  and regressors  $x_1, \dots, x_k$ .<sup>3</sup> To draw the added-variable plot for one of the regressors, say the first,  $x_1$ , conduct the following two auxiliary regressions:

1. Regress  $y$  on all the regressors excluding  $x_1$ . The residuals from this regression are *the part of  $y$  that is not “explained” by all the regressors except for  $x_1$* .
2. Regress  $x_1$  on the other regressors and again obtain residuals. These residuals represent *the part of  $x_1$  that is not explained by the other regressors*; put another way, the residuals are the part of  $x_1$  that remains when we condition on the other regressors.

<sup>3</sup> Although it is not usually of interest, when there is an intercept in the model, it is also possible to construct an added-variable plot for the constant regressor,  $x_0$ , which is equal to 1 for every case.

The added-variable plot for  $x^1$  is simply a scatterplot, with the residuals from Step 1 on the vertical axis and the residuals from Step 2 on the horizontal axis.

The `avPlots()` function in the `car` package works both for linear and generalized linear models. It has arguments for controlling which plots are drawn, point labeling, and plot layout, and these arguments are the same as for the `residualPlots()` function described in [Section 8.2.1](#).

Added-variable plots for the Canadian occupational-prestige regression (in [Figure 8.3](#)) are produced by the following command:

### ***avPlots (prestige.mod.2, id=list (n=2, cex=0.6))***

The argument `id=list (n=2, cex=0.6)` identifies up to four points in each graph: the two that are furthest from the mean on the horizontal axis and the two with the largest absolute residuals from the fitted line. Because the case labels in the Prestige data set are long, we used `cex=0.6` to reduce the size of the printed labels to 60% of their default size.

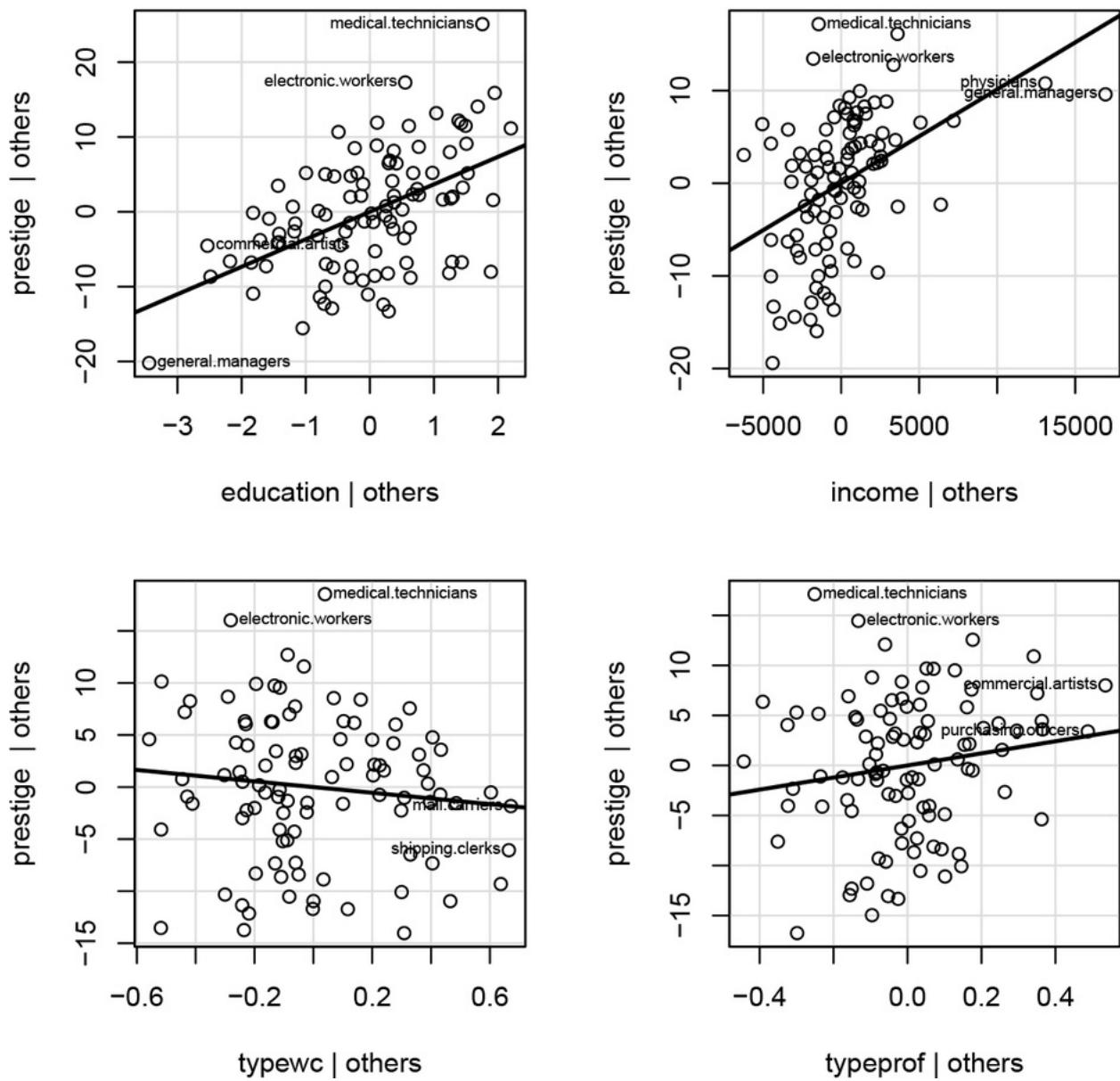
The added-variable plot has several interesting and useful properties:

- The least-squares line on the added-variable plot for the regressor  $x_j$  has the same slope  $b_j$  as  $x_j$  in the full regression. Thus, for example, the slope in the added-variable plot for education is  $b_1 = 3.67$ , and the slope in the addedvariable plot for income is  $b_2 = 0.00101$ .<sup>4</sup>
- The residuals from the least-squares line in the added-variable plot are the same as the residuals  $e_i$  from the regression of the response on *all* the regressors.
- Because positions on the horizontal axis of the added-variable plot show values of  $x_j$  conditional on the other regressors, points far to the left or right represent cases for which the value of  $x_j$  is unusual given the values of the other regressors. The variation of the variable on the horizontal axis is the conditional variation of  $x_j$ , and the added-variable plot therefore allows us to visualize the precision of estimation of  $b_j$ , along with the leverage of each case on the regression coefficient (see [Section 8.3.2](#) for a discussion of leverage in regression).
- For factors, an added-variable plot is produced for each of the contrasts that are used to define the factor, and thus if we change the way that contrasts are coded for a factor, the corresponding added-variable plots will change as well.

<sup>4</sup> Income is in dollars per year, so the slope for income is in prestige points per dollar. Units are always important in interpreting slope estimates: In the current example, an additional dollar of annual income is a very small increment.

**Figure 8.3** Added-variable plots for the regression of prestige on education, income, and type in the Prestige data set.

## Added-Variable Plots



The added-variable plot allows us to visualize the effect of each regressor after adjusting for all the other regressors in the model, effectively reducing the  $(k + 1)$ -dimensional regression problem to a sequence of 2D graphs.

In [Figure 8.3](#), the plot for income has a positive slope, but the slope appears to be influenced by two high-income occupations (physicians and general managers) that pull down the regression line at the right. There don't seem to be any particularly noteworthy points in the added-variable plots for the other regressors.

Although added-variable plots are useful for studying the impact of cases on the regression coefficients (see [Section 8.3.3](#)), they can prove misleading for diagnosing other sorts of problems, such as nonlinearity. A further disadvantage of the added-variable plot is that the variables on both axes are sets of residuals, and so neither the response nor the regressors is displayed directly.

Sall (1990) and R. D. Cook and Weisberg (1991) generalize added-variable plots to terms with more than one degree of freedom, such as a factor or polynomial regressors. Following Sall, we call these graphs *leverage plots*. For terms with 1 degree of freedom, leverage plots are very similar to added-variable plots, except that the slope in the plot is always equal to 1, not to the corresponding regression coefficient. Although leverage plots can be misleading in certain circumstances,<sup>5</sup> they can be useful for locating groups of cases that are jointly high leverage or influential. Leverage, influence, and related ideas are explored in [Section 8.3](#). There is a `leveragePlots()` function in the **car** package, which works only for linear models.

<sup>5</sup> For example, if a particular case causes one dummy-regressor coefficient to get larger and another smaller, these changes can cancel each other out in the leverage plot for the corresponding factor, even though a different pattern of results for the factor would be produced by removing the case.

## 8.2.4 Marginal-Conditional Plots

Marginal-conditional plots are a pedagogical tool to help understand the distinction between the unconditional and conditional relationships of the response variable to a specific regressor. The unconditional relationship is visualized in a scatterplot of the response versus the regressor. The conditional relationship is visualized by an added-variable plot. The `mcPlot()` and `mcPlots()` functions in the **car** package compare these two graphs either in the same scale or superimposed on the same plot.

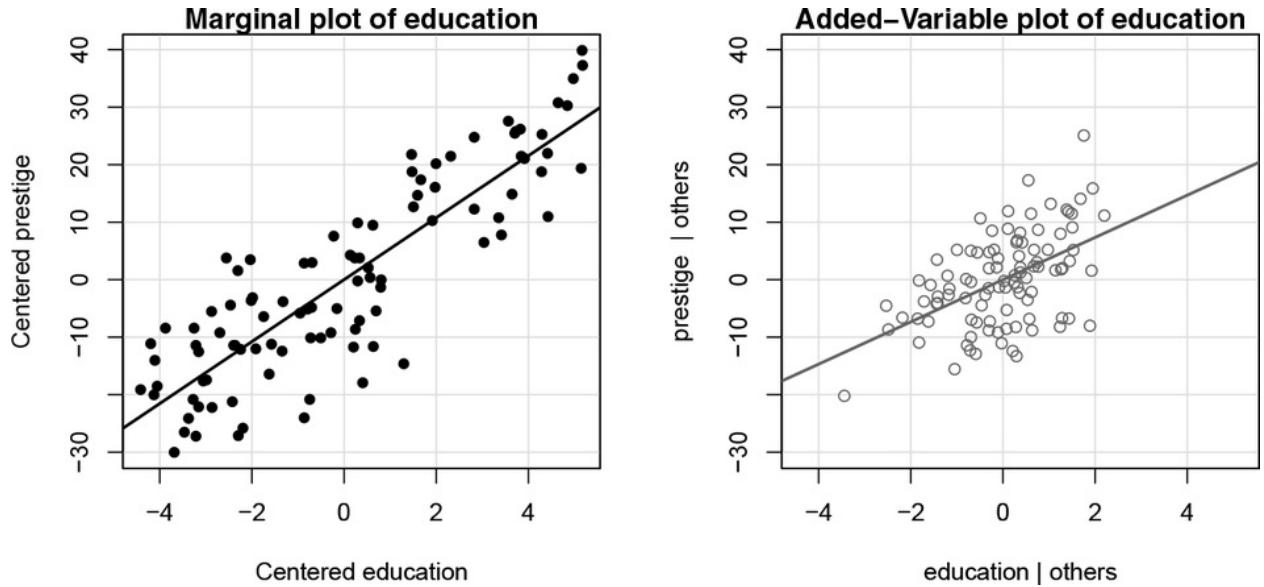
[Figure 8.4](#) is the marginal-conditional plot for education in the model for the Canadian occupational-prestige data used in the last few sections, drawn as two separate panels (by setting `overlaid=FALSE`):

***mcPlots(prestige.mod.2, ~ education, overlaid=FALSE)***

The scatterplot on the left displays the response prestige on the vertical axis and the regressor education on the horizontal axis. The variables on both axes are centered by subtracting their respective sample means. Centering changes the labels on the tick marks to make the scatterplot comparable to the added-variable plot, but it does not change the shape of the scatterplot. Marginally, prestige increases linearly with education. The line shown on the graph has slope given by the OLS regression of prestige on education alone. As is typical of functions in the **car** package, `mcPlot()` draws a marginal-conditional plot for one regressor, and `mcPlots()` draws one or more plots. See help ("`mcPlot`") for descriptions of available options.

The graph on the right of [Figure 8.4](#) displays the conditional added-variable plot for education after adjusting for the other regressors in the model. The added-variable plot has the same size and scaling as the marginal plot. Both plots display one point for each case in the data. If both the response and the regressor were independent of the other regressors, then these two plots would be identical. If the response were perfectly correlated with the other regressors, then the points in the second plot would all have vertical coordinates of zero, and if the regressor were perfectly correlated with the other regressors, then all the horizontal coordinates in the second plot would equal zero. In this example, we see that conditioning education on the other regressors accounts for most of the variation in education (the  $R^2$  for the regression of education on income and type is 0.83), as reflected by much smaller variation in horizontal coordinates in the conditional plot than in the marginal plot. Similarly, much of the variation in the response in the marginal plot is accounted for by the other regressors, reflected in the relatively small variation in the response in the conditional plot. The regression line in the conditional plot has slope equal to the education regression coefficient from the full model, `prestige.mod.2` (page 388).

**Figure 8.4** Marginal-conditional plots for education in the regression of prestige on education, income, and type of occupation in the Prestige data set. The panel on the left is the centered marginal scatterplot for prestige versus education; the panel on the right is the conditional added-variable plot for the two variables in the model.



## 8.3 Unusual Data

Unusual data can wreak havoc with least-squares estimates and may prove interesting in their own right. Unusual data in regression include outliers, high-leverage points, and influential cases.

### 8.3.1 Outliers and Studentized Residuals

*Regression outliers* are  $y$ -values that are unusual *conditional on* the values of the predictors. An illuminating route to search for outliers is via the *mean-shift*

$$\text{outlier model, } y = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k + \gamma d_i + \varepsilon$$

where  $d^i$  is a dummy regressor coded 1 for case  $i$  and zero for all others. If  $\gamma \neq 0$ , then the conditional expectation of the  $i$ th case has the same dependence on  $x_1, \dots, x_k$  as the other cases, but its intercept is shifted from  $\beta_0$  to  $\beta_0 + \gamma$ . The  $t$ -statistic for testing the null hypothesis  $H_0: \gamma = 0$  against a two-sided alternative has  $n - k - 2$  degrees of freedom if the errors are normally distributed and is the appropriate test for a single mean-shift outlier at case  $i$ . Remarkably, this  $t$ -statistic turns out to be identical to the  $i$ th Studentized residual,  $e_{Ti}$  (Equation 8.4, page 387), and so we can get the test statistics for the  $n$  different null hypotheses,  $H_{0i}$ : case  $i$  is not a mean-shift outlier,  $i = 1, \dots, n$ , at minimal computational cost.

Our attention is generally drawn to the largest absolute Studentized residual, and this presents a problem: Even if the Studentized residuals were independent, which they are not, there would be an issue of simultaneous inference entailed by picking the largest of  $n$  test statistics. The dependence of the Studentized residuals complicates the issue. We can deal with this problem (1) by a *Bonferroni adjustment* of the  $p$ -value for the largest absolute Studentized residual, multiplying the usual two-tail  $p$  by the sample size  $n$ , or (2) by constructing a quantile-comparison plot of the Studentized residuals with a confidence envelope that takes their dependence into account.

To illustrate, we reconsider Duncan's occupational-prestige data (introduced in [Section 1.5](#)), regressing prestige on occupational income and education levels:

```
mod.duncan <- lm (prestige ~ income + education, data=Duncan)
```

The generic `qqPlot()` function in the **car** package has a method for linear models, plotting Studentized residuals against the corresponding quantiles of  $t(n - k - 2)$ . By default, `qqPlot()` generates a 95% pointwise confidence envelope for the Studentized residuals, using a parametric version of the bootstrap, as suggested by Atkinson (1985):<sup>6</sup>

```
qqPlot(mod.duncan, id=list(n=3))
```

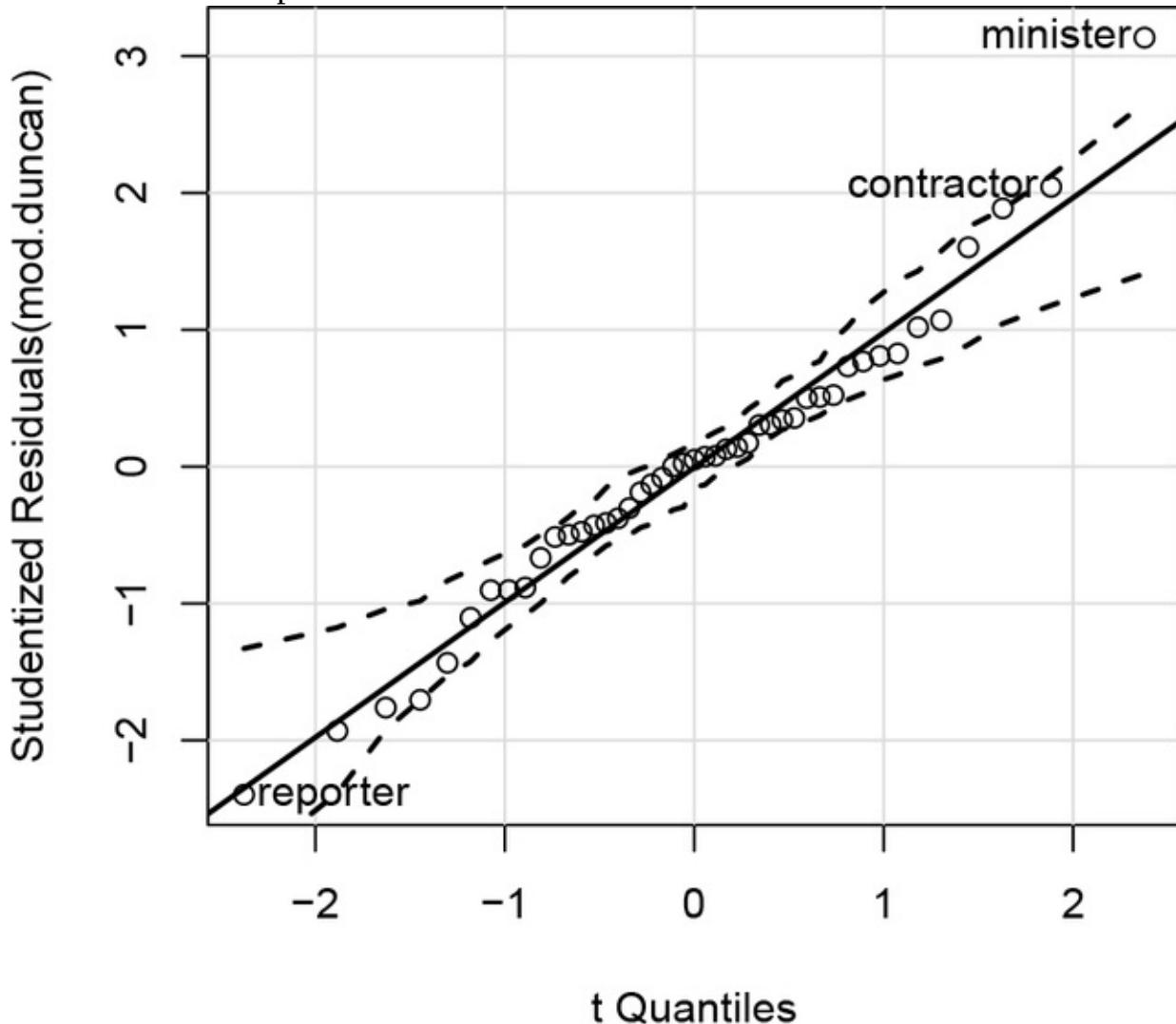
|          |          |            |
|----------|----------|------------|
| minister | reporter | contractor |
| 6        | 9        | 17         |

<sup>6</sup> Bootstrap methods in R are described in [Section 5.1.3](#) and in an online appendix to the book.

The resulting plot is shown in [Figure 8.5](#). Setting the argument `id=list(n=3)` to the `qqPlot()` function returns the names and indices of the three cases with the largest absolute Studentized residuals and identifies these points in the graph (see [Section 3.5](#) on point identification); of these points, only minister strays slightly outside of the confidence envelope. If you repeat this command, your plot may look a little different from ours because the envelope is computed by simulation. The distribution of the Studentized residuals looks heavy-tailed

compared to the reference  $t$ -distribution, and perhaps a method of robust regression would be more appropriate for these data.<sup>7</sup>

**Figure 8.5** Quantile-comparison plot of Studentized residuals from Duncan's occupational-prestige regression, showing the pointwise 95% simulated confidence envelope.



<sup>7</sup> R functions for robust and resistant regression are described in an online appendix to the *R Companion*.

The `outlierTest()` function in the `car` package locates the largest Studentized residual in absolute value and computes the Bonferroni-corrected  $t$ -test:

### **outlierTest (mod. duncan)**

```
No Studentized residuals with Bonferroni p < 0.05
Largest |rstudent|:
      rstudent unadjusted p-value Bonferroni p
minister    3.1345          0.0031772      0.14297
```

The Bonferroni-adjusted  $p$ -value is fairly large, .14, and so we conclude that it isn't very surprising that the biggest Studentized residual in a sample of size  $n = 45$  would be as great as 3.135. The somewhat different conclusions suggested by the QQ-plot and the Bonferroni outlier test have a simple explanation: The confidence envelope in the QQ-plot is based on *pointwise* 95% confidence intervals while the outlier test adjusts for *simultaneous* inference.

### **8.3.2 Leverage: Hat-Values**

Cases that are relatively far from the center of the predictor space, taking account of the correlational pattern among the predictors, have potentially greater influence on the least-squares regression coefficients; such points are said to have *high leverage*. The most common measures of leverage are the  $h_i$  or *hat-values*.<sup>8</sup> The  $h_i$  are bounded between zero and 1 (in models with an intercept, they are bounded between  $1/n$  and 1); their sum,  $\sum h_i$ , is always equal to the number of coefficients in the model, including the intercept, and so in a model with an intercept, the average hat-value is  $1/n$ . Problems in which there are a few very large  $h_i$  can be troublesome because large-sample normality of some linear combinations of the predictors is likely to fail, and high-leverage cases may exert undue influence on the results (see below).

<sup>8</sup>\* The name “hat-values” comes from the relationship between the observed vector of responses (i.e., the  $y_s$ ) and the fitted values (i.e., the  $\hat{y}$ s or “y-hats”). The vector of fitted values is given by  $\hat{y} = \mathbf{H}\beta$ , called the *hat-matrix*, projects  $y$  into the subspace spanned by the columns of the model matrix  $\mathbf{X}$ . Because  $\mathbf{H} = \mathbf{H}'\mathbf{H}$ , the hat-values  $h_i$  are simply the diagonal entries of the hat-matrix.

The `hatvalues()` function works for both linear and generalized linear models. One way of examining the hat-values and other individual-case diagnostic statistics is to construct *index plots*, graphing the statistics against the

corresponding case indices.

For example, the following command uses the **car** function `influenceIndexPlot()` to produce [Figure 8.6](#), which includes index plots of Studentized residuals, the corresponding Bonferroni  $p$ -values for outlier testing, the hat-values, and Cook's distances (discussed in the next section) for Duncan's occupational-prestige regression:

***influenceIndexPlot (mod.duncan, id=list (n=3))***

The occupations railroad engineer (RR.engineer), conductor, and minister stand out from the rest in the plot of hat-values, indicating that their predictor values are unusual relative to the other occupations. In the plot of  $p$ -values for the outlier tests, cases for which the Bonferroni bound is bigger than 1 are set equal to 1, and here only one case (minister) has a Bonferroni  $p$ -value much less than 1.

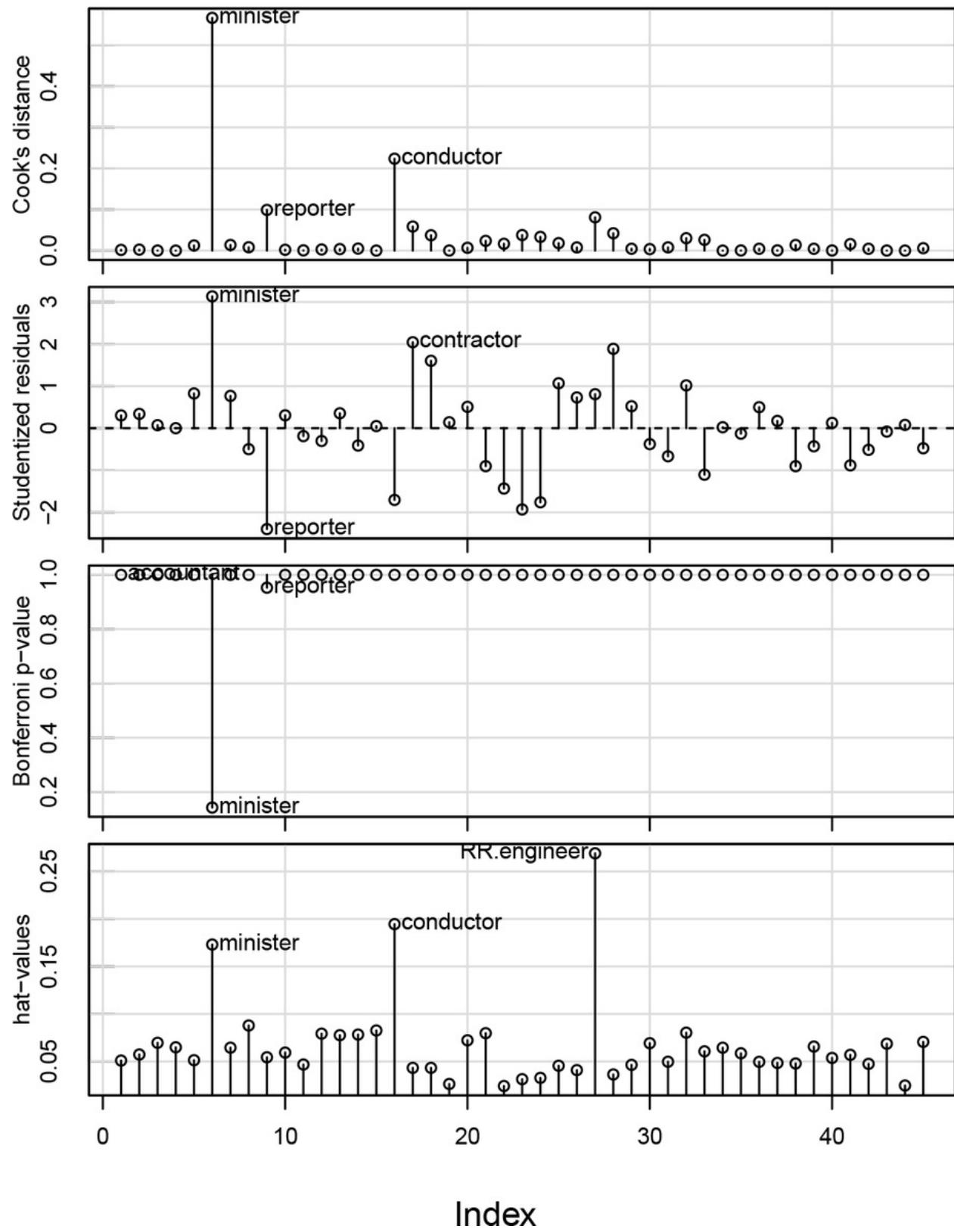
### 8.3.3 Influence Measures

A case that is both outlying and has high leverage exerts *influence* on the regression coefficients, in the sense that if the case is removed, the coefficients change considerably. As usual, let  $\mathbf{b}$  be the estimated value of the coefficient vector  $\beta$  and as new notation define  $\mathbf{b}(-i)$  to be the estimate of  $\beta$  computed without the  $i$ th case.<sup>9</sup> Then the difference  $\mathbf{b}(-i) - \mathbf{b}$  directly measures the influence of the  $i$ th case on the estimate of  $\beta$ . If this difference is small, then the influence of case  $i$  is small, while if the difference is large, then its influence is large.

<sup>9</sup> If vector notation is unfamiliar, simply think of  $\mathbf{b}$  as the collection of estimated regression coefficients,  $b_0, b_1, \dots, b_k$ .

**Figure 8.6** Index plots of diagnostic statistics for Duncan's occupational-prestige regression, indentifying the three most extreme cases in each plot.

## Diagnostic Plots



## Cook's Distance

It is convenient to summarize the size of the difference  $\mathbf{b}_{(-i)} - \mathbf{b}$  by a single number, and this can be done in several ways. The most common summary measure of influence is *Cook's distance* (R. D. Cook, 1977),  $D_i$ , which is just a weighted sum of squares of the differences between the individual elements of the coefficient vectors.<sup>10</sup> Interestingly, Cook's distance can be computed from diagnostic statistics that we have already encountered,

$$D_i = \frac{e_{Si}^2}{k+1} \times \frac{h_i}{1-h_i}$$

$$D_i = \frac{(\mathbf{b}_{(-i)} - \mathbf{b})' \mathbf{X}' \mathbf{X} (\mathbf{b}_{(-i)} - \mathbf{b})}{(k+1)\hat{\sigma}^2}$$

<sup>10\*</sup> In matrix notation,

where  $e_{Si}$  is the squared standardized residual (Equation 8.3 on page 387) and  $h_i$  is the hat-value for case  $i$ . The first factor may be thought of as a measure of outlyingness and the second as a measure of leverage. Cases for which  $D_i$  is largest are potentially influential cases. If any noteworthy  $D_i$  are apparent, then a prudent approach is to remove the corresponding cases temporarily from the data, refit the regression, and see how the results change. Because an influential case can affect the fit of the model at other cases, it is best to remove cases one at a time, refitting the model at each step and reexamining the resulting Cook's distances.

The generic function `cooks.distance()` has methods for linear and generalized linear models. Cook's distances are also plotted, along with Studentized residuals and hat-values, by the `influenceIndexPlot()` function, as illustrated for Duncan's regression in [Figure 8.6](#). The occupation minister is the most influential according to Cook's distance, and we therefore see what happens when we delete this case and refit the model:

```
mod.duncan.2 <- update(mod.duncan,
subset= rownames(Duncan) != "minister")
compareCoefs(mod.duncan, mod.duncan.2)
```

Calls:

```
1: lm(formula = prestige ~ income + education, data =
Duncan)
2: lm(formula = prestige ~ income + education, data =
Duncan, subset = rownames(Duncan) != "minister")
```

|             | Model 1 | Model 2 |
|-------------|---------|---------|
| (Intercept) | -6.06   | -6.63   |
| SE          | 4.27    | 3.89    |
| income      | 0.599   | 0.732   |
| SE          | 0.120   | 0.117   |
| education   | 0.5458  | 0.4330  |
| SE          | 0.0983  | 0.0963  |

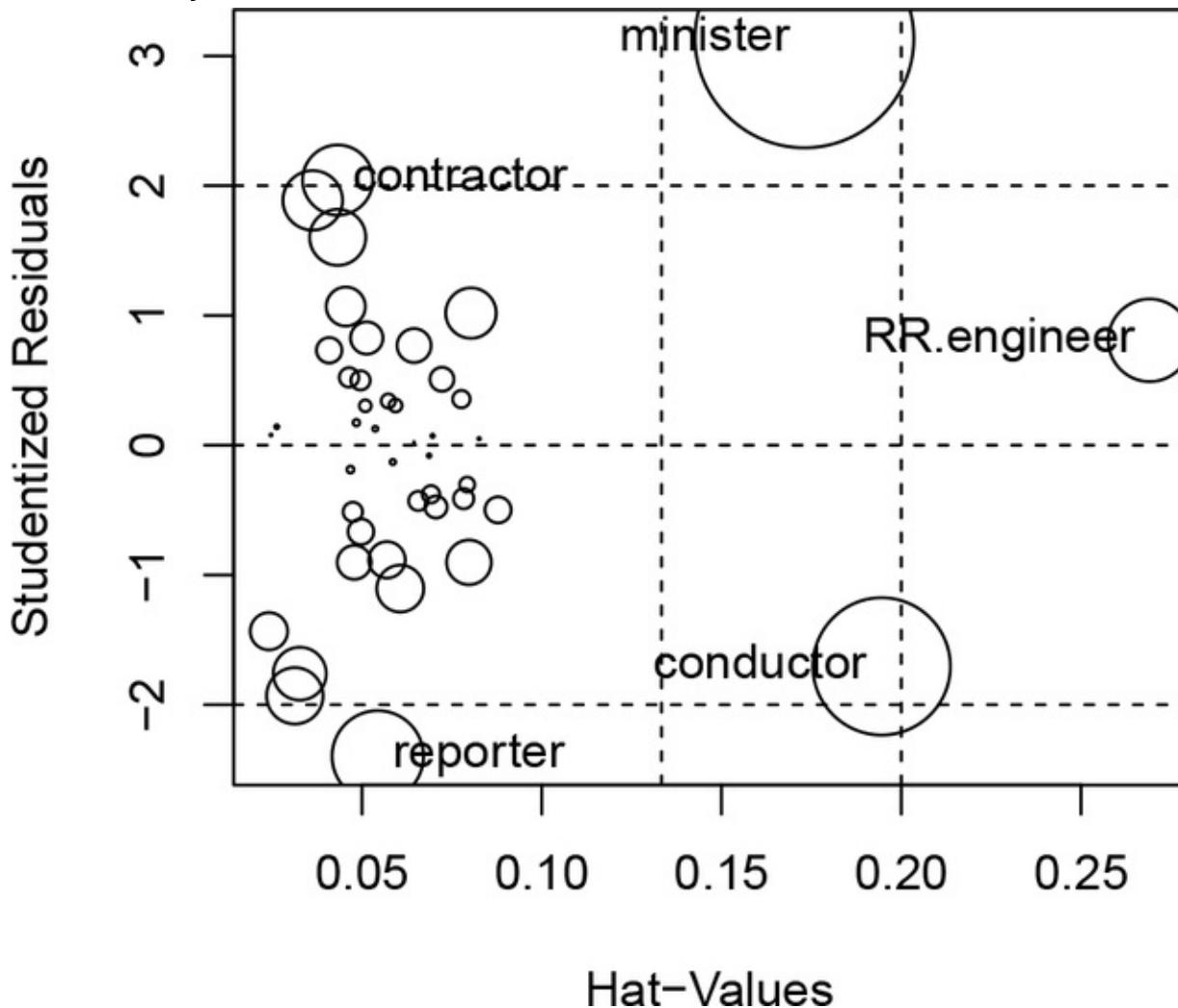
Removing minister increases the coefficient for income by about 20% and decreases the coefficient for education by about the same amount. Standard errors are much less affected. In other problems, removing a case can change “statistically significant” results to “nonsignificant” ones and vice-versa.

The influencePlot () function in the **car** package provides an alternative to index plots of diagnostic statistics:

```
influencePlot(mod.duncan, id=list(n=3))
```

|             | StudRes  | Hat      | CookD    |
|-------------|----------|----------|----------|
| minister    | 3.13452  | 0.173058 | 0.566380 |
| reporter    | -2.39702 | 0.054394 | 0.098985 |
| conductor   | -1.70403 | 0.194542 | 0.223641 |
| contractor  | 2.04380  | 0.043255 | 0.058523 |
| RR.engineer | 0.80892  | 0.269090 | 0.080968 |

**Figure 8.7** Plot of hat-values, Studentized residuals, and Cook's distances for Duncan's occupational-prestige regression. The size of the circles is proportional to Cook's  $D_i$ .



This command produces a *bubble plot*, shown in [Figure 8.7](#), combining the display of Studentized residuals, hat-values, and Cook's distances, with the areas of the circles proportional to Cook's  $D_i$ . As usual, the id argument is used to label points. In this case, the n=3 points with the largest hat-values, Cook's distance, or absolute Studentized residuals will be flagged, so more than three points in all are labeled.

We invite the reader to continue the analysis by examining influence diagnostics for Duncan's regression after the case minister has been removed.

## Influence Separately for Each Coefficient

Rather than summarizing influence by looking at all coefficients simultaneously, we could create  $k + 1$  measures of influence by looking at individual differences  $\text{dfbeta}_{ij} = b_{(-i)j} - b_j$  for  $j = 0, \dots, k$

where  $b_j$  is the coefficient computed using all of the data, and  $b_{(-i)j}$  is the same coefficient computed with case  $i$  omitted. As with  $D_i$ , computation of the  $\text{dfbeta}_{ij}$  can be accomplished efficiently without having to refit the model. The  $\text{dfbeta}_{ij}$  are expressed in the metric (units of measurement) of the coefficient  $b_j$ . A standardized version,  $\text{dfbetas}_{ij}$ , divides  $\text{dfbeta}_{ij}$  by an estimate of the standard error of  $b_j$  computed with case  $i$  removed.

The `dfbeta()` function in R takes a linear-model or generalized-linear-model object as its argument and returns all of the  $\text{dfbeta}_{ij}$ ; similarly, `dfbetas()` computes the  $\text{dfbetas}_{ij}$ . For example, for Duncan's regression:

```

dfbs.duncan <- dfbetas(mod.duncan)
head(dfbs.duncan) # first few rows

      (Intercept)      income   education
accountant -2.2534e-02 6.6621e-04 0.03594387
pilot       -2.5435e-02 5.0877e-02 -0.00811827
architect   -9.1867e-03 6.4837e-03 0.00561927
author      -4.7204e-05 -6.0177e-05 0.00013975
chemist     -6.5817e-02 1.7005e-02 0.08677706
minister    1.4494e-01 -1.2209e+00 1.26301904

```

We could examine each column of the dfbetas matrix separately (e.g., via an index plot), but because we are not really interested here in influence on the regression intercept, and because there are just two slope coefficients, we instead plot influence on the income coefficient against influence on the education coefficient ([Figure 8.8](#)):

```

plot(dfbs.duncan[ , c("income", "education")]) # for b1 and b2
showLabels(dfbs.duncan[ , "income"],
           dfbs.duncan[ , "education"],
           labels=rownames(Duncan), method="identify")
# remember to exit from point identification mode

```

The negative relationship between the  $\text{dfbetas}_{ij}$  values for the two predictors reflects the *positive* correlation of the predictors themselves. Two pairs of values stand out: Consistent with our earlier remarks, the cases minister and conductor make the income coefficient smaller and the education coefficient larger. We also identify the occupation RR.engineer in the plot.

## Added-Variable Plots as Influence Diagnostics

The added-variable plots introduced in [Section 8.2.3](#) are a useful diagnostic for finding potentially jointly influential points, which correspond to sets of points that are out of line with the rest of the data and are at the extreme left or right of the horizontal axis. When two or more such points act in concert to affect the fitted regression surface, they may not be revealed by individual-case statistics

such as Cook's  $D_i$ . [Figure 8.9](#), for example, shows the added-variable plots for income and education in Duncan's regression:

```
avPlots (mod.duncan, id=list (n=3, method="mahal"))
```

The argument `id=list (n=3, method="mahal")` serves to identify the three points in each panel with the largest Mahalanobis distances from the center of all the points.<sup>11</sup> The cases minister, conductor, and RR.engineer (railroad engineer) have high leverage on both coefficients. The cases minister and conductor also work together to decrease the income slope and increase the education slope; RR.engineer, on the other hand, is more in line with the rest of the data.

Removing *both* minister and conductor changes the regression coefficients much more so than deleting minister alone:

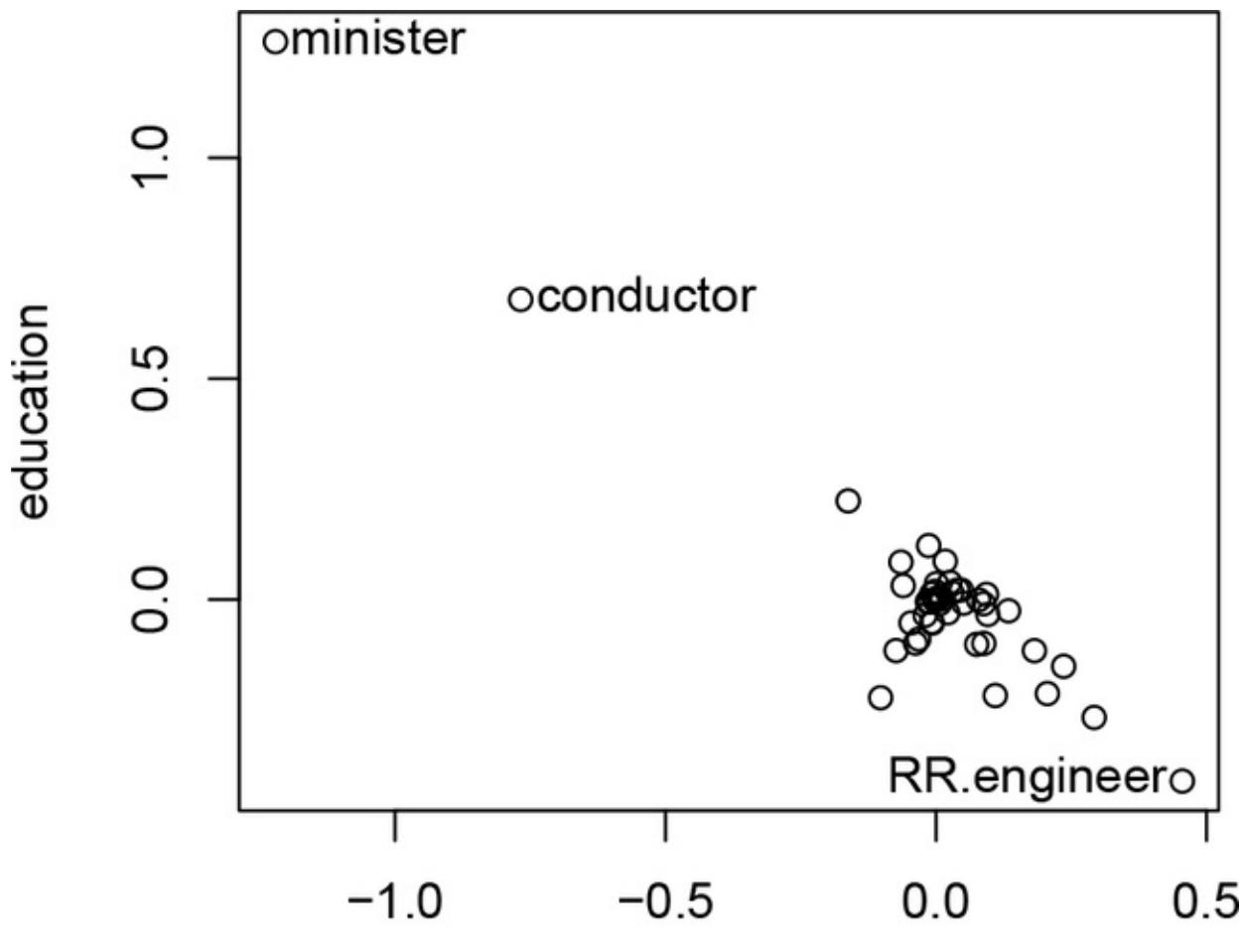
```
mod.duncan.3 <- update(mod.duncan,
subset = - whichNames(c("minister", "conductor"), Duncan))
compareCoefs(mod.duncan, mod.duncan.2, mod.duncan.3, se=FALSE)
```

Calls:

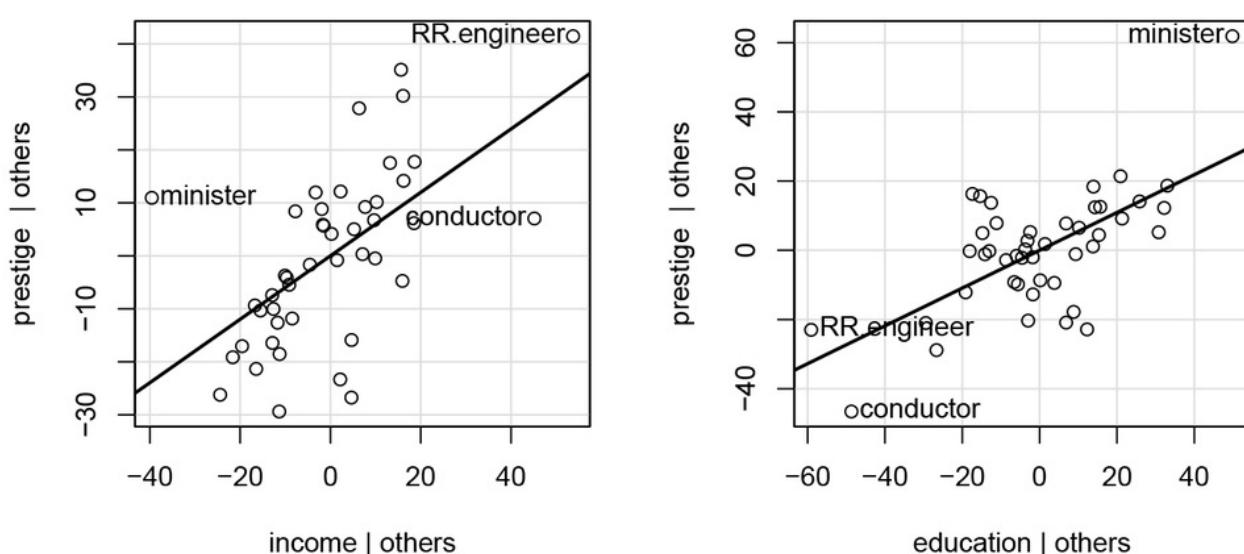
```
1: lm(formula = prestige ~ income + education, data =
Duncan)
2: lm(formula = prestige ~ income + education, data =
Duncan, subset = rownames(Duncan) != "minister")
3: lm(formula = prestige ~ income + education, data =
Duncan, subset = -whichNames(c("minister", "conductor"),
Duncan))
```

|             | Model 1 | Model 2 | Model 3 |
|-------------|---------|---------|---------|
| (Intercept) | -6.06   | -6.63   | -6.41   |
| income      | 0.599   | 0.732   | 0.867   |
| education   | 0.546   | 0.433   | 0.332   |

**Figure 8.8**  $dfbeta_{ij}$  values for the income and education coefficients in Duncan's occupational-prestige regression. Three points were identified interactively.



**Figure 8.9** Added-variable plots for Duncan's occupational-prestige regression.  
Added-Variable Plots



[11](#) The *Mahalanobis* or *generalized distance* takes into account the standard deviation of each variable and their correlation. Although the graphics functions in the **car** package have reasonable general defaults for point identification, we can also identify points interactively in diagnostic plots. Interactive point identification is supported by the argument `id=list(method="identify")` to `avPlots()` and other **car** graphics functions. See help ("showLabels") and the discussion of point identification in [Section 3.5](#).

We use the `whichNames()` function in the **car** package to return the indices of the cases "minister" and "conductor" in the Duncan data frame; setting the subset argument to the negative of these indices excludes the two cases from the regression.

## 8.4 Transformations After Fitting a Regression Model

Suspected outliers, and possibly cases with high leverage, should be studied individually to decide if they should be included in an analysis or not. Influential cases can cause changes in conclusions in an analysis and also require special treatment. Other systematic features in residual plots, such as curvature or apparent nonconstant variance, require action on the part of the analyst to modify the *structure* of the model to match the data more closely. Apparently distinct problems can also interact: For example, if the errors have a skewed distribution, then apparent outliers may be produced in the direction of the skew. Transforming the response to make the errors less skewed can solve this problem. Similarly, properly modeling a nonlinear relationship may bring apparently outlying cases in line with the rest of the data.

Transformations were introduced in [Section 3.4](#) in the context of examining data and with the understanding that regression modeling is often easier and more effective when the predictors behave as if they were normal random variables. Transformations can also be used *after* fitting a model, to improve a model that does not adequately represent the data. The methodology in these two contexts is very similar.

### 8.4.1 Transforming the Response

#### Box-Cox Transformations

The goals of fitting a model that exhibits linearity, constant variance, and normality can in principle require three different response transformations, but experience suggests that one transformation is often effective for all of these tasks. The most common method for selecting a transformation of the response in regression is due to Box and Cox (1964). If the response  $y$  is a strictly positive variable, then the Box-Cox power transformations, introduced in [Section 3.4.2](#) and implemented in the `bcPower()` function in the **car** package, are often effective. We replace the response  $y$  by  $T_{BC}(y, \lambda)$ , where

(8.7)

$$T_{BC}(y, \lambda) = y^{(\lambda)} = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \text{when } \lambda \neq 0 \\ \log y & \text{when } \lambda = 0 \end{cases}$$

The power parameter  $\lambda$  determines the transformation. The Box-Cox family essentially replaces  $y$  by  $y^\lambda$ , with  $y^0$  interpreted as  $\log(y)$ .<sup>12</sup>

<sup>12</sup> The subtraction of 1 and division by  $\lambda$  is inessential in that it doesn't alter the *shape* of the power transformation  $y^\lambda$ , with the caveat that dividing by  $\lambda$  does preserve the *order* of the data when  $\lambda$  is negative: For negative  $\lambda$ , such as  $\lambda = -1$  (the inverse transformation), the simple power  $y^\lambda$  *reverses* the order of the  $y$ -values.

Box and Cox proposed selecting the value of  $\lambda$  by analogy to the method of maximum likelihood, so that the residuals from the linear regression of  $T_{BC}(y, \lambda)$  on the predictors are as close to normally distributed as possible.<sup>13</sup> The **car** package provides three functions for estimating  $\lambda$ :

- The `boxCox()` function, a slight generalization of the `boxcox()` function in the **MASS** package (Venables & Ripley, 2002),<sup>14</sup> was illustrated in a related context in [Section 3.4.2](#).
- We illustrated the `powerTransform()` function in [Section 7.2.5](#) and will do so again in the current section.
- The `inverseResponsePlot()` function provides a visual method for selecting a normalizing response transformation, but we will not present it here (see [Section 8.9](#) for a brief description).

[13\\*](#) If  $T_{BC}(y, \lambda^0)|\mathbf{x}$  is normally distributed, then  $T_{BC}(y, \lambda_1)|\mathbf{x}$  cannot be normally distributed for  $\lambda_1 \neq \lambda^0$ , and so the distribution changes for every value of  $\lambda$ . The method Box and Cox proposed ignores this fact to get a maximum-likelihood-like estimate that turns out to have properties similar to those of maximum-likelihood estimates. In the interest of brevity, in the sequel, we refer to Box-Cox and similar estimates of transformation parameters as “maximum-likelihood estimates.”

[14](#) `boxCox()` adds a family argument, providing greater flexibility in the choice of a response transformation.

By way of illustration, we introduce an example that is of historical interest, because it was first used by Box and Cox (1964). The data, in the `Wool` data set in the **carData** package, are from an industrial experiment to study the strength of wool yarn under various conditions. Three predictors were varied in the experiment: `len`, the length of each sample of yarn in millimeters; `amp`, the amplitude of the loading cycle in angular minutes; and `load`, the amount of weight used in grams. The response, `cycles`, was the number of cycles until the sample failed. Data were collected using a  $3 \times 3 \times 3$  design, with each of the predictors at three equally spaced levels. We fit a linear model treating each of the three predictors as numeric variables with linear effects:

```
brief(wool.mod <- lm(cycles ~ len + amp + load, data=Wool))
```

|            | (Intercept) | len  | amp  | load  |
|------------|-------------|------|------|-------|
| Estimate   | 4521        | 13.2 | -536 | -62.2 |
| Std. Error | 1622        | 2.3  | 115  | 23.0  |

Residual SD = 488 on 23 df, R-squared = 0.729

This model fits the data poorly, as a residual plot versus fitted values (not shown) reveals obvious curvature. We can attempt to remedy the situation by transforming the response:

```

summary(p1 <- powerTransform(wool.mod))

bcPower Transformation to Normality
  Est Power Rounded Pwr Wald Lwr Bnd Wald Upr Bnd
Y1   -0.0592          0     -0.1789      0.0606

Likelihood ratio test that transformation parameter is equal to 0
(log transformation)
  LRT df  pval
LR test, lambda = (0) 0.92134 1 0.337

Likelihood ratio test that no transformation is needed
  LRT df  pval
LR test, lambda = (1) 84.076 1 <2e-16

```

The maximum-likelihood estimate of the transformation parameter is , with the 95% confidence interval for  $\lambda$  running from  $-0.179$  to  $0.061$ . The  $p$ -value for a test of  $\lambda = 0$  is large, and the  $p$ -value for  $\lambda = 1$  is very small. Both the likelihood-ratio tests and the Wald confidence interval suggest  $\lambda = 0$ , or a log transformation, as a reasonable choice, while leaving the response untransformed,  $\lambda= 1$ , is not acceptable. Replacing cycles by log (cycles) results in null residual plots, as the reader can verify.

## Zero or Negative Responses

The Box-Cox power family requires that the response variable be strictly positive, but in some instances, zero or negative values may occur in the data. A common approach to this problem is to add a “start” to the response to make all values of the response positive and then to apply the Box-Cox power family of transformations. Adding a start is problematic, however, because the transformation selected, and the effectiveness of the transformation, can be greatly affected by the value of the start. If the start is too small, then the transformed zero and negative cases are likely to become overly influential in the fit. If the start is too large, then information about variation in the response is potentially attenuated.

As a partial remedy, we recommend the use of the *Box-Cox with negatives* family of transformations, implemented in the bcnPower () function in the **car** package, in place of the standard Box-Cox power family. The Box-Cox with negatives family was introduced by Hawkins and Weisberg (2017) and is defined

in [Section 3.4.2](#). Like the Box-Cox family, there is a power parameter  $\lambda$ . In addition, there is a location parameter  $\gamma \geq 0$  that takes the place of a start. As  $\gamma$  approaches zero, the Box-Cox with negatives transformation approaches the standard Box-Cox power family.

In [Section 6.5](#), we fit a Poisson regression to Ornstein's data on interlocking directorates among Canadian corporations, regressing the number of interlocks maintained by each firm on the log of the firm's assets, nation of control, and sector of operation. Because number of interlocks is a count, the Poisson model is a natural starting point, but the original source (Ornstein, 1976) used a least-squares regression similar to the following:

```
mod.ornstein <- lm (interlocks ~ log (assets) + nation + sector,
data=Ornstein)
```

About 10% of the values of interlocks are zero, so we use the "bcnPower" family to select a normalizing transformation:

```
summary(p2 <- powerTransform(mod.ornstein, family="bcnPower"))

bcnPower transformation to Normality

Estimated power, lambda
  Est Power Rounded Pwr Wald Lwr Bnd Wald Upr Bnd
Y1    0.3545      0.33     0.2928    0.4162

Location gamma was fixed at its lower bound
  Est gamma Std Err. Wald Lower Bound Wald Upper Bound
Y1      0.1        NA            NA            NA

Likelihood ratio tests about transformation parameters
  LRT df pval
LR test, lambda = (0) 138.93  1    0
LR test, lambda = (1) 306.76  1    0
```

As is often the case when there are many zeros or negative values, the estimate of  $\gamma$  is close to zero, and so it is set to a value somewhat larger than zero (0.1)

and treated as fixed. The estimate of  $\lambda$  is then found by the maximum-likelihood method proposed by Box and Cox. The rounded estimate of the power is , or cube root, with a fairly narrow Wald confidence interval. No confidence interval is given for  $\gamma$  because its estimate is too close to zero. The likelihood-ratio tests provide evidence against  $\lambda = 0$  or  $\lambda = 1$ .<sup>15</sup> The suggestion is therefore to fit the model

```
Ornstein$interlocks.tran <- bcnPower (Ornstein$interlocks, lambda=1/3,  
gamma=0.1)
```

```
mod.ornstein.2 <- update (mod.ornstein, interlocks.tran ~ .)
```

<sup>15</sup> The likelihood-ratio-type tests concern  $\lambda$  only and treat  $\gamma$  as a nuisance parameter by averaging over  $\gamma$ .

and then proceed with the analysis. In this instance, the cube-root transformation is defined for zero values, and because  $\gamma$  is small, we could simply have used the cube-root transformation of interlocks to get virtually the same results.

Additional options for power transformations are given by help ("powerTransform"), help ("bcPower"), and help ("boxCox"). The powerTransform () function works for linear models, for multivariate linear models, and for linear mixed models. It is not useful, however, for non-Gaussian generalized linear models, where transformation of the expected response is accomplished by selection of a link function.

## Understanding Models With a Transformed Response

A common complaint about using a transformed response is that the resulting model is uninterpretable because the results are expressed in strange, transformed units. In Ornstein's interlocking-directorate regression, for example, a model with the response given by the cube root of the number of interlocks doesn't produce directly interpretable coefficients. We think that this complaint is misguided:

1. Most regression models are at best approximations, and allowing for a transformed response can help make for a better approximate model. As long as the transformation is monotone, tests of effects have a

straightforward interpretation. For example, in the Ornstein regression, we find that the transformed number of interlocks increases with  $\log(\text{assets})$ , even though the coefficient estimate is in uninterpretable units.

2. Some common transformations have straightforward interpretations. A few examples: Increasing the log base-2 by 1 implies doubling the response. If the response is the time in seconds required by a runner to traverse 100 meters, then the inverse of the response is the average velocity of the runner in units of 100 meters per second. If the response is the area of an apartment in square meters, than its square root is a linear measure of size in meters.
3. In any event, invariant regression summaries such as effect plots can be “untransformed” using an inverse transformation, as we have illustrated by example in [Figure 7.10](#) (page 369). For the bcnPower() transformation family, where the inverse is complicated, we supply the function bcnPowerInverse() to reverse the transformation; if, alternatively,  $z = y^\lambda$  is a simple power transformation of the response  $y$  (taken as  $z = \log(y)$  when  $\lambda = 0$ ), then the inverse transformation is just  $z^{1/\lambda}$  for  $\lambda \neq 0$  and  $\exp(z)$  for  $\lambda = 0$ .

## 8.4.2 Predictor Transformations

As outlined in [Section 3.4](#), predictor transformations can, and typically should, be performed *before* fitting models to the data. Even well-behaved predictors, however, aren’t necessarily linearly related to the response, and graphical diagnostic methods are available that can help select a transformation *after* fitting a model. Moreover, some kinds of nonlinearity can’t be fixed by transforming a predictor, and other strategies, such as polynomial regression or regression splines, may be entertained.

## Component-Plus-Residual and CERES Plots

*Component-plus-residual plots*, also called *partial-residual plots*, are a simple graphical device that can be effective for detecting the need to transform a predictor, say  $x_j$ , to a new variable  $T(x_j)$ , for some transformation  $T$ . The plot has  $x_{ij}$  on the horizontal axis and the *partial residuals*, , on the vertical axis. R. D. Cook (1993) shows that if the regressions of  $x_j$  on the other  $xs$  are approximately linear, then the regression function in the component-plus-residual plot provides a visualization of  $T$ . Alternatively, if the regressions of  $x_j$  on the other  $xs$

resemble polynomials, then a modification of the component-plus-residual plot due to Mallows (1986) can be used.

The `crPlots()` function in the `car` package constructs component-plus-residual plots for linear and generalized linear models. By way of example, we return to the Canadian occupational-prestige regression (from [Section 8.2.1](#)), this time fitting a regression model for prestige in which the predictors are income, education, and women. A scatterplot matrix of the response and the three predictors ([Figure 3.14](#) on page 147) suggests that the predictors are not all linearly related to each other, but no more complicated than quadratic regressions should provide reasonable approximations. Consequently, we draw the component-plus- residual plots specifying `order=2`, permitting quadratic relationships among the predictors:

```
prestige.mod.3 <- lm (prestige ~ income + education + women,  
data=Prestige)
```

```
crPlots (prestige.mod.3, order=2)
```

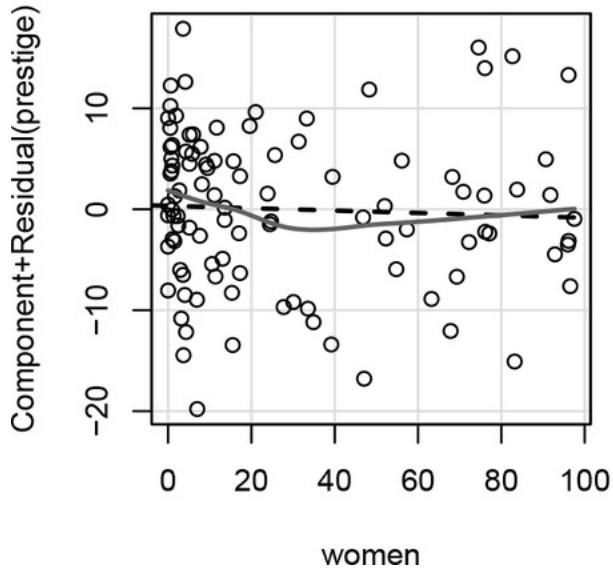
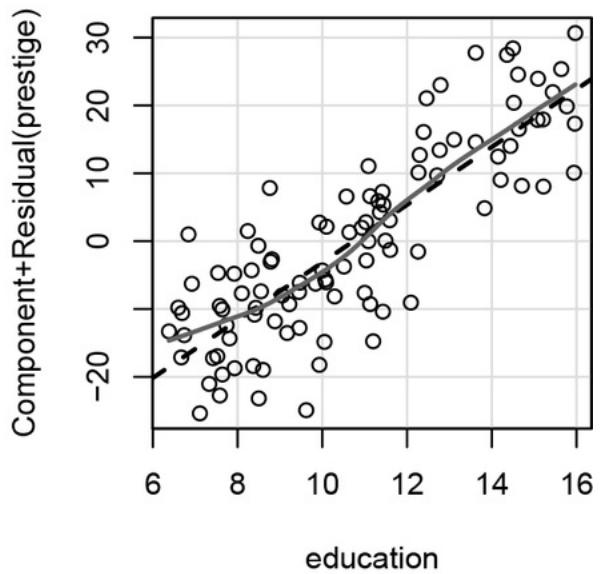
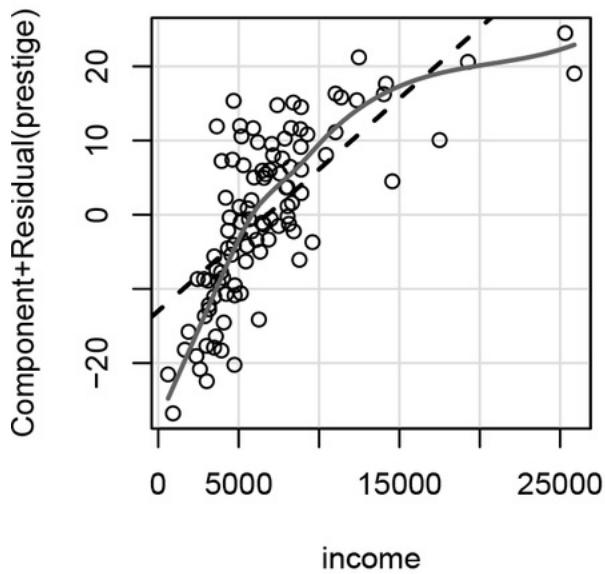
The component-plus-residual plots for the three predictors appear in [Figure 8.10](#). The broken line on each panel is the *partial fit*,  $b_j x_j$ , assuming linearity in the partial relationship between  $y$  and  $x_j$ . The solid line is a loess smooth, and it should suggest a transformation if one is appropriate, for example, via the bulging rule (see [Section 3.4.3](#)). Alternatively, the smooth might suggest a quadratic or cubic partial regression or, in more complex cases, the use of a regression spline.

For the Canadian occupational-prestige regression, the component-plus-residual plot for income is the most clearly curved, and transforming this variable first and refitting the model is therefore appropriate. In contrast, the component-plus-residual plot for education is only slightly nonlinear, and the partial relationship is not simple (in the sense of [Section 3.4.3](#)). Finally, the component-plus-residual plot for women looks mildly quadratic (although the lack-of-fit test computed by the `residualPlots()` function does not suggest a “significant” quadratic effect), with prestige first declining and then rising as women increases.

**Figure 8.10** Component-plus-residual plots of `order=2` for the Canadian

occupational-prestige regression.

## Component + Residual Plots



Trial-and-error experimentation moving income down the ladder of powers and roots suggests that a log transformation of this predictor produces a reasonable fit to the data:

```
prestige.mod.4 <- update (prestige.mod.3,
. ~ . + log2(income) - income)
```

This is the model that we fit in [Section 4.2.2](#). The component-plus-residual plot for women in the revised model (not shown) is broadly similar to the plot for women in [Figure 8.10](#), and the lack-of-fit test computed by `residualPlots()` has a *p*-value of 0.025, suggesting a quadratic regression:

```
prestige.mod.5 <- update(prestige.mod.4,
  . ~ . - women + poly(women, 2))

brief(prestige.mod.5, pvalues=TRUE)
(Intercept) education log2(income) poly(women, 2)1
Estimate      -1.11e+02  3.77e+00   9.36e+00    15.088
Std. Error     1.40e+01  3.47e-01   1.30e+00    9.336
Pr(>|t|)       4.16e-12  1.98e-18   1.26e-10   0.109
                poly(women, 2)2
Estimate        15.871
Std. Error       6.970
Pr(>|t|)        0.025

Residual SD = 6.95 on 97 df, R-squared = 0.843
```

The *p*-value for the test of the quadratic term for women in the new model is also .025.

If the regressions among the predictors are strongly nonlinear and not well described by polynomials, then component-plus-residual plots may not be effective in recovering nonlinear partial relationships between the response and the predictors. For this situation, R. D. Cook (1993) provides another generalization of component-plus-residual plots called *CERES plots* (for *C*ombining conditional *E*xpectations and *R*ESiduals). CERES plots use nonparametric-regression smoothers rather than polynomial regressions to adjust for nonlinear relationships among the predictors. The `ceresPlots()` function in the `car` package implements Cook's approach.

Experience suggests that nonlinear relationships among the predictors induce problems for component-plus-residual plots only when these relationships are strong. In such cases, a component-plus-residual plot can appear nonlinear even when the true partial regression is linear—a phenomenon termed *leakage*. For the Canadian occupational-prestige regression, quadratic component-plus-

residual plots (in [Figure 8.10](#)) and CERES plots are nearly identical to the standard component-plus-residual plots, as the reader may verify.

## Adding Partial Residuals to Effect Plots

Traditional component-plus-residual plots, as implemented in the `crPlots()` function, are drawn only for numeric predictors that enter a regression model additively. Fox and Weisberg (2018, *in press*) show how partial residuals can be added to effect plots for linear and generalized linear models of arbitrary complexity, including models with interactions between numeric predictors and factors and between numeric predictors. These methods are implemented in the **effects** package.

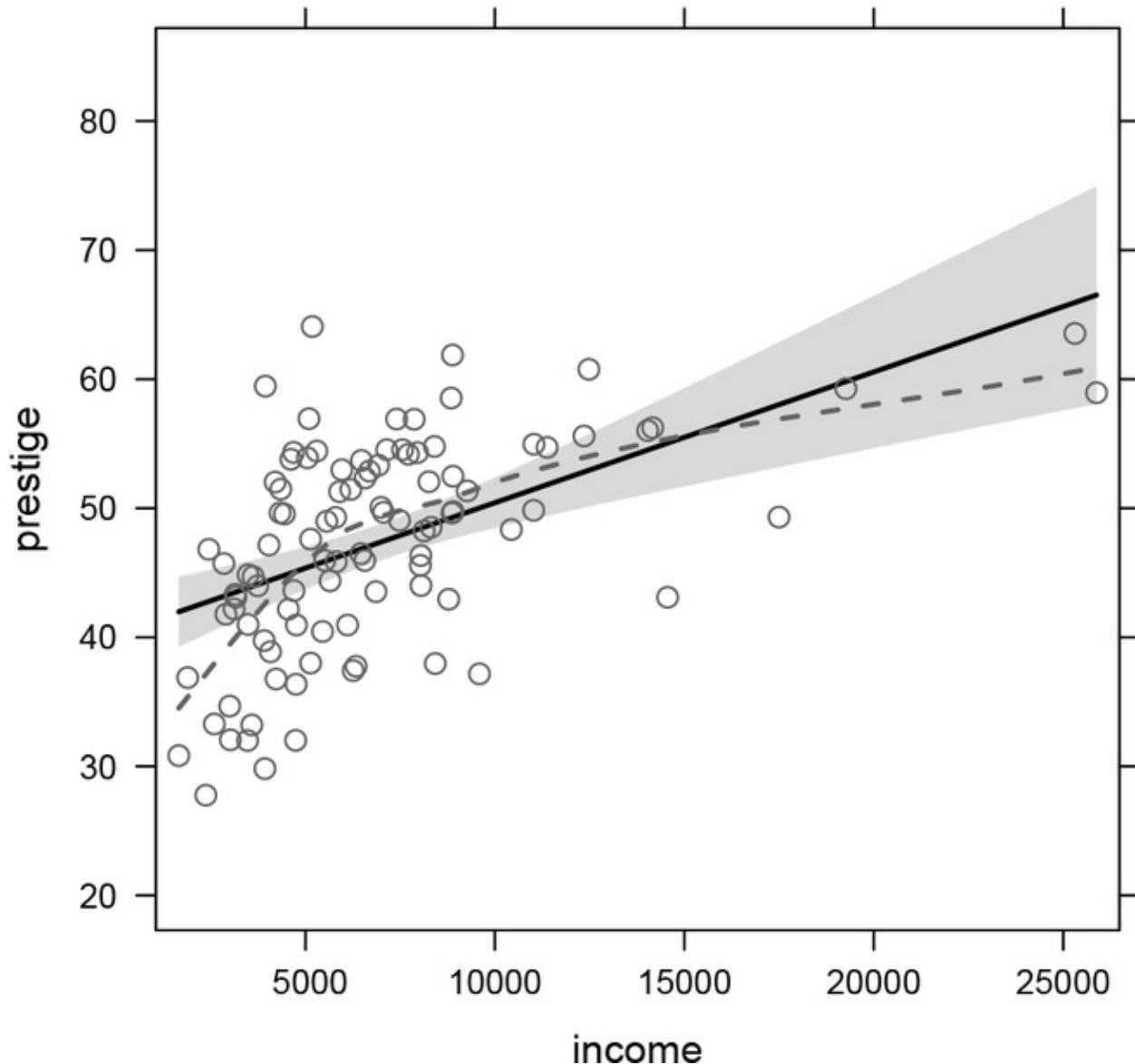
To illustrate, let's return to the regression model `prestige.mod.2` for the Canadian occupational-prestige data, in which prestige is regressed on income, education, and type. An effect plot with partial residuals for income in this model is a traditional component-plus-residual plot and is shown in [Figure 8.11](#):<sup>16</sup>

```
library ("effects")  
  
plot (Effect ("income", prestige.mod.2, residuals=TRUE),  
      partial.residuals=list (lty="dashed"))
```

<sup>16</sup> A subtle, and unimportant, distinction is that the partial residuals in the effect plot add back in the intercept from the regression model, together with constant terms involving the coefficients and means of the other predictors, and so the scaling of the vertical axis of the plot is different from that of a traditional component-plus-residual plot. The *shape* of the plot—that is, the configuration of points—is the same, however.

**Figure 8.11** Effect plot with partial residuals for income in the regression of prestige on income, education, and type for the Canadian occupational-prestige data.

## income effect plot



The straight line represents the fitted effect, and the curved broken line is a loess smooth of the partial residuals.<sup>17</sup> The plot shows obvious unmodeled nonlinearity in the partial relationship of prestige to income (and is very similar to the quadratic component-plus-residual plot for income in the model regressing prestige on income, education, and women, shown at the top left of [Figure 8.10](#)).

[17](#) We specify a broken line (lty="dashed") for the smooth because the default is to plot lines of different colors for the model fit and the smooth, and color isn't available to us in this book.

The graph at the top of [Figure 8.12](#) shows the effect plot with partial residuals for the predictors income and type simultaneously:

```
plot(Effect(c("income", "type"), prestige.mod.2, residuals=TRUE),  
partial.residuals=list(span=0.9, lty="dashed"), lattice=list(layout=c(3, 1)))
```

This effect plot is for the interaction of income with type in a model in which these predictors enter additively, as reflected in the parallel straight lines in the graph representing the fitted model, prestige.mod.2. We use a large span, partial.residuals=list (span=0.9, lty="dashed"), for the loess smoother because of the small numbers of cases in the various levels of type; we also arrange the panels horizontally via the argument lattice=list (layout= c (3, 1)) (see help ("plot.eff") for details). The loess smooths are nearly linear but with different slopes—greatest for blue-collar ("bc") occupations and smallest for professional and managerial ("prof") occupations—suggesting an un-modeled interaction between the two predictors.

Refitting the model reveals that the income  $\times$  type interaction has a very small *p*-value, and the effect plot for income and type in this model, shown at the bottom of [Figure 8.12](#), supports the model:<sup>18</sup>

```

prestige.mod.6 <- update(prestige.mod.2,
  . ~ income*type + education, data=Prestige)
Anova(prestige.mod.6)

Anova Table (Type II tests)

Response: prestige
          Sum Sq Df F value    Pr(>F)
income       1059  1 25.41 2.3e-06
type         591  2   7.09  0.0014
education    1068  1 25.63 2.1e-06
income:type   890  2 10.68 6.8e-05
Residuals    3791 91

```

```

plot(Effect(c("income", "type"), prestige.mod.6,
            residuals=TRUE),
      partial.residuals=list(span=0.9, lty="dashed"),
      lattice=list(layout=c(3, 1)))

```

[18](#) Using the log of income in place of income in the respecified model straightens the slight curvature barely discernable in the panels for blue-collar and white-collar occupations at the bottom of [Figure 8.12](#).

This example nicely illustrates how unmodeled interaction can be reflected in apparent nonlinearity in a component-plus-residual plot.

## 8.5 Nonconstant Error Variance

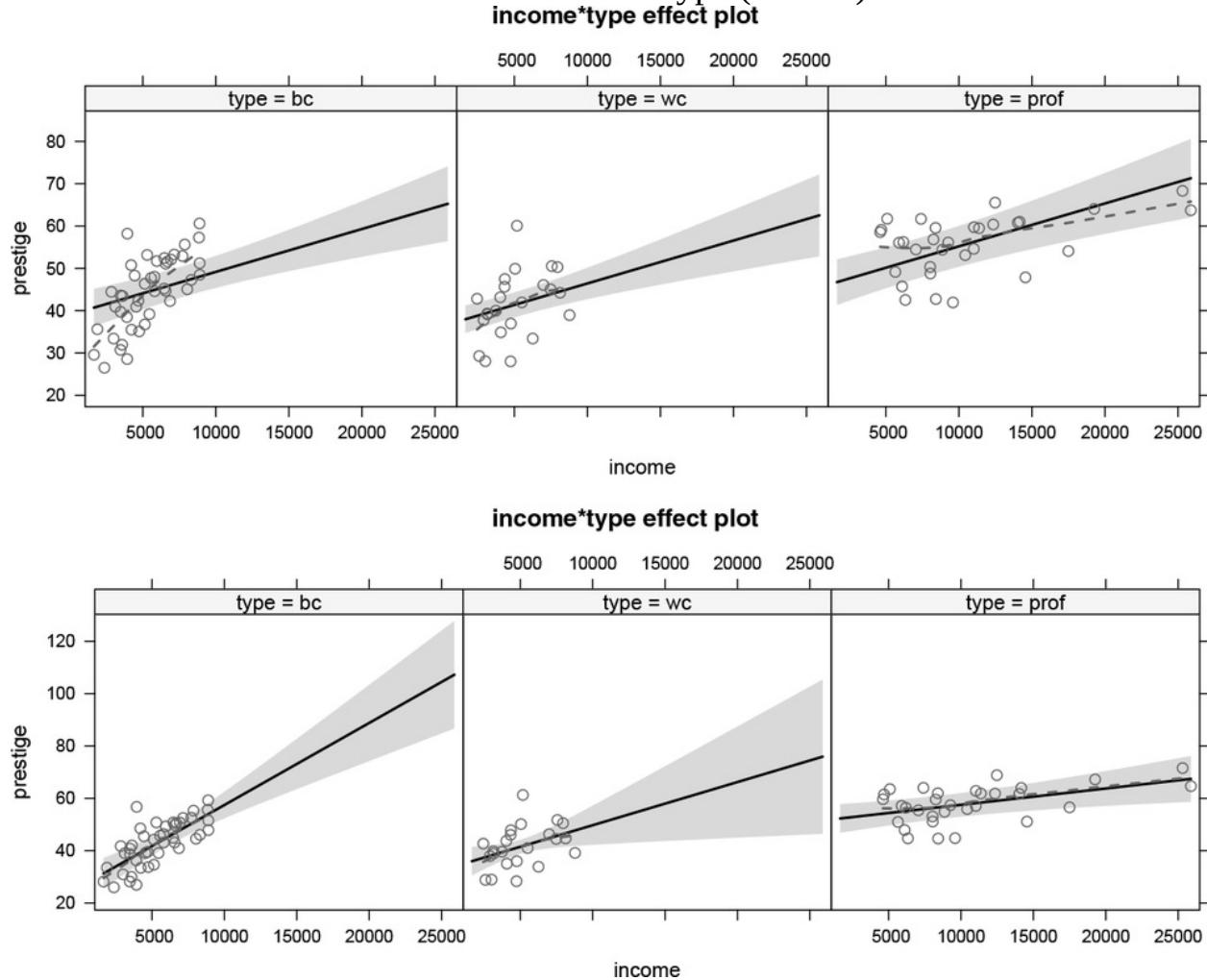
One of the assumptions of the standard linear model is that the error variance is fully known apart from an unknown constant,  $\sigma^2$ . It is, however, possible that the error variance depends on one or more of the predictors, on the magnitude of the response, or systematically on some other variable.

To detect nonconstant variance as a function of a variable  $z$ , we can plot the Pearson residuals versus  $z$ . Nonconstant variance would be diagnosed if the

variability of the residuals in the graph either increased from left to right, decreased from left to right, or displayed another systematic pattern, such as large variation in the middle of the range of  $z$  and smaller variation at the edges.

In multiple regression, there are many potential plotting directions. Because obtaining a two-dimensional graph entails projecting the predictors from many dimensions onto one horizontal axis, however, we can never be sure if a 2D plot showing nonconstant variability really reflects nonconstant error variance or some other problem, such as unmodeled nonlinearity (R. D. Cook, 1998, [Section 1.2.1](#)).

**Figure 8.12** Effect plots with partial residuals for income and type in the regression of prestige on income, education, and type: additive model (top), model with interaction between income and type (bottom).



For an example, we return to the bank transactions data (introduced in [Section](#)

[5.1.1](#)), relating minutes of labor time in branches of a large bank to the number of transactions of two types, t1 and t2; the data are in the Transact data set in the **carData** package:

```
mod.transact <- lm(time ~ t1 + t2, data=Transact)
brief(mod.transact)

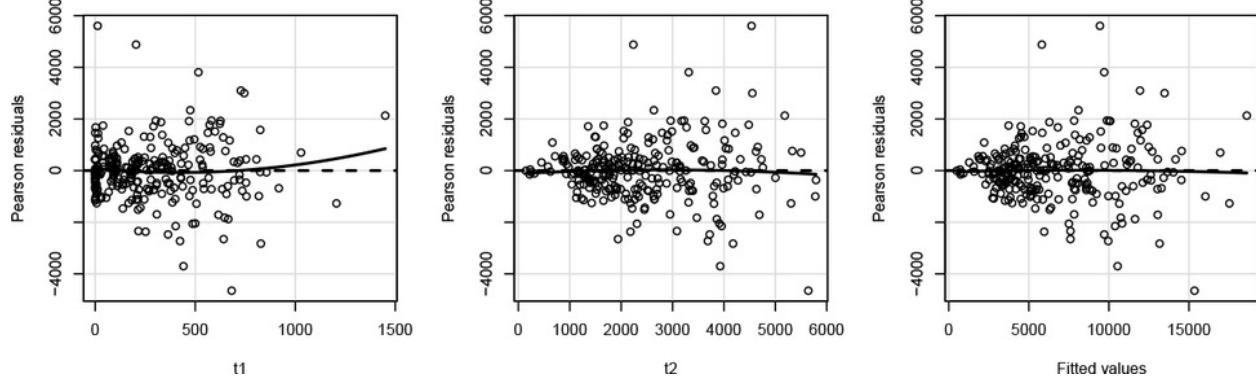
            (Intercept)      t1      t2
Estimate          144 5.462 2.0345
Std. Error        171 0.433 0.0943

Residual SD = 1143 on 258 df, R-squared = 0.909

residualPlots(mod.transact, tests=FALSE, layout=c(1, 3))
```

Two of the residual plots shown in [Figure 8.13](#) exhibit a characteristic fan shape, with increasing variability moving from left to right for t2 and for the plot against fitted values, but not as obviously for t1. This pattern suggests nonconstant residual variance, possibly as a function of t2 only.

**Figure 8.13** Residual plots for the transactions data.



## 8.5.1 Testing for Nonconstant Error Variance

Breusch and Pagan (1979) and R. D. Cook and Weisberg (1983) suggest a score test for nonconstant error variance in a linear model. The assumption underlying the test is that the variance is constant, or it depends on the conditional mean of

$$\text{the response, } \text{Var}(\varepsilon_i) = \sigma^2 g[\text{E}(y|\mathbf{x})]$$

or it depends on some linear combination of regressors  $z_1, \dots, z_p$ ,

$$(8.8) \quad \text{Var}(\varepsilon_i) = \sigma^2 g(\gamma_1 z_{i1} + \dots + \gamma_p z_{ip})$$

In typical applications, the  $z$ s are selected from the  $x$ s and often include all of the  $x$ s, but other choices of  $z$ s are possible. For example, in an educational study, variability of test scores might differ among the schools in the study, and a set of dummy regressors for schools would be candidates for the  $z$ s.

The `ncvTest()` function in the `car` package implements the score test. We compute four score tests for the bank transactions regression:

```
(test.t1 <- ncvTest(mod.transact, ~ t1))
```

Non-constant Variance Score Test

Variance formula: ~ t1

Chisquare = 26.525, Df = 1, p = 2.6e-07

```
(test.t2 <- ncvTest(mod.transact, ~ t2))
```

Non-constant Variance Score Test

Variance formula: ~ t2

Chisquare = 76.589, Df = 1, p = <2e-16

```
(test.t1t2 <- ncvTest(mod.transact, ~ t1 + t2))
```

```
Non-constant Variance Score Test  
Variance formula: ~ t1 + t2  
Chisquare = 82.932, Df = 2, p = <2e-16
```

### ***ncvTest(mod.transact)***

```
Non-constant Variance Score Test  
Variance formula: ~ fitted.values  
Chisquare = 61.659, Df = 1, p = 4.08e-15
```

All four tests are for the null hypothesis that the  $\sigma^2$ s are equal to zero in Equation 8.8 against the alternatives that nonconstant variance is a function of  $t_1$  alone, a function of  $t_2$  alone, a function of both  $t_1$  and  $t_2$  in some linear combination, and, finally, a function of the conditional mean of the response, as captured by the fitted values. In this instance, all four tests have very small  $p$ -values, suggesting likely nonconstant variance.

An approximate test of the null hypothesis that residual variance depends on  $t_2$  alone versus the alternative that it depends on both  $t_1$  and  $t_2$  is obtained by computing the difference between the chi-square test statistics in `test.t1t2` and `test.t2` above:

```
(stat <- test.t1t2$ChiSquare - test.t2$ChiSquare)  
[1] 6.3429  
  
(df <- test.t1t2$Df - test.t2$Df)  
[1] 1  
  
pchisq(stat, df, lower.tail=FALSE)  
[1] 0.011785
```

Even though this test suggests that the error variance should be modeled as a function of both  $t_1$  and  $t_2$ , residual variation is much more strongly related to  $t_2$  than to  $t_1$ , and so we might, as a reasonable approximation, refit the regression by weighted least squares, with weights given by  $1/t_2$ . Alternatively, and more

naturally, the model could be fit as a generalized linear model with the identity link and gamma errors, as suggested for this example by Cunningham and Heathcote (1989). Finally, because OLS estimates are unbiased even if the variance function is incorrectly specified, the estimates could be obtained by OLS, but with standard errors and tests computed using either the bootstrap ([Section 5.1.3](#)) or a sandwich coefficient-variance estimator ([Section 5.1.2](#)). These corrections may be used by many functions in the `car` package, including `linearHypothesis()`, `deltaMethod()`, `Anova()`, `Confint()`, `S()`, `brief()`, and `Predict()`.

## 8.6 Diagnostics for Generalized Linear Models

Most of the diagnostics of the preceding sections extend straightforwardly to generalized linear models. These extensions typically take advantage of the computation of maximum-likelihood estimates for generalized linear models by *iterated weighted least squares* (IWLS: see [Section 6.12](#)), which in effect approximates the true log-likelihood for a GLM by a weighted-least-squares problem. At convergence of the IWLS algorithm, diagnostics are formed as if the weighted-least-squares problem were the problem of interest, and so the exact diagnostics for the weighted-least-squares fit are approximate diagnostics for the original GLM. Seminal work on the extension of linear-least-squares diagnostics to generalized linear models was done by Pregibon (1981), Landwehr, Pregibon, and Shoemaker (1980), Wang (1985, 1987), and Williams (1987). We focus here on methods that differ from their application in linear models.

### 8.6.1 Residuals and Residual Plots

One of the major philosophical, though not necessarily practical, differences between linear-model diagnostics and GLM diagnostics is in the definition of residuals. In linear models, the ordinary residual is the difference  $y - \hat{y}$ , which is meant to mimic the statistical error  $\epsilon = y - E(y|\mathbf{x})$ . Apart from Gaussian generalized linear models, there is no additive error in the definition of a GLM, and so the idea of a residual has a much less firm footing.

Residuals for GLMs are generally defined in analogy to linear models. Here are the various types of GLM residuals that are available in R:

- *Response residuals* are simply the differences between the observed response and its estimated expected value: . These correspond to the ordinary residuals in the linear model. Apart from the Gaussian case, the response residuals are not used in diagnostics, however, because they ignore the non-constant variance that is intrinsic to non-Gaussian GLMs.
- *Working residuals* are the residuals from the final IWLS fit. The working residuals may be used to define partial residuals for component-plus-residual plots and effect plots (see below) but are not usually accessed directly by the user.
- *Pearson residuals* are casewise components of the Pearson goodness-of-fit statistic for the model,

$$e_{Pi} = \frac{y_i - \hat{\mu}_i}{\sqrt{\widehat{\text{Var}}(y_i|\mathbf{x})/\hat{\phi}}}$$

where  $\hat{\phi}$  is the estimated dispersion parameter in the GLM. Formulas for  $\text{Var}(y|\mathbf{x})$  are given in the last column of [Table 6.2](#) (page 274). This definition of  $e_{Pi}$  corresponds exactly to the Pearson residuals defined in Equation 8.6 (page 387) for WLS regression. These are a basic set of residuals for use with a GLM because of their direct analogy to linear models. For a model named `m`, the command `residuals(m, type="pearson")` returns the Pearson residuals.

- *Standardized Pearson residuals* correct for conditional response variation and

$$e_{PSi} = \frac{y_i - \hat{\mu}_i}{\sqrt{\widehat{\text{Var}}(y_i|\mathbf{x})(1 - h_i)}}$$

for the leverage of the cases:

To compute the  $e_{PSi}$ , we need to define the hat-values  $h_i$  for GLMs. The  $h_i$  are taken from the final iteration of the IWLS procedure for fitting the model, and have the usual interpretation, except that, unlike in a linear model, the hat-values in a generalized linear model depend on  $y$  as well as on the configuration of the  $x$ s.

- *Deviance residuals*,  $e_{Di}$ , are the square roots of the casewise components of the residual deviance, attaching the sign of . In the linear model, the deviance residuals reduce to the Pearson residuals. The deviance residuals are often the preferred form of residual for generalized linear models and are returned by the

command residuals (m, type="deviance").

$$e_{DSi} = \frac{e_{Di}}{\sqrt{\hat{\phi}(1 - h_i)}}$$

- *Standardized deviance residuals* are
- The  $i$ th *Studentized residual* in a linear model is the scaled difference between the response and the fitted value computed without case  $i$ . Because of the special structure of the linear model, these differences can be computed without actually refitting the model removing case  $i$ , but this is not true for generalized linear models. While computing  $n$  regressions to get the Studentized residuals is not impossible, it is not a desirable option when the sample size is large. An approximation due to Williams (1987) is therefore used instead:

$$e_{Ti} = \text{sign}(y_i - \hat{\mu}_i) \sqrt{(1 - h_i)e_{DSi}^2 + h_i e_{PSi}^2}$$

The approximate Studentized residuals are computed when the function `rstudent()` is applied to a GLM. A Bonferroni outlier test using the standard-normal distribution may be based on the largest absolute Studentized residual and is implemented in the `outlierTest()` function.

As an example, we return to the Canadian women's labor force participation data described in [Section 6.7](#). We define a binary rather than a polytomous response, with categories working or not working outside the home, and fit a logistic-regression model to the data:

```
mod.working <- glm (partic != "not.work" ~ hincome + children,
family=binomial, data=Womenlf)
```

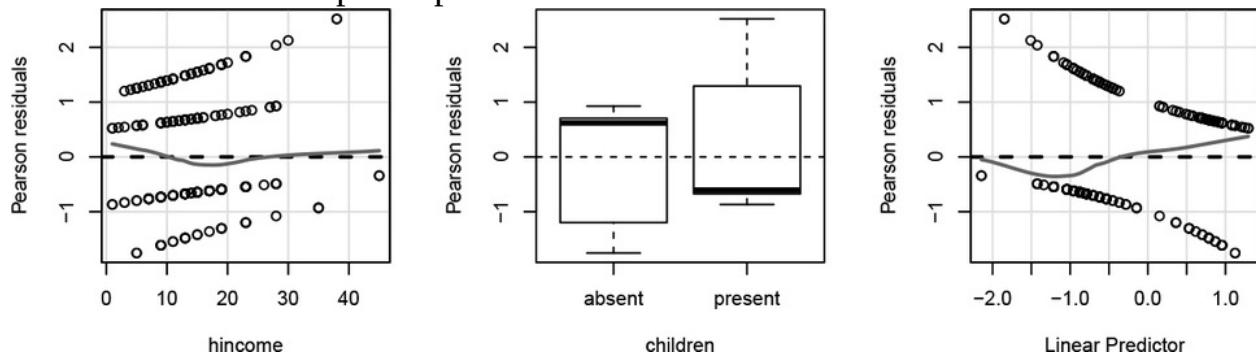
```
brief (mod.working)
```

|                | (Intercept) | hincome | childrenpresent |
|----------------|-------------|---------|-----------------|
| Estimate       | 1.336       | -0.0423 | -1.576          |
| Std. Error     | 0.384       | 0.0198  | 0.292           |
| exp (Estimate) | 3.803       | 0.9586  | 0.207           |

Residual deviance = 320 on 260 df

The expression partic != "not.work" creates a logical vector, which serves as the binary response variable in the model.

**Figure 8.14** Residual plots for the binary logistic regression fit to the Canadian women's labor force participation data.



The `residualPlots()` function provides basic plots of residuals versus the predictors and versus the linear predictor ([Figure 8.14](#)):

```
residualPlots(mod.working, layout=c(1, 3))
```

|          | Test stat | Pr(> Test stat ) |
|----------|-----------|------------------|
| hincome  | 1.23      | 0.27             |
| children |           |                  |

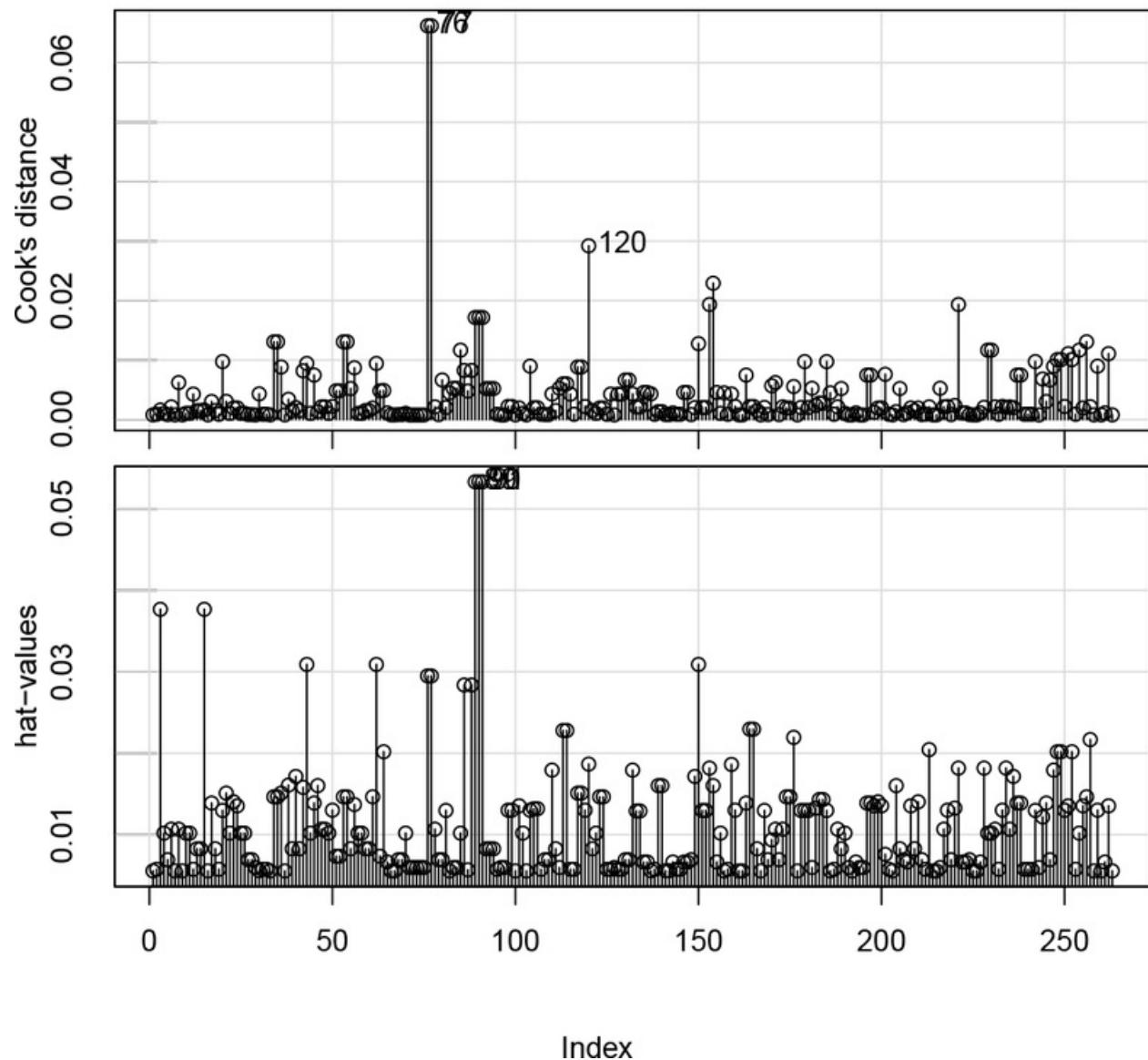
The function plots Pearson residuals versus each of the predictors in turn. Instead of plotting residuals against fitted values, however, `residualPlots()` plots residuals against the estimated linear predictor, `.` Each panel in the graph by default includes a smooth fit rather than a quadratic fit; a lack of fit test is provided only for the numeric predictor `hincome` and not for the factor `children` or for the estimated linear predictor.

In binary regression models, plots of Pearson residuals or of deviance residuals are strongly patterned. In a plot against the linear predictor, the residuals can only assume two values, depending on whether the response is equal to zero or 1. Because the factor children only has two levels, the residuals when plotted against hincome can only take on four distinct values, corresponding to the combinations of the binary response and the two levels of children. A correct model requires that the conditional mean function in any residual plot be constant as we move across the plot, and a fitted smooth helps us to learn about the conditional mean function even in these highly discrete cases. Neither of the smooths, against hincome or the linear predictor, shown in [Figure 8.14](#), is especially curved. The lack-of-fit test for hincome has a large  $p$ -value, confirming our view that this plot does not indicate lack of fit.

The residuals for children are shown as boxplots because children is a factor. Recalling that the heavy line in each boxplot represents the median, most of the residuals for children "absent" are positive but with a strong negative skew, while most of the residuals for children "present" are negative with a positive skew, implying that the model used may be inadequate. This residual plot suggests that the interaction of children and hincome should be explored, an exercise we leave to the interested reader.

**Figure 8.15** Index plots of diagnostic statistics for the logistic regression fit to the Canadian women's labor force participation data.

## Diagnostic Plots



### 8.6.2 Influence Measures

An approximation to Cook's distance for GLMs is

$$D_i = \frac{e_{PSi}^2}{(k+1)} \times \frac{h_i}{1-h_i}$$

These values are returned by the `cooks.distance()` function.

[Figure 8.15](#) shows index plots of Cook's distances and hat-values, produced by the command:

```
influenceIndexPlot (mod.working, vars=c ("Cook", "hat"), id=list (n=3))
```

Setting `vars=c ("Cook", "hat")` limits the graphs to these two diagnostics. Cases 76 and 77 have the largest Cook's distances, although even these are quite small. We remove both Cases 76 and 77 as a check:

```
compareCoefs (mod.working,
  update (mod.working, subset=-c(76, 77)))

Calls:
1: glm(formula = partic != "not.work" ~ hincome + children,
       family = binomial, data = Womenlf)
2: glm(formula = partic != "not.work" ~ hincome + children,
       family = binomial, data = Womenlf, subset = -c(76, 77))

          Model 1 Model 2
(Intercept)     1.336   1.609
SE             0.384   0.405

hincome        -0.0423 -0.0603
SE              0.0198  0.0212

childrenpresent -1.576  -1.648
SE              0.292   0.298
```

The reader can verify that removing just one of the two cases does not alter the results much, but removing *both* cases changes the coefficient of husband's income by more than 40%, about one standard error. Apparently, the two cases are an influential pair that partially mask each other, and removing them both is required to produce a meaningful change in the coefficient for `hincome`. Cases 76 and 77 are women working outside the home even though both have children and high-income husbands.

### 8.6.3 Graphical Methods: Added-Variable Plots, Component-Plus-Residual Plots, and Effect Plots With Partial Residuals

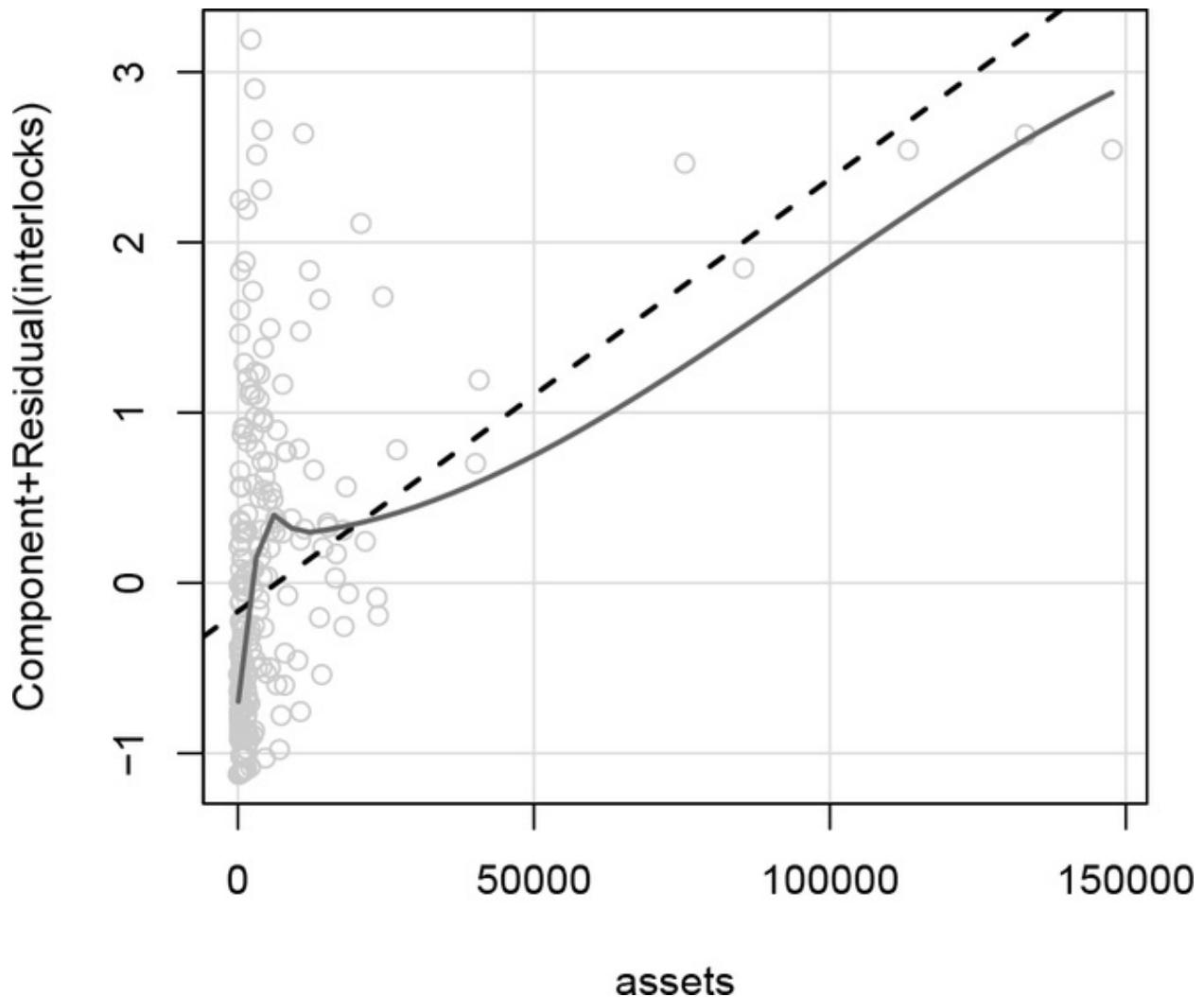
Added-variable plots are extended to generalized linear models by approximating the two fitted regressions required to generate the plot. By default, the `avPlots()` function uses an approximation suggested by Wang (1985). Added-variable plots for binary-regression models can be uninformative, however, because of the extreme discreteness of the response variable.

Component-plus-residual and CERES plots also extend straightforwardly to generalized linear models. Nonparametric smoothing of the resulting scatterplots can be important to interpretation, especially in models for binary responses, where the discreteness of the response makes the plots difficult to decode. Similar, if less striking, effects due to discreteness can also occur for binomial and count data.

For an illustrative component-plus-residual plot, we reconsider Ornstein's interlocking-directorate quasi-Poisson regression from [Section 6.5](#), but now we fit a model that uses assets as a predictor rather than the log of assets:

```
mod.ornstein.qp <- glm (interlocks ~ assets + nation + sector,  
family=quasipoisson, data=Ornstein)  
  
crPlots (mod.ornstein.qp, ~ assets, col="gray")
```

**Figure 8.16** Component-plus-residual plot for assets in the quasi-Poisson regression fit to Ornstein's interlocking-directorate data.



```
mod.ornstein.qp <- glm(interlocks ~ assets + nation + sector,
family=quasipoisson, data=Ornstein)
crPlots(mod.ornstein.qp, ~ assets, col="gray")
```

The component-plus-residual plot for assets is shown in [Figure 8.16](#). This plot is hard to interpret because of the extreme positive skew in assets, but it appears as if the assets slope is a good deal steeper at the left than at the right. The bulging rule, therefore, points toward transforming assets down the ladder of powers, and indeed the log rule in [Section 3.4.1](#) suggests replacing assets by its logarithm *before* fitting the regression (which, of course, is what we did originally):

```

mod.ornstein.qp.2 <- update(mod.ornstein.qp,
  . ~ log2(assets) + nation + sector)
crPlots(mod.ornstein.qp.2, ~ log2(assets))

```

The linearity of the follow-up component-plus-residual plot in [Figure 8.17](#) confirms that the log-transform is a much better scale for assets.

We continue with a reexamination of the binary logistic-regression model fit to Mroz's women's labor force participation data in [Section 6.3](#). One of the predictors in this model, the log of the woman's expected wage rate (lwg), has an unusual definition: For women in the labor force, for whom the response lfp = "yes", lwg is the log of the women's *actual* wage rate, while for women not in the labor force, for whom lfp = "no", lwg is the log of the *predicted* wage rate from the regression of wages on the other predictors.

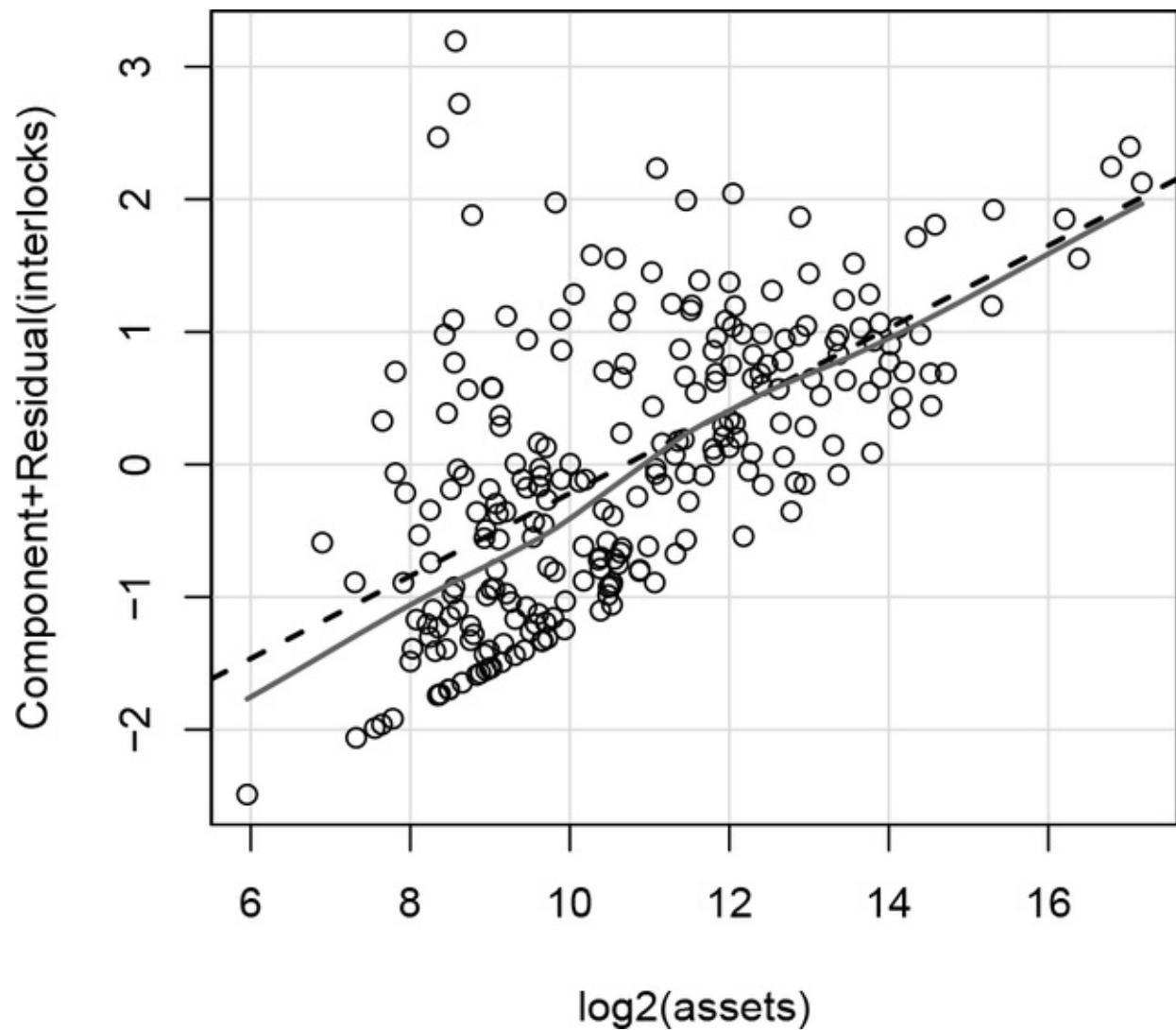
To obtain a component-plus-residual plot for lwg ([Figure 8.18](#)):

```

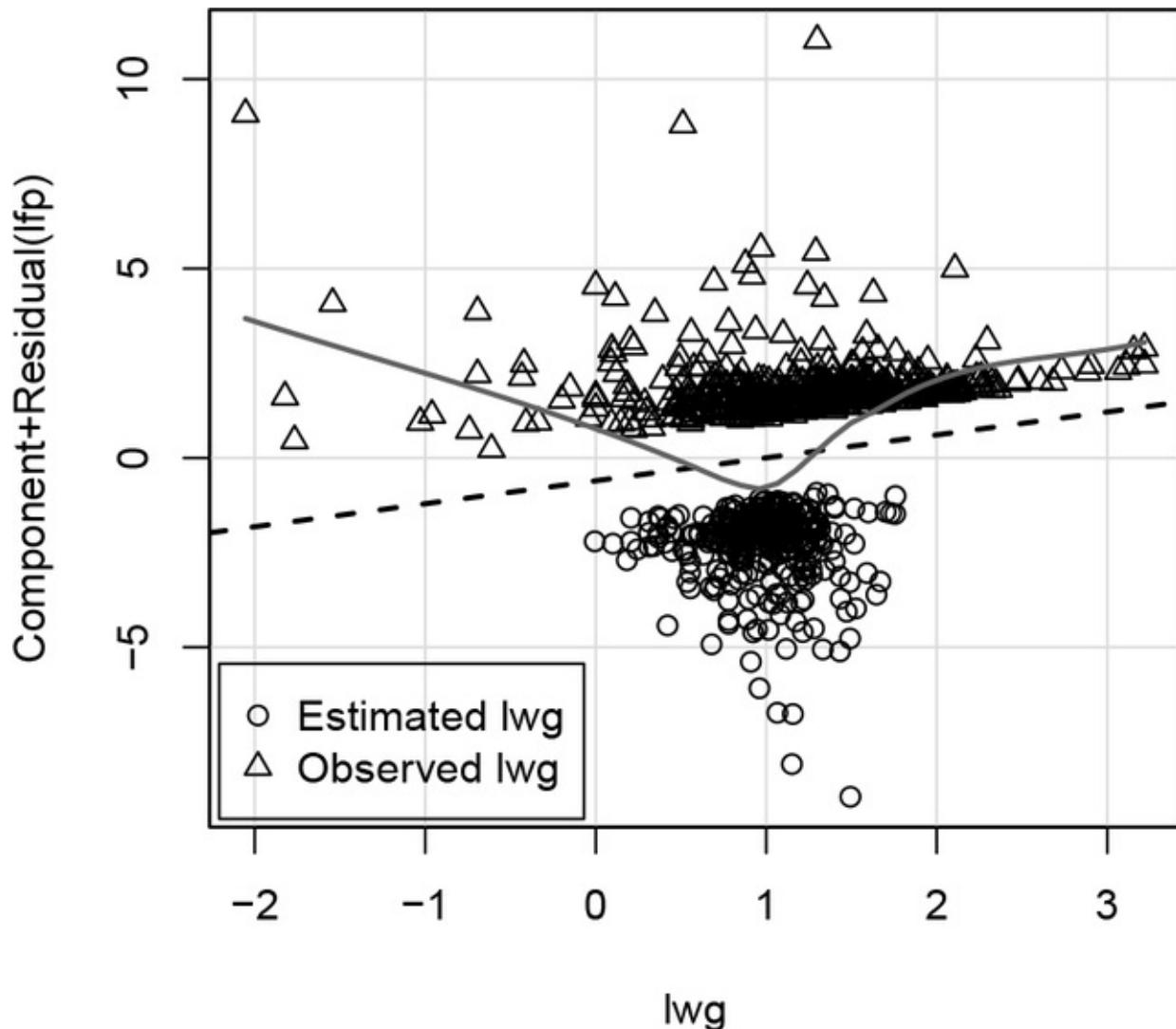
mod.mroz <- glm(lfp ~ k5 + k618 + age + wc + hc + lwg + inc,
  family=binomial, data=Mroz)
crPlots(mod.mroz, "lwg", pch=as.numeric(Mroz$lfp))
legend("bottomleft", c("Estimated lwg", "Observed lwg"),
  pch=1:2, inset=0.01)

```

**Figure 8.17** Component-plus-residual plot for the log of assets in the respecified quasi-Poisson regression for Ornstein's data.



**Figure 8.18** Component-plus-residual plot for  $\text{lwg}$  in the binary logistic regression for Mroz's women's labor force participation data.



We specify the pch argument to `crPlots()` to use different plotting symbols for the two values of `lfp` and add a legend to the graph.<sup>19</sup> The peculiar split in the plot reflects the binary response variable, with the lower cluster of points corresponding to `lfp = "no"` and the upper cluster to `lfp = "yes"`. It is apparent that `lwg` is much less variable when `lfp = "no"`, inducing an artificially curvilinear relationship between `lwg` and `lfp`: We expect fitted values (such as the values of `lwg` when `lfp = "no"`) to be more homogeneous than observed values, because fitted values lack a residual component of variation.

<sup>19</sup> See [Chapter 9](#) on customized graphics in R.

We leave it to the reader to construct component-plus-residual or CERES plots for the other predictors in the model.

For a final example, we return to a binary logistic regression fit to the Cowles and Davis volunteering data from [Section 6.3.2](#):

```
cowles.mod <- glm (volunteer ~ sex + neuroticism*extraversion,  
data=Cowles, family=binomial)
```

We showed effect plots for Cowles and Davis's model in [Figure 6.3](#) (page 285). To determine whether the linear-by-linear interaction between neuroticism and extraversion is supported by the data, we can add partial residuals to the effect plots of the interaction ([Figure 8.19](#)):

```
plot (predictorEffects (cowles.mod, ~ neuroticism + extraversion,  
residuals=TRUE),  
  
partial.residuals=list (span=3/4, lty="dashed"), lattice=list (layout=c (1,  
4)))
```

The predictor effect plot for neuroticism is shown at the left, for extraversion at the right. In each case, the other predictor in the interaction increases across its range from the bottom panel to the top panel, as indicated by the black vertical line in the strip at the top of each panel. By default, the conditioning predictor in each plot is set to four values equally spaced across its range. We use the argument `partial.residuals=list (span=3/4, lty="dashed")` to plot () to increase the span of the loess smoother (shown as a broken line in each panel) slightly to 3/4 from the default of 2/3 and the argument `lattice=list (layout=c (1,4))` to orient the panels in each effect plot vertically. The data appear to support the linear-by-linear form of the interaction.

## 8.7 Diagnostics for Mixed-Effects Models

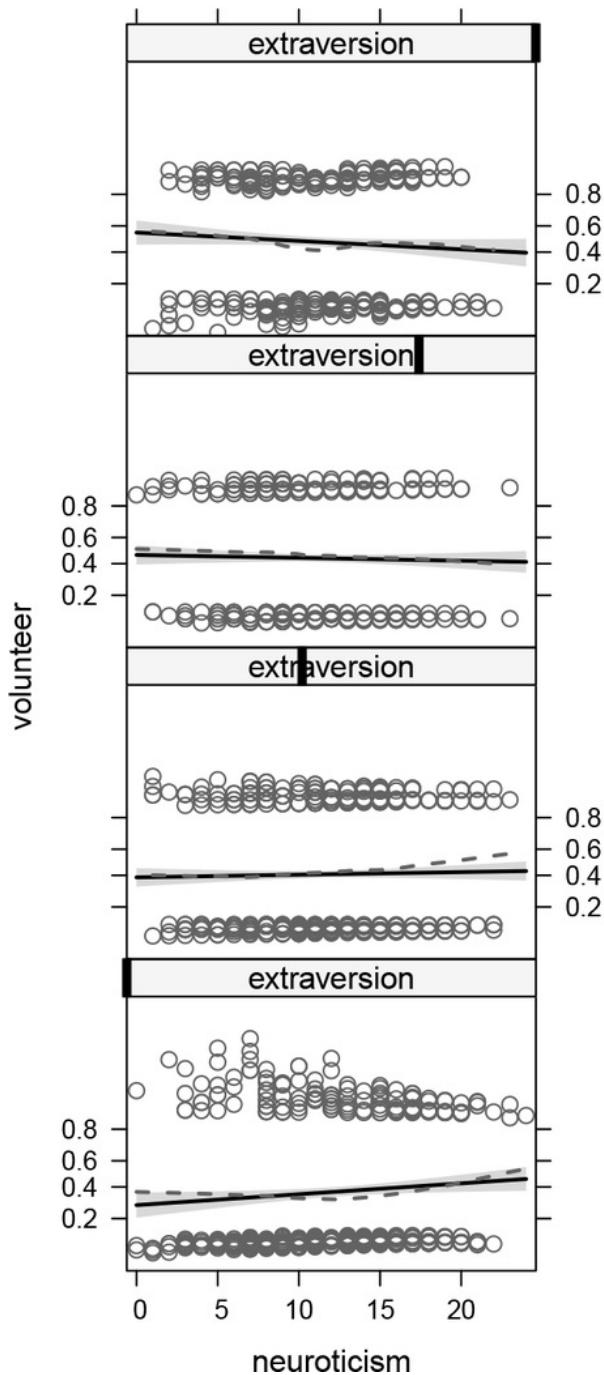
Regression diagnostics for mixed-effects models are relatively less developed than for linear and generalized linear models. We focus in this section on component-plus-residual plots, which are implemented for mixed-effects models in the **effects** package, and on deletion diagnostics for influential data, which are implemented in the **car** package.

### 8.7.1 Mixed-Model Component-Plus-Residual Plots

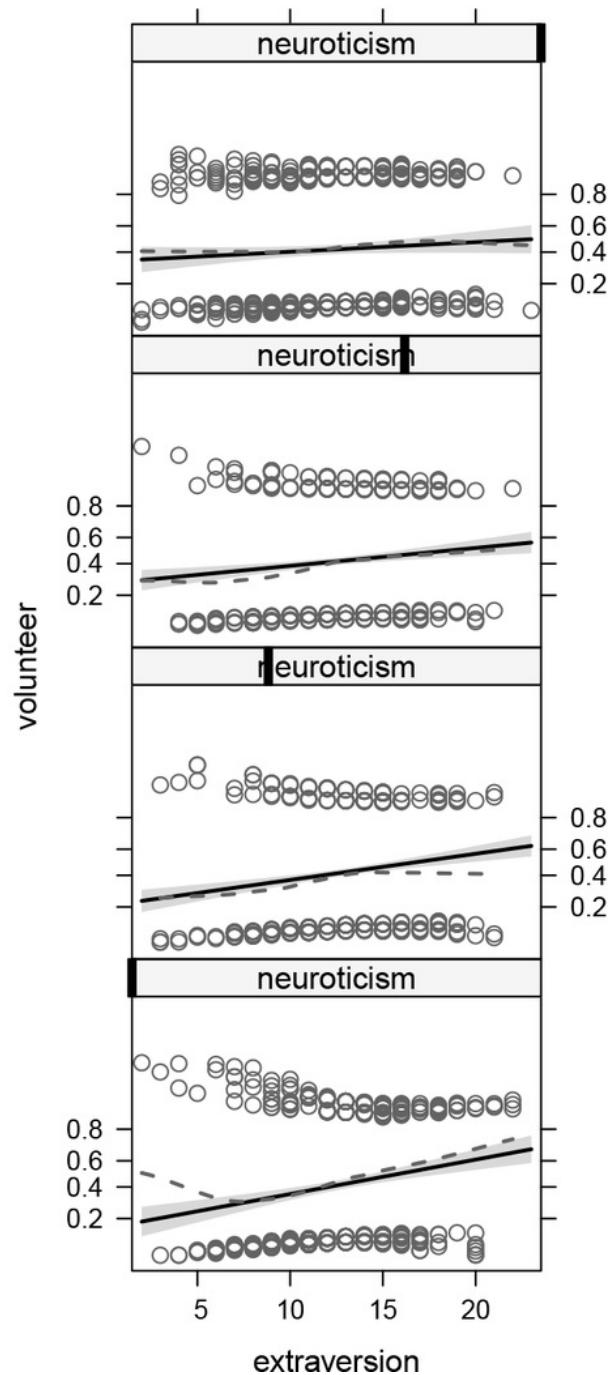
Computing partial residuals for the fixed effects in mixed models is a straightforward extension of the computations for linear and generalized linear models. The `crPlots()` function in the **car** package doesn't handle mixed-effects models, but the `predictorEffects()` function and the more basic and general `Effect()` function in the **effects** package do, for models fit by the `lmer()` and `glmer()` functions in the **lme4** package and for models fit by `lme()` in the **nlme** package.

**Figure 8.19** Predictor effect plots with partial residuals for neuroticism (left) and extraversion (right) in Cowles and Davis's logistic regression for volunteering, in which these two predictors are modeled with a linear-by-linear interaction.

**neuroticism predictor effect plot**



**extraversion predictor effect plot**



To illustrate these capabilities, we return to a linear mixed-effects model that we fit with `lme()` in [Section 7.2.6](#) to the Blackmore longitudinal data on exercise and eating disorders:

```
library("nlme")
Blackmore$tran.exercise <- bcnPower(Blackmore$exercise,
    lambda=0.25, gamma=0.1)
blackmore.mod.6.lme<- lme(tran.exercise ~ I(age - 8)*group,
```

```

random = ~ 1 | subject,
correlation = corCAR1(form = ~ I(age - 8) | subject),
data=Blackmore)
S(blackmore.mod.6.lme)

```

Linear mixed model fit by REML, Data: Blackmore

Fixed Effects:

Formula: tran.exercise ~ I(age - 8) \* group

|                         | Estimate | Std.Error | df  | t value | Pr(> t ) |
|-------------------------|----------|-----------|-----|---------|----------|
| (Intercept)             | -0.1962  | 0.1427    | 712 | -1.38   | 0.170    |
| I(age - 8)              | 0.0668   | 0.0231    | 712 | 2.89    | 0.004    |
| grouppatient            | -0.1569  | 0.1843    | 229 | -0.85   | 0.396    |
| I(age - 8):grouppatient | 0.1894   | 0.0290    | 712 | 6.52    | 1.3e-10  |

Random effects:

Formula: ~1 | subject  
           (Intercept) Residual  
 StdDev:       0.884       1.12

Correlation Structure: Continuous AR(1)

Formula: ~I(age - 8) | subject

Parameter estimate(s):

Phi

0.664

Number of Observations: 945

Number of Groups: 231

| logLik  | df | AIC    | BIC    |
|---------|----|--------|--------|
| -1493.8 | 7  | 3001.6 | 3035.5 |

The fit is in the fourth-root transformed scale for the response, computed by bcnPower (), with fixed effects for age (with origin set to age 8, the start of the study), group ("patient" or "control"), and their interaction; a random intercept by subject; and continuous first-order autoregressive errors.

We obtain a component-plus-residual plot for the age  $\times$  group interaction just as

we would for a linear or generalized linear model, with the result appearing in [Figure 8.20](#):

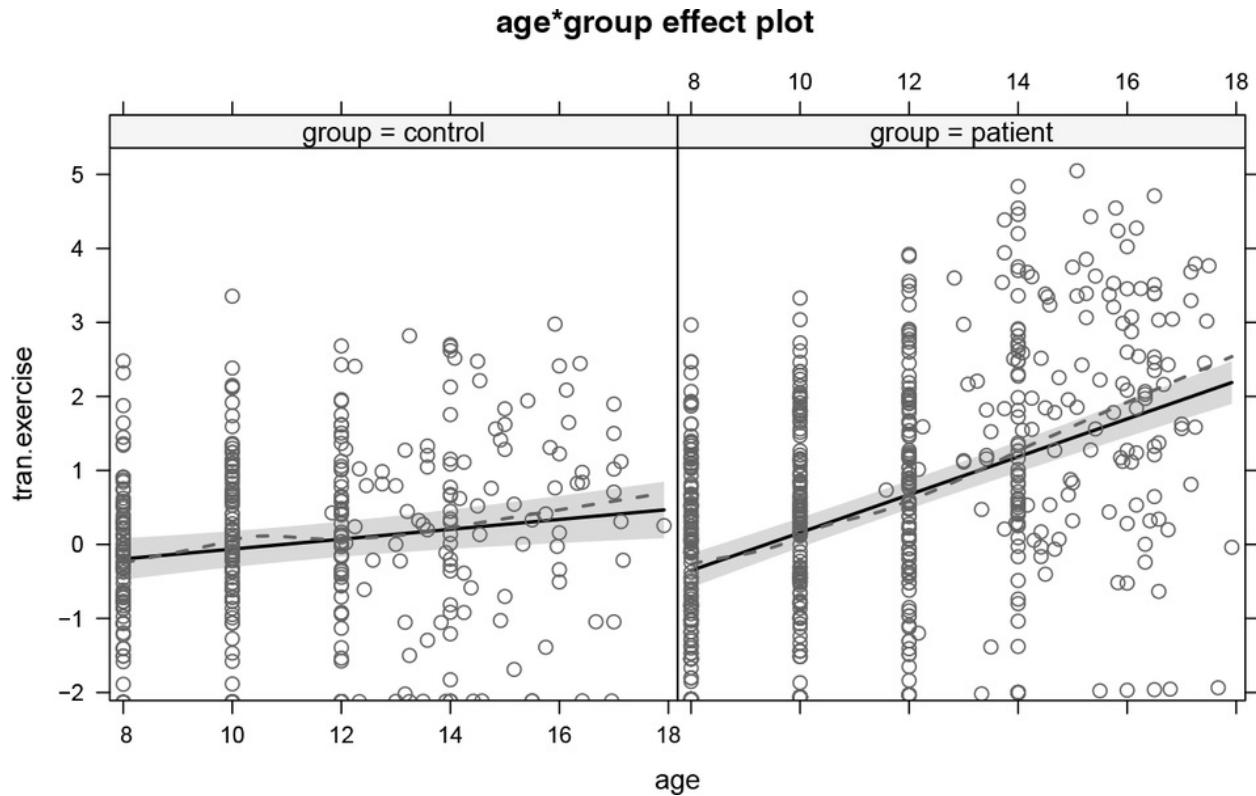
```
plot(Effect(c("age", "group"), blackmore.mod.6.lme,  
residuals=TRUE), partial.residual=list(lty="dashed"))
```

To display partial residuals, the plot must be in the default linear-predictor scale, so we do not untransform the response as we did in [Figure 7.10](#) (page 369). The earlier values of age in the graph are discrete because observations were taken at 2-year intervals up to the point of hospitalization for patients or the date of the interview for controls. Nonlinearity in the component-plus-residual plot is slight, supporting the specification of the model.

## 8.7.2 Influence Diagnostics for Mixed Models

In mixed-effects models, it is natural to focus deletion diagnostics on clusters of cases as well as potentially on individual cases. In linear models (as discussed in [Section 8.3](#)), we can efficiently compute the effect of deleting individual cases on statistics such as estimated regression coefficients without having to refit the model. Approximate influence diagnostics for generalized linear models (in [Section 8.6.2](#)) start with the maximum-likelihood estimates of the parameters of a model for the full data set and take one step toward the new maximum-likelihood estimates when a case is deleted, rather than the computationally more expensive fully iterated solution omitting each case.

**Figure 8.20** Component-plus-residual plot for the age  $\times$  group interaction in the linear mixed-effects model fit to the Blackmore data.



A similar approximation has been suggested for mixed-effects models by several authors (R. Christensen, Pearson, & Johnson, 1992; Demidenko & Stukel, 2005; Shi & Chen, 2008; Pan, Fie, & Foster, 2014), but not, to our knowledge, for models as general as those fit by the `lme()`, `lmer()`, and `glmer()` functions. In the `car` package, we therefore take a more computationally demanding approach, actually deleting subsets of cases corresponding to clusters and refitting the model.<sup>20</sup>

<sup>20</sup> The `influence.ME` package (Nieuwenhuis, te Grotenhuis, & Pelzer, 2012) takes a similar approach, but our implementation is a bit more general and offers some computational efficiencies: We accommodate models fit by `lme()` as well as those fit by `lmer()` and `glmer()`. We also start at the parameter estimates for the full data set, and, for models fit by `lmer()` or `glmer()`, we optionally allow the user to abbreviate the computation. Finally, we plan to provide parallel computations to take advantage of computers with multiple processors or cores (but have not yet implemented this feature).

Deletion diagnostics are computed with a method provided by the `car` package for the standard-R `influence()` generic function; for example, for the linear mixed-effects model fit to the Blackmore data:

```

system.time(inf.blackmore <-
  influence(blackmore.mod.6.lme, groups="subject"))

user   system elapsed
39.42     0.00   39.46

```

The `influence()` function refits the model deleting each of the 231 subjects (i.e., “clusters”) in turn, saving information in the returned object `inf.blackmore`, of class “`influence.lme`”. On our computer, the computation takes about a minute.

The `influenceIndexPlot()` function in the `car` package has a method for objects of class “`influence.lme`” and can create plots for both the fixed effects, displaying influence on the coefficients and Cook’s distances, and the random-effect parameters, shown respectively in [Figures 8.21](#) and [8.22](#):

**`influenceIndexPlot (inf.blackmore)`**

**`influenceIndexPlot (inf.blackmore, var="var.cov.comps")`**

Comparing the extreme values plotted in the panels of [Figure 8.21](#) to the magnitudes of the corresponding estimated fixed-effect parameters suggests that none of the subjects have substantial influence on the estimated fixed effects. In the Blackmore data set, the cases for patients appear before those for controls. Because the dummy regressor for group is coded 1 for patients and zero for controls, the (Intercept) parameter in the model is the intercept for controls, and the I (age - 8) parameter is the slope for controls. As a consequence, patients have no influence on these two fixed-effect coefficients, explaining the initial strings of zero values in the first two panels of [Figure 8.21](#).

The designation of the random-effect parameters in [Figure 8.22](#) requires a bit more explanation: The panel with vertical axis labeled `reStruct.subject` displays influence on the estimated standard deviation of the random intercepts, the panel with axis labeled `corStruct` displays influence on the estimated autoregression parameter for the errors, and the panel with axis labeled `lSigma` shows influence on the estimated *log* of the residual standard deviation. Noting that the latter is  $\log(\sigma) = \log(1.123) = 0.116$  for the full data, we can see that none of the subjects have much influence on the random-effect parameter estimates.

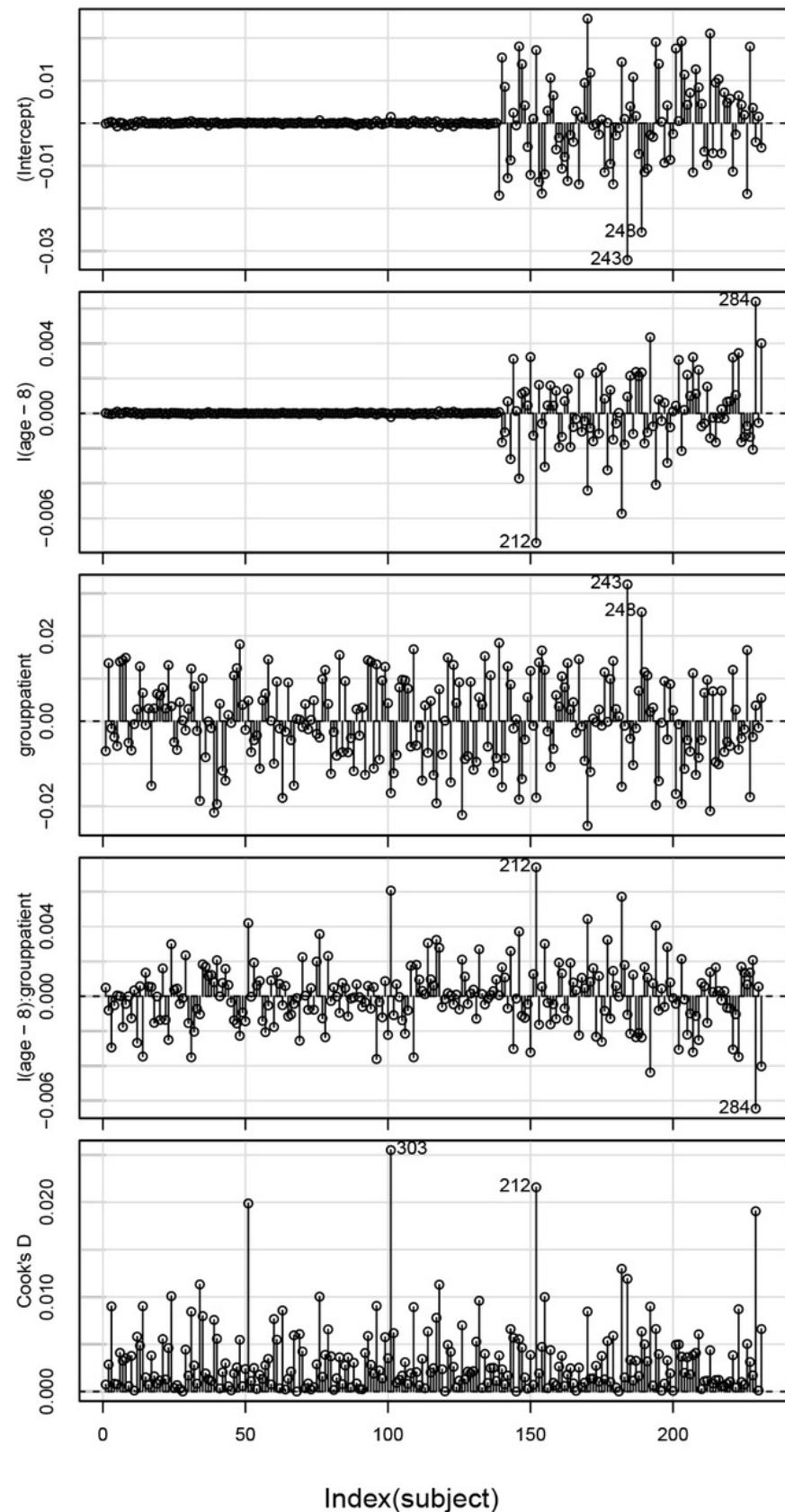
For more information on the influence diagnostics provided by the **car** package for mixed models, see help ("influence.mixed.models") and help ("influenceIndexPlot").

## 8.8 Collinearity and Variance Inflation Factors

When there are strong linear relationships among the predictors in a linear model, the precision of the estimated regression coefficients declines compared to what it would have been were the predictors uncorrelated with each other. Other important aspects of regression analysis beyond coefficients, such as prediction, may be much less affected by collinearity (as discussed in Weisberg, 2014, Sections 4.1.5 and 10.1, and Fox, 2016, chap. 13).

**Figure 8.21** Cluster deletion diagnostics for the subjects in the linear mixed-effect model fit to the Blackmore data: influence on the fixed effects coefficients, including Cook's distances.

## Diagnostic Plots



The estimated sampling variance of the  $j$ th regression coefficient may be written

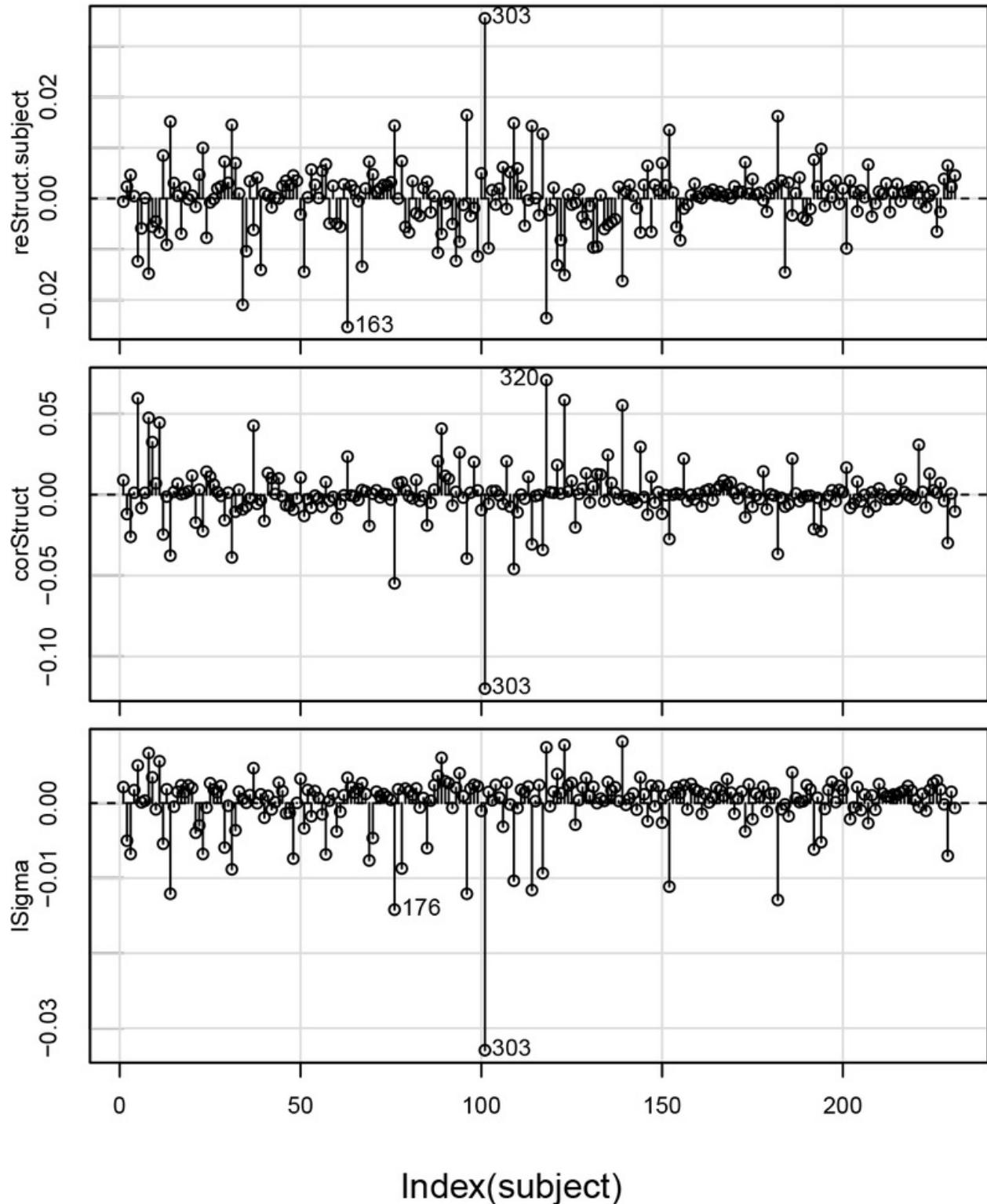
$$\widehat{\text{Var}}(b_j) = \frac{\widehat{\sigma}^2}{(n - 1)s_j^2} \times \frac{1}{1 - R_j^2}$$

as

where  $\widehat{\sigma}^2$  is the estimated error variance,  $s_j^2$  is the sample variance of  $x_j$ , and  $R_j^2$  is called the *variance inflation factor* (VIF <sub>$j$</sub> ) for  $b_j$ , is a function of the multiple correlation  $R_j$  from the regression of  $x_j$  on the other  $xs$ . The variance inflation factor is a simple measure of the harm produced by collinearity: The square root of the VIF indicates how much the confidence interval for  $\beta_j$  is expanded relative to similar uncorrelated data, were it possible for such data to exist, as would be the case, for example, in a designed experiment. If we wish to explicate the collinear relationships among the predictors, then we can examine the coefficients from the regression of each predictor with a large VIF on the other predictors.

**Figure 8.22** Cluster deletion diagnostics for the subjects in the linear mixed-effect model fit to the Blackmore data: influence on the estimated random-effects parameters. The top panel is for the standard deviation of the subject-specific intercepts, the middle panel is for the estimated autoregression parameter for the errors, and the bottom panel is for the log of the residual standard deviation.

## Diagnostic Plots



The variance inflation factor is not straightforwardly applicable to sets of related

regressors for multiple-degree-of-freedom effects, such as polynomial regressors, spline regressors, or contrasts constructed to represent a factor. Fox and Monette (1992) generalize the notion of variance inflation by considering the relative size of the joint confidence region for the coefficients associated with a related set of regressors. The resulting measure is called a *generalized variance inflation factor* or GVIF.<sup>21</sup> If there are  $p$  regressors in a term, then  $\text{GVIF}^{1/2p}$  is a one-dimensional expression of the decrease in precision of estimation due to collinearity, analogous to taking the square root of the usual variance inflation factor. When  $p = 1$ , the GVIF reduces to the VIF.

<sup>21</sup>\* Let  $\mathbf{R}_{11}$  represent the correlation matrix among the regressors in the set in question,  $\mathbf{R}_{22}$  the correlation matrix among the other regressors in the model, and  $\mathbf{R}$  the correlation matrix among all of the regressors in the model. Fox and Monette show that the squared area, volume, or hypervolume of the ellipsoidal joint confidence region for the coefficients in either set is expanded by the

$$\text{GVIF} = \frac{\det \mathbf{R}_{11} \det \mathbf{R}_{22}}{\det \mathbf{R}}$$

generalized variance inflation factor

relative to similar data in which the two sets of regressors are uncorrelated with each other. This measure is independent of the bases selected to span the subspaces of the two sets of regressors and so, for example, is independent of the contrast-coding scheme employed for a factor.

The `vif()` function in the **car** package calculates variance inflation factors for the terms in a linear model. When each term has one degree of freedom, the VIFs are returned; otherwise the GVIFs are calculated.

As a first example, consider the data on the 1980 U.S. Census undercount in the data frame Erickson (Erickson, Kadane, & Tukey, 1989):

### **summary(Erickson)**

| minority      | crime         | poverty       |          |
|---------------|---------------|---------------|----------|
| Min. : 0.70   | Min. : 25.0   | Min. : 6.80   |          |
| 1st Qu.: 5.03 | 1st Qu.: 48.0 | 1st Qu.: 9.95 |          |
| Median :15.80 | Median : 55.0 | Median :12.50 |          |
| Mean :19.44   | Mean : 63.1   | Mean :13.47   |          |
| 3rd Qu.:28.20 | 3rd Qu.: 73.0 | 3rd Qu.:16.60 |          |
| Max. :72.60   | Max. :143.0   | Max. :23.90   |          |
| language      | highschool    | housing       | city     |
| Min. : 0.20   | Min. :17.5    | Min. : 7.0    | city :16 |
| 1st Qu.: 0.50 | 1st Qu.:27.4  | 1st Qu.: 9.4  | state:50 |
| Median : 0.85 | Median :31.6  | Median :11.5  |          |
| Mean : 1.93   | Mean :33.6    | Mean :15.7    |          |
| 3rd Qu.: 2.35 | 3rd Qu.:41.7  | 3rd Qu.:20.3  |          |
| Max. :12.70   | Max. :51.8    | Max. :52.1    |          |
| conventional  | undercount    |               |          |
| Min. : 0.00   | Min. :-2.31   |               |          |
| 1st Qu.: 0.00 | 1st Qu.: 0.32 |               |          |
| Median : 0.00 | Median : 1.45 |               |          |
| Mean : 11.73  | Mean : 1.92   |               |          |
| 3rd Qu.: 9.75 | 3rd Qu.: 3.31 |               |          |
| Max. :100.00  | Max. : 8.18   |               |          |

These variables describe 66 areas of the United States, including 16 major cities, the 38 states without major cities, and the “remainders” of the 12 states that contain the 16 major cities. The following variables are included:

- minority: percentage of residents who are black or Hispanic
- crime: serious crimes per 1000 residents
- poverty: percentage of residents who are poor
- language: percentage having difficulty speaking or writing English
- highschool: percentage of those 25 years of age or older who have *not* finished high school
- housing: percentage of dwellings in small, multiunit buildings
- city: a factor with levels "state" and "city"

- conventional: percentage of households counted by personal enumeration (rather than by mail-back questionnaire with follow-ups)
- undercount: the estimated percentage undercount (with negative values indicating an estimated overcount)

The Erickson data set figured in a U.S. Supreme Court case concerning correction of the Census undercount.

We regress the Census undercount on the other variables in the data set:

```
mod.census <- lm(undercount ~ ., data=Ericksen)
brief(mod.census, pvalues=TRUE)
```

|  | (Intercept) | minority | crime     | poverty      | language |
|--|-------------|----------|-----------|--------------|----------|
| Estimate                                       | -0.611      | 0.079834 | 0.0301    | -0.1784      | 0.2151   |
| Std. Error                                     | 1.721       | 0.022609 | 0.0130    | 0.0849       | 0.0922   |
| Pr(> t )                                       | 0.724       | 0.000827 | 0.0241    | 0.0401       | 0.0232   |
|  | highschool  | housing  | citystate | conventional |          |
| Estimate                                       | 0.0613      | -0.0350  | -1.160    |              | 0.036989 |
| Std. Error                                     | 0.0448      | 0.0246   | 0.771     |              | 0.009253 |
| Pr(> t )                                       | 0.1764      | 0.1613   | 0.138     |              | 0.000186 |
| Residual SD = 1.43 on 57 df, R-squared = 0.708 |             |          |           |              |          |

The dot (.) on the right-hand side of the model formula represents all of the variables in the Erickson data frame with the exception of the response, undercount.

Checking for collinearity, we see that the three coefficients for minority, poverty, and highschool have variance inflation factors exceeding 4, indicating that confidence intervals for these coefficients are more than twice as wide as they would be for uncorrelated predictors:

```
vif(mod.census)
```

|            |         |         |              |
|------------|---------|---------|--------------|
| minority   | crime   | poverty | language     |
| 5.0091     | 3.3436  | 4.6252  | 1.6356       |
| highschool | housing | city    | conventional |
| 4.6192     | 1.8717  | 3.5378  | 1.6913       |

To illustrate the computation of generalized variance inflation factors, we return to Ornstein's least-squares interlocking-directorate regression (fit on page 408), where it turns out that collinearity is relatively slight:

### **vif(mod.ornstein)**

|             | GVIF   | Df | GVIF^(1/(2*Df)) |
|-------------|--------|----|-----------------|
| log(assets) | 1.9087 | 1  | 1.3816          |
| nation      | 1.4434 | 3  | 1.0631          |
| sector      | 2.5968 | 9  | 1.0544          |

The vif () function can also be applied to generalized linear models,<sup>22</sup> such as the quasi-Poisson regression model fit to Ornstein's data (page 423):

### **vif(mod.ornstein.qp.2)**

|              | GVIF   | Df | GVIF^(1/(2*Df)) |
|--------------|--------|----|-----------------|
| log2(assets) | 2.6171 | 1  | 1.6178          |
| nation       | 1.6196 | 3  | 1.0837          |
| sector       | 3.7178 | 9  | 1.0757          |

<sup>22</sup> Thanks to a contribution from Henric Nilsson.

The same approach works for the fixed-effect coefficients in linear and generalized linear mixed models.

Other, more complex, approaches to collinearity include principal-components analysis of the predictors or standardized predictors and singular-value decomposition of the model matrix or the mean-centered model matrix. These,

too, are simple to implement in R: See the `princomp()`, `prcomp()`, `svd()`, and `eigen()` functions (the last two of which are discussed in [Section 10.3](#)).

## 8.9 Additional Regression Diagnostics

There are several regression-diagnostics functions in the `car` package that we don't describe in this chapter:

`boxCoxVariable()`: Computes a *constructed variable* (Atkinson, 1985) for the Box-Cox transformation of the response variable in a linear model. The constructed variable is added to the regression equation, and an added-variable plot for the constructed variable in the augmented regression then provides a visualization of leverage and influence on the determination of the normalizing power transformation of the response.

`boxTidwell()`: Fits the model , for positive  $xs$ , estimating the linearizing power transformation parameters  $\lambda_1, \dots, \lambda_k$  by maximum likelihood using an algorithm suggested by Box and Tidwell (1962). Not all of the regressors ( $xs$ ) in the model need be candidates for transformation. Constructed variables for the Box-Tidwell power transformations are simply  $x_j \log(x_j)$ ; added-variable plots for the constructed variables, added to the linear regression of  $y$  on the  $xs$ , visualize leverage and influence on the determination of the linearizing transformations .

`durbinWatsonTest()`: Computes autocorrelations and generalized Durbin-Watson statistics (Durbin & Watson, 1950, 1951), along with their bootstrapped  $p$ -values, for the residuals from a linear model fit to time-series data. The `durbinWatsonTest()` function is discussed in the online appendix to the *R Companion* on time-series regression.

`inverseResponsePlot()`: For a linear model, this function provides a visual method for selecting a normalizing response transformation. The function draws an *inverse response plot* (Weisberg, 2014), with the response  $y$  on the vertical axis and the fitted values on the horizontal axis, and uses nonlinear least squares to estimate the power  $\lambda$  in the equation , adding the fitted curve to the graph.

`leveneTest()`: Computes Levene's test (see Conover, Johnson, & Johnson, 1981) for homogeneity of variance across groups defined by one or more factors in an analysis of variance.

`spreadLevelPlot()`: Applied to a linear model, `spreadLevelPlot()` generalizes Tukey's *spread-level plot* for exploratory data analysis (Tukey,

1977), plotting log absolute studentized residuals versus log fitted values (see Fox, 2016, Section 12.2). A positive relationship in the spread-level plot indicates a tendency of residual variation to increase with the magnitude of the response, and the slope of a line fit to the plot (say  $b$ ) can be used to select a variance-stabilizing power transformation (the  $1 - b$  power). The spread-level plot requires a positive response and positive fitted values; negative fitted values are ignored.

`yjPower ()`: This function, which implements a method due to Yeo and Johnson (2000), provides an alternative to the `bcnPower ()` family of modified power transformations when there are zero or negative values in the variable to be transformed.

## 8.10 Complementary Reading and References

Residuals and residual plots for linear models are discussed in Weisberg (2014, [Sections 9.1–9.2](#)). Added-variable plots are discussed in Weisberg (2014, [Section 3.1](#)). Outliers and influence are taken up in Weisberg (2014, chap. 9).

Diagnostics for unusual and influential data are described in Fox (2016, chap. 11); for nonnormality, nonconstant error variance, and nonlinearity in Fox (2016, chap. 12); and for collinearity in Fox (2016, chap. 13).

Diagnostics for generalized linear models are taken up in Fox (2016, Section 15.4).

A general treatment of residuals in models without additive errors, which expands on the discussion in [Section 8.6.1](#), is given by Cox and Snell (1968).

For further information on various aspects of regression diagnostics, see R. D. Cook and Weisberg (1982, 1994, 1997, 1999), Fox (1991), R. D. Cook (1998), and Atkinson (1985).

# 9 Drawing Graphs

John Fox and Sanford Weisberg

One of the strengths of R is its ability to produce high-quality statistical graphs. R's strength in graphics reflects the origin of S at Bell Labs, long a center for innovation in statistical graphics.

We find it helpful to make a distinction between *analytical graphics* and *presentation graphics* (see, e.g., Weisberg, 2004). At many places in the *R Companion*, we describe various analytical graphs, which are plots designed for discovery of patterns in data, to help the analyst better understand the data. These graphs should be easy to draw and interpret, and they quite likely will be discarded when the analyst moves on to the next graph. The **car** package includes a number of functions for this very purpose, such as `scatterplot()`, `scatterplotMatrix()`, `residualPlots()`, and `avPlots()` (see, in particular, [Chapters 3](#) and [8](#)). A desirable feature of analytical graphs is the ability of the analyst to interact with them, by identifying points, removing points to see how a model fit to the data changes, and so on. Standard R graphs allow only limited interaction, for example, via the `identify()` function, a limitation that we view as a deficiency in the current R system (but see the discussion of other graphics packages in R in [Section 9.3](#)).

Presentation graphs have a different goal and are more likely to be published in reports, in journals, online, and in books, and then examined by others. While in some instances, presentation graphs are nothing more than carefully selected analytical graphs (for example, an effect plot), they can be much more elaborate, with more attention paid to the design of the graph, where representing the data with clarity and simplicity is the overall goal. Although the default graphs produced by R are often useful and aesthetically pleasing, they may not meet the requirements of a publisher. R has a great deal of flexibility for creating presentation graphs, and that is the principal focus of this chapter.

There are many useful and sophisticated kinds of graphs that are readily available in R. Frequently, there is a `plot()` method to produce a standard graph or set of graphs for objects of a given class—try the `plot()` command with a "data.frame" or an "lm" (linear-model) object as the argument, for example. In some cases, `plot()` produces useful analytical graphs (e.g., when applied to a linear model), and in others it produces presentation graphs (e.g., when applied to a *regression tree* computed with the **rpart** package, Therneau, Atkinson, & Ripley, 2018).

Standard R graphics are based on a simple metaphor: Creating a standard R graph is like drawing with a pen, in indelible ink, on a sheet of paper. We

typically create a simple graph with a function like `plot()` and build more elaborate graphs by adding to the simple graph, using functions like `lines()`, `points()`, and `legend()`. Apart from resizing the graphics window in the usual manner by dragging a side or corner with the mouse, once we put something in the graph, we can't remove it—although we can draw over it. In most cases, if we want to change a graph, we must redraw it. This metaphor works well for creating presentation graphs, because the user can exert control over every step of the drawing process; it works less well when ease of construction is paramount, as in data-analytic graphics. The ink-on-paper metaphor is thus both a strength and weakness of traditional R graphics. [Section 9.1](#) describes a general approach to constructing custom graphs in R based on the ink-on-paper metaphor.

In [Section 9.2](#), we develop an illustrative complex graph in detail—a multipanel plot that explains how local linear nonparametric regression works—using the tools introduced in the preceding section. We also show how to control the layout of multipanel graphs.

Because R is an open system, it is no surprise that other approaches to statistical graphics have also been developed for it, most important, the **lattice** and **ggplot2** packages. We briefly introduce these packages, along with some others, in [Section 9.3](#).

This chapter and the following chapter on programming deal with general matters, and we have employed many of the techniques described here in the earlier chapters of the *R Companion*. Rather than introducing general material near the beginning of the book, however, we prefer to regard previous examples of customized R graphs as background and motivation.

## 9.1 A General Approach to R Graphics

For the most part, the discussion in this chapter is confined to two-dimensional coordinate plots, and a logical first step in drawing such a graph is to define a coordinate system. Sometimes that first step will include drawing axes and axis labels on the graph, along with a rectangular frame enclosing the plotting region and possibly other elements such as points and lines; sometimes, however, these elements will be omitted or added in separate steps to assert greater control over what is plotted. The guts of the graph generally consist of plotted points, lines, text, and, occasionally, shapes, arrows, and other features.

Our general approach is to draw as much of the graph as possible in an initial `plot()` command and then to add elements to the initial graph. The current section describes, in a general way, how to perform these tasks.

### 9.1.1 Defining a Coordinate System: `plot()`

The `plot()` function is generic:<sup>1</sup> There are therefore really many `plot()` functions, and the method function that is invoked depends on the arguments that are passed to `plot()`—specifically on the class of the first argument. If the first argument is a numeric vector—for example, the numeric vector `x` in the command `plot(x, y)`—then the default method, the function `plot.default()`, is invoked. If the first argument is a linear-model object (i.e., an object of class "lm"), then the method `plot.lm()` is called rather than the default method. If the first argument is a formula—for example, `plot(y ~ x)`—then the method `plot.formula()` is used, which simply decodes arguments like `formula`, `data`, and `subset` to set up a suitable call to `plot.default()`.

<sup>1</sup> See Sections 1.7 and 10.9 for an explanation of how generic functions and their methods work in R.

The default `plot()` method can be employed to make a variety of point and line graphs; `plot()` can also be used to define a coordinate space, which is our main reason for discussing it here. The list of arguments to `plot.default()` is a reasonable point of departure for understanding how to use the traditional R graphics system:

```
args("plot.default")  
  
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,  
log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,  
ann = par("ann"), axes = TRUE, frame.plot = axes,  
panel.first = NULL, panel.last = NULL, asp = NA, ...)  
NULL
```

To see in full detail what the arguments mean, consult `help("plot.default")`;<sup>2</sup> the following points are of immediate interest, however:

The first two arguments, `x` and `y`, can provide, respectively, the horizontal and vertical coordinates of points or lines to be plotted and also implicitly define a data coordinate system for the graph. The argument `x` is required, and all other arguments—including `y`—are optional. In constructing a complex graph, a good initial step is often to use `x` and `y` to establish the ranges for the axes. If, for example, we want horizontal coordinates to range from -10 to 10, and vertical coordinates to range from 0 to 1, then the initial command

```
plot(c(-10, 10), c(0, 1), type="n", xlab="", ylab="")
```

is sufficient to set up the coordinate space for the plot, as we explain in more detail shortly.

The argument `type`, naturally enough, determines the type of graph to be drawn, of which there are several: The default type, "`p`", plots points at the

coordinates specified by `x` and `y`. The character used to draw the points is given by the argument `pch`, and `pch` can designate a vector of characters of the same length as `x` and `y`, which may therefore be different for each point. Specifying `type="l"` (the letter “el”) produces a line graph, and specifying `type="n"`, as in the command above, sets up the plotting region to accommodate the data but plots nothing. Other types of graphs include “`b`”, both points and lines; “`o`”, points and lines overlaid; “`h`”, “histogram-like” vertical lines; and “`s`” and “`S`”, staircase-like lines, starting horizontally and vertically, respectively.

The arguments `xlim` and `ylim` may be used to define the limits of the horizontal and vertical axes; these arguments are usually unnecessary because R will pick reasonable limits from `x` and `y`, but they provide an additional measure of control over the graph. For example, extending the limits of an axis can provide room for a legend or other explanatory text, and contracting the limits can cause some data to be omitted from the graph. If we are drawing several graphs, we may want all of them to have the same range for one or both axes, and that can also be accomplished with the arguments `xlim` and `ylim`.

The `log` argument makes it easy to define logarithmic axes: `log="x"` produces a logged horizontal axis; `log="y"`, a logged vertical axis; and `log="xy"` (or `log="yx"`), logged axes for both variables. Base-10 logarithms are used, and the conversion from data values to their logs is automatic. `xlab` and `ylab` take character-string or *expression* arguments, which are used to label the axes. An expression can be used to produce mathematical notation in labels, such as superscripts, subscripts, mathematical symbols, and Greek letters.<sup>3</sup> Similarly, the argument `main` may be used to place a title above the plot, or the `title()` function may be called subsequently to add main or axis titles. The default axis label, `NULL`, is potentially misleading, in that by default `plot()` constructs labels from the arguments `x` and `y`. To suppress the axis labels, either specify empty labels (e.g., `xlab=""`) or set `ann=FALSE`.

Setting `axes=FALSE` and `frame.plot=FALSE` respectively suppresses drawing axes and a box around the plotting region. A frame can subsequently be added by the `box()` function, and customized axes can be added using the `axis()` function.

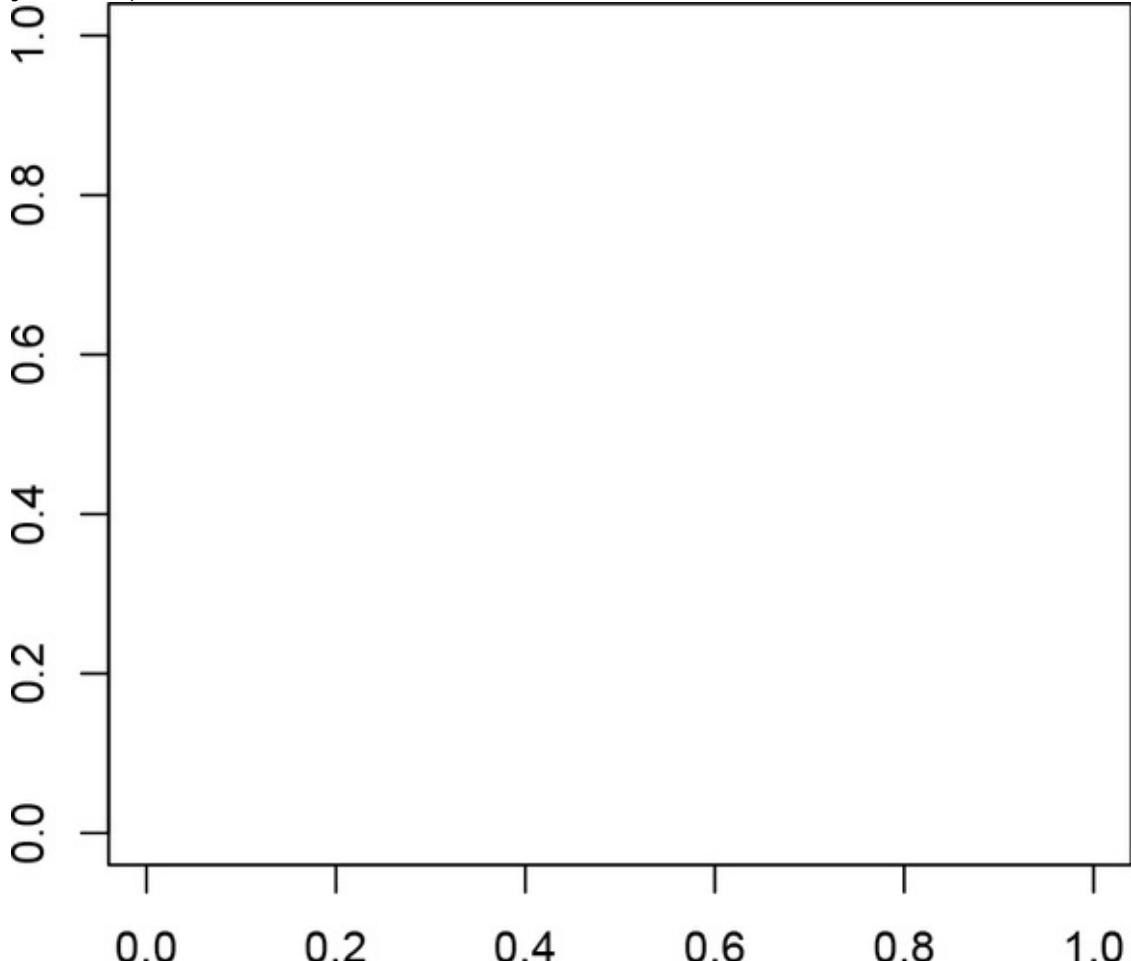
The `col` argument may be used to specify the color (or colors) for the points and lines drawn on the plot. (Color specification in R is described in [Section 9.1.4](#).)

`cex` (for character expansion) specifies the relative size of points in the

graph; the default size is `cex=1`; `cex` may be a vector, controlling the size of each point individually.

The arguments `lty` and `lwd` select the type and width of lines drawn on the graph. (See [Section 9.1.3](#) for more information on drawing lines.)

**Figure 9.1** Empty plot, produced by `plot(c(0, 1), c(0, 1), type="n", xlab="", ylab="")`.



[2](#) In this chapter, we will not discuss all of the arguments of the graphics functions that we describe. Details are available in the documentation for the various graphics functions, in the sources that we cite, and in the references at the end of the chapter. With a little patience and trial-and-error, we believe that most readers of this book will be able to master the subtleties of R documentation.

[3](#) The ability of R to typeset mathematics in graphs is both useful and unusual; for details, see `help("plotmath")` and Murrell and Ihaka (2000); also see [Figures 9.11](#) (page 451) and [9.13](#) (page 456) for examples.

For example, the following command sets up the blank plot in [Figure 9.1](#), with axes and a frame, but without axis labels:

```
plot(c(0, 1), c(0, 1), type="n", xlab="", ylab="")
```

## 9.1.2 Graphics Parameters: par ()

Many of the arguments to plot (), such as pch and col, get defaults from the par () function if they are not specified explicitly in the call to plot (). The par () function is used to set and retrieve a variety of graphics parameters and thus behaves similarly to the options () function, which sets global defaults for R. For instance:

```
par("col")
```

```
[1] "black"
```

Consequently, unless their color is changed explicitly, all points and lines in a standard R graph are drawn in black. To change the general default plotting color to red, for example, we could enter the command par (col="red") (see [Section 9.1.4](#) on colors in R).

To print the current values of all of the plotting parameters, call par () with no arguments. Here are the names of all the graphics parameters:

```
names(par())
```

```
[1] "xlog"      "ylog"      "adj"       "ann"       "ask"        "bg"        "bty"       "cex"       "cex.axis"  "cex.lab"    "cex.main"   "cex.sub"    "cin"       "col"       "col.axis"  "col.lab"   "col.main"   "col.sub"    "cra"       "crt"       "csi"       "cxy"       "din"       "err"       "family"    "fg"        "fig"       "fin"       "font"     "font.axis" "font.lab"   "font.main"  "font.sub"   "lab"       "las"       "lheight"   "ljoin"    "lmitre"    "lty"       "lend"      "mai"       "mar"       "mex"       "mfcol"    "mfg"       "mfrow"    "mgp"       "mkh"       "new"       "oma"       "omd"       "omi"       "page"    "pch"       "pin"       "plt"       "ps"        "pty"       "smo"       "srt"       "tck"       "tcl"       "usr"       "xaxp"    "xaxs"      "xaxt"    "xpd"       "yaxp"    "yaxs"      "yaxt"    "ylbias"
```

[Table 9.1](#) presents brief descriptions of some of the plotting parameters that can be set or retrieved by par (); many of these can also be used as arguments to plot () and other graphics functions, but some, in particular the parameters that concern the layout of the plot window (e.g., mfrow), can only be set using par (), and others (e.g., usr) only return information and cannot be modified directly by the user. For complete details on the plotting parameters available in R, see help

("par").

It is sometimes helpful to change plotting parameters in `par()` temporarily for a particular graph or set of graphs and then to change them back to their previous values after the plot is drawn. For example,

```
oldpar <- par(lwd=2)
plot(x, y, type="l")
par(oldpar)
```

draws lines at twice their normal thickness. The original setting of `par("lwd")` is saved in the variable `oldpar`, and after the plot is drawn, `lwd` is reset to its original value.<sup>4</sup> Alternatively, and usually more simply, closing the current graphics-device window returns graphical parameters to their default values. In RStudio, we can clear the plotting device by pressing the “broom” button at the top of the *Plots* tab.

<sup>4</sup> This example is artificial because `plot(x, y, type="l", lwd=2)` also works: Setting the `lwd` parameter globally affects *all* subsequent plots; using the `lwd` argument to `plot()` affects only the *current* graph.

### 9.1.3 Adding Graphical Elements: `axis()`, `points()`, `lines()`, `text()`, et al.

Having defined a coordinate system, we typically want to add graphical elements, such as points and lines, to the plot. Several functions useful for this purpose are described in this section.

**Table 9.1**

| Parameter      | Default Value         | Purpose   |
|----------------|-----------------------|---|
| adj            | 0.5                   | text-string justification: 0 = left,<br>0.5 = centered, 1 = right   |
| ann            | TRUE                  | annotate graph (axes, etc.)   |
| cex            | 1                     | relative character expansion (size)   |
| col            | "black"               | default color   |
| las            | 0                     | orientation of axis labels: 0 = parallel to<br>axis, 1 = horizontal                                       |
| lty            | "solid"               | default line type: name or number   |
| lwd            | 1                     | default line width  |
| mar*           | c(5.1, 4.1, 4.1, 2.1) | plot margins in lines of text: bottom,<br>left, top, right  |
| mfcol*, mfrow* | c(1, 1)               | plot array, filled by columns or rows:<br>number of rows, columns   |
| fig*           | c(0, 1, 0, 1)         | coordinate of figure region as fractions<br>of plotting device: $x$ -min, $x$ -max,<br>$y$ -min, $y$ -max |
| oma*           | c(0, 0, 0, 0)         | outer margins in lines of text: bottom,<br>left, top, right   |
| new            | FALSE                 | if FALSE, next high-level plotting<br>function call clears plot   |
| pch            | 1                     | plotting symbol: number or character  |
| pin+           | current values        | size of plot in inches: width, height   |
| pty            | "m"                   | type of plotting region: "m" maximal;<br>"s" square   |
| srt            | 0                     | rotation of character strings, in degrees:<br>0 = horizontal  |
| usr+           | current values        | user coordinates, range of data region:<br>$x$ -min, $x$ -max, $y$ -min, $y$ -max                         |
| xpd            | FALSE                 | allow drawing outside the data region   |

## points () and lines ()

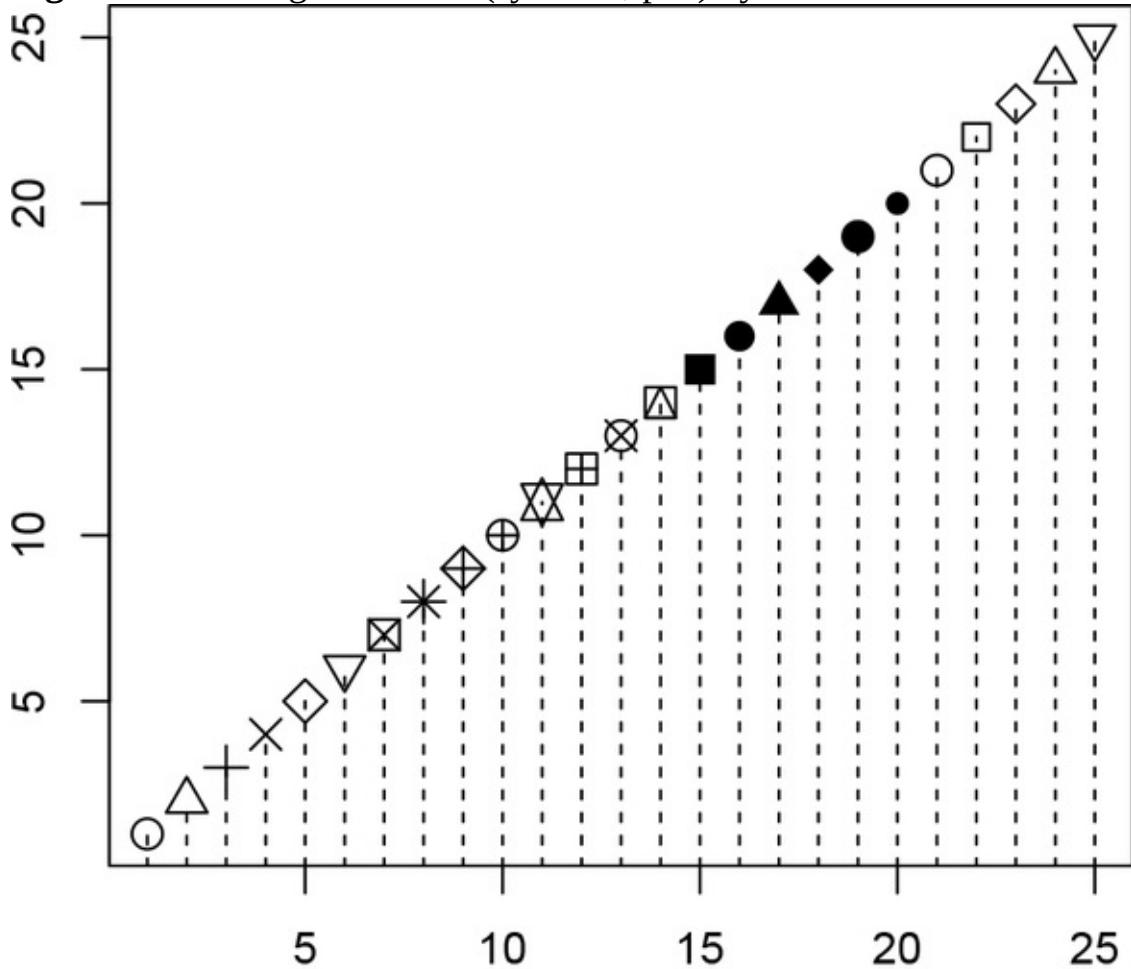
As you might expect, points () and lines () add points and lines to the current plot; either function can be used to plot points, lines, or both, but their default behavior is implied by their names. The pch argument is used to select the plotting character (symbol), as the following example (which produces [Figure 9.2](#)) illustrates:

```
plot (1:25, pch=1:25, cex=1.5, xlab="Symbol Number", ylab="")
lines (1:25, type="h", lty="dashed")
```

The plot () command graphs the symbols numbered 1 through 25; because the y argument to plot () isn't given, an *index plot* is produced, with the values of x on

the *vertical* axis plotted against their indices—in this case, also the numbers from 1 through 25. The argument `cex=1.5` (“character expansion”) sets the size of the plotted symbols to 1.5 times the default size. Finally, the `lines()` function draws broken vertical lines (selected by `lty="dashed"`; see below) up to the symbols; because `lines()` is given only one vector of coordinates, these too are interpreted as vertical coordinates, to be plotted against their indices as horizontal coordinates. Specifying `type="h"` draws spikes (or, in a questionable mnemonic, “histogram-like” lines) up to the points.

**Figure 9.2** Plotting characters (symbols, `pch`) by number.



### Symbol Number

One can also plot arbitrary characters, as the following example (shown in [Figure 9.3](#)) illustrates:

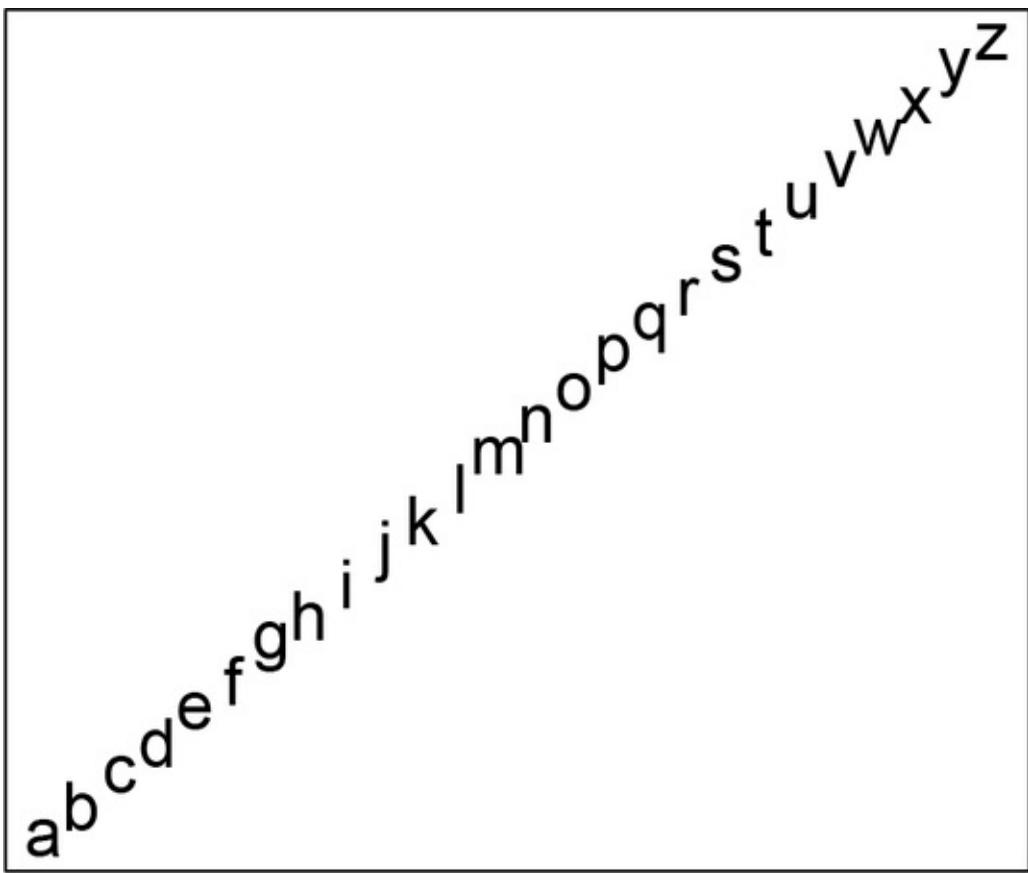
```
head(letters) # first 6 lowercase letters  
[1] "a" "b" "c" "d" "e" "f"  
  
plot(1:26, xlab="letters", ylab="", pch=letters, cex=1.5,  
axes=FALSE, frame.plot=TRUE)
```

Once again, ylab="" suppresses the vertical axis label, axes=FALSE suppresses tick marks and axes, and frame.plot=TRUE adds a box around the plotting region and is equivalent to entering the separate command box () after the plot () command.

As shown in [Figure 9.4](#), several different line types are available in R plots:

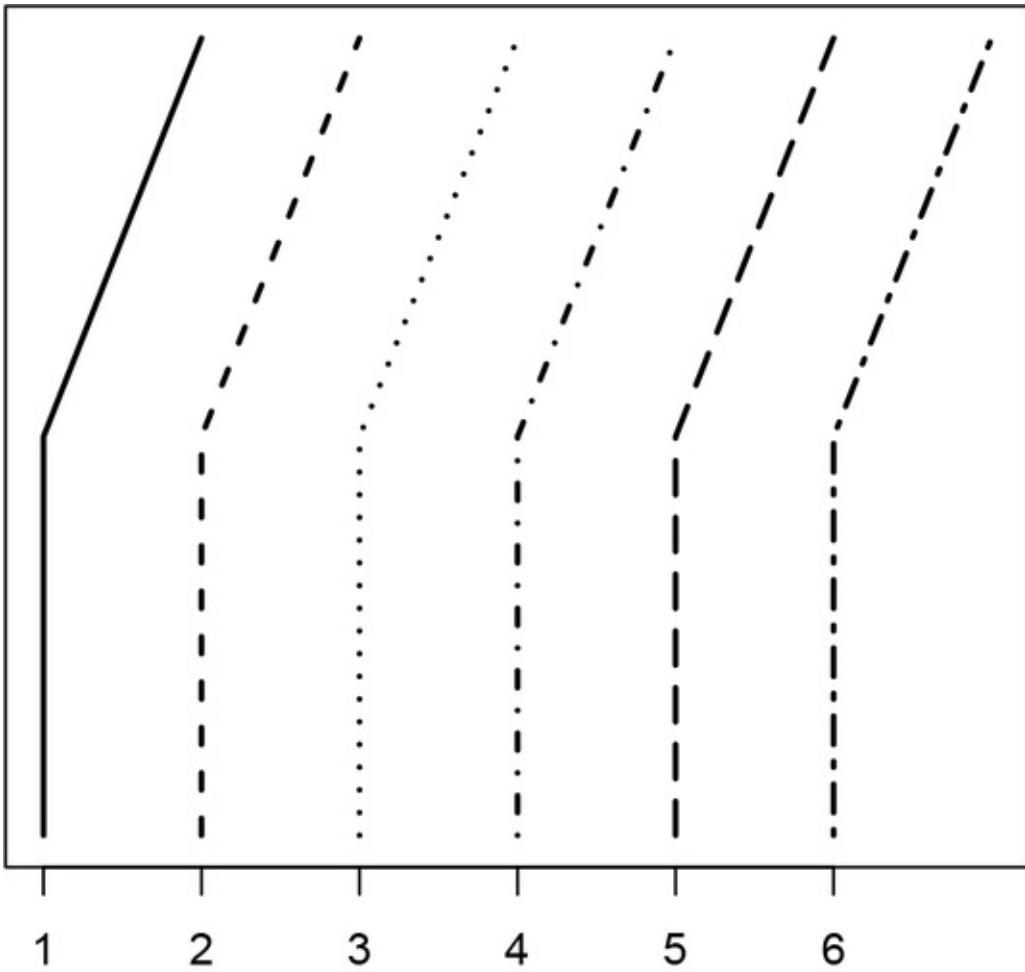
```
plot(c(1, 7), c(0, 1), type="n", axes=FALSE,  
      xlab="Line Type (lty)", ylab="", frame.plot=TRUE)  
axis(1, at=1:6) # x-axis  
for (lty in 1:6)  
  lines(c(lty, lty, lty + 1), c(0, 0.5, 1), lty=lty, lwd=2)
```

**Figure 9.3** Plotting characters—the lowercase letters.



letters

**Figure 9.4** Line types (lty), by number.

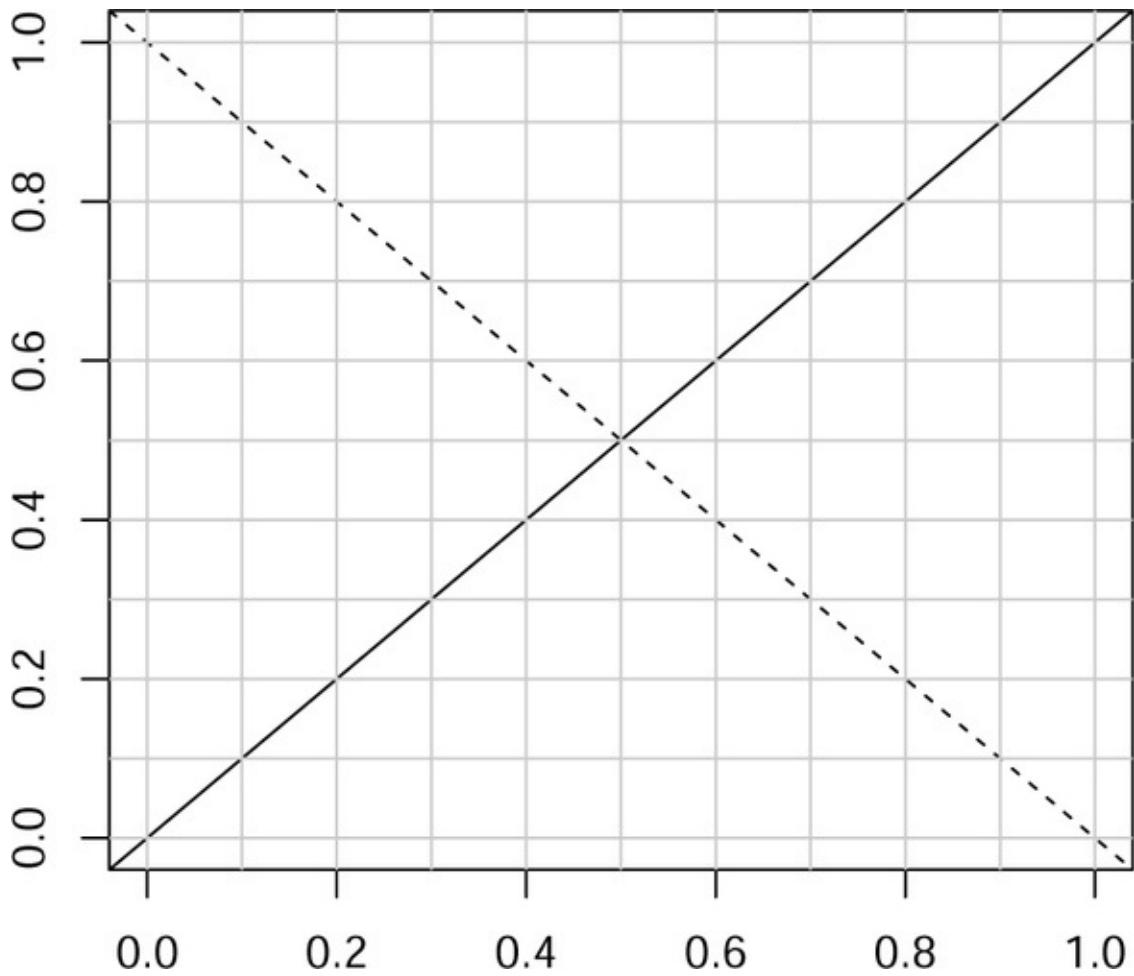


### Line Type (lty)

The `lines()` function connects the points whose coordinates are given by its first two arguments, `x` and `y`. If a coordinate is `NA`, then the line drawn is discontinuous. Line type (`lty`) may be specified by number (as here) or by name, such as "solid", "dashed", and so on.<sup>5</sup> Line width is similarly given by the `lwd` parameter, which defaults to 1. The exact effect varies according to the graphics device used to display the plot, but the general unit seems to be pixels: Thus, for example, `lwd=2` (as in [Figure 9.4](#)) specifies a line 2 pixels wide. We use a `for()` loop (see [Section 10.4.2](#)) to generate the six lines shown in [Figure 9.4](#).

<sup>5</sup> The names corresponding to line types 1 through 6 are "solid", "dashed", "dotted", "dotdash", "longdash", and "twodash"; setting `lty="blank"` suppresses the line.

**Figure 9.5** Lines created by the `abline()` function.



## **abline ()**

The `abline ()` function can be used to add straight lines to a graph. We describe several of its capabilities here; for details and other options, see help ("`abline`").

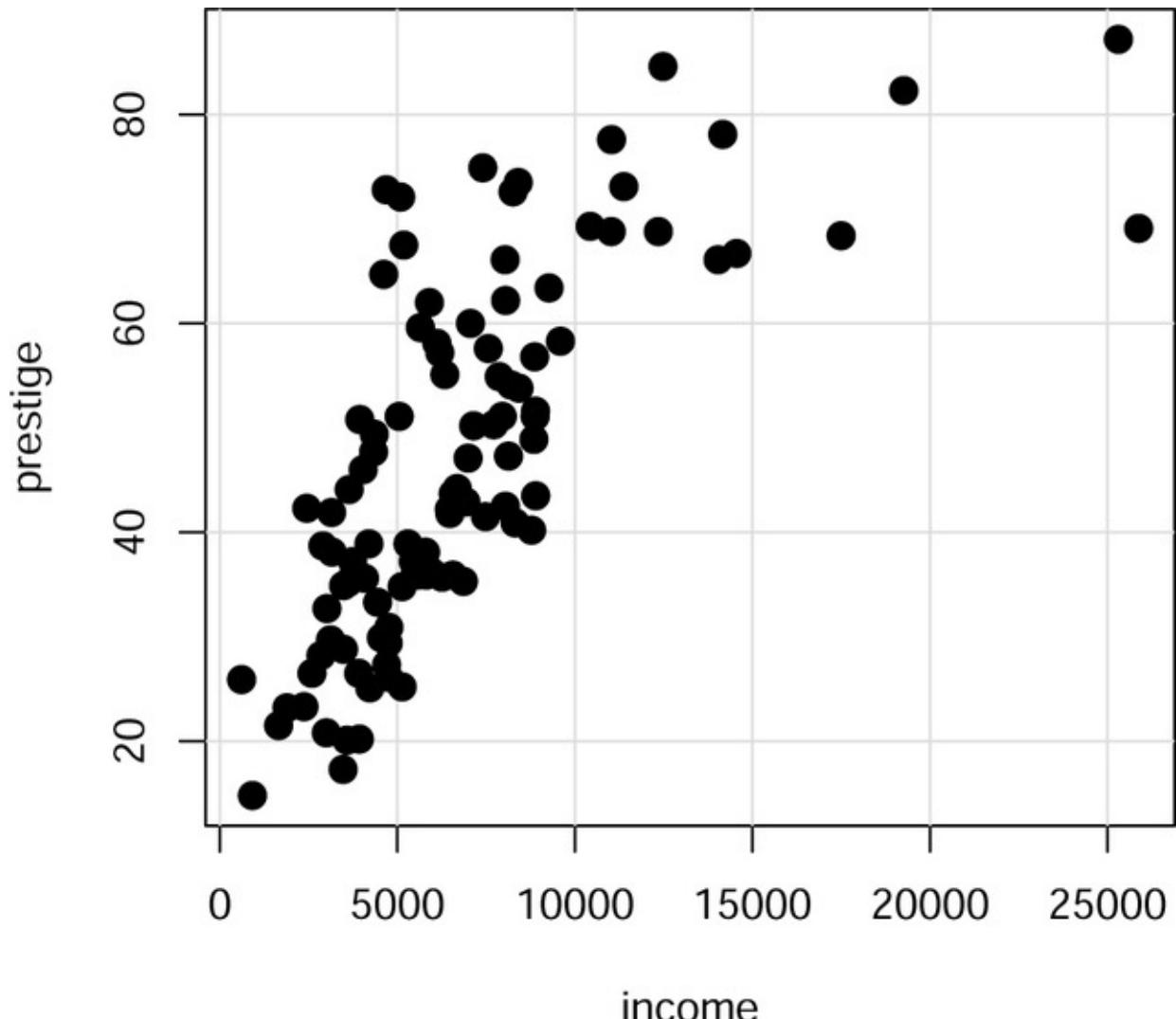
- Called with a simple regression model object as its argument—for example, `abline (lm (y ~ x))`—`abline ()` draws the regression line. If there's a single coefficient in the model, as in `abline (lm (y ~ x - 1))`, `abline ()` draws the regression line through the origin.
- Called with two numbers as arguments, as in `abline (a, b)`, or with a two-element numeric vector as its argument, as in `abline (c (a, b))`, `abline ()` draws a line with intercept  $a$  and slope  $b$ .
- Called with argument  $h$  or  $v$ , each of which can set to a single number or a numeric vector, `abline ()` draws horizontal or vertical lines at the specified  $y$ - or  $x$ -values.

[Figure 9.5](#), created by the following commands, illustrates the use of `abline ()`:

```
plot(c(0, 1), c(0, 1), type="n", xlab="", ylab="")
abline(0, 1)
abline(c(1, -1), lty="dashed")
abline(h=seq(0, 1, by=0.1), v=seq(0, 1, by=0.1), col="gray")
axis () and grid ()
```

In the plot () command (on page 444) for drawing [Figure 9.4](#), the argument axes=FALSE suppresses both the horizontal and vertical axis tick marks and tick labels. We use the axis () function to draw a customized horizontal (bottom) axis, but let the plot stand with no vertical (left) axis tick marks or labels. The first argument to axis () indicates the position of the axis: 1 corresponds to the bottom of the graph, 2 to the left side, 3 to the top, and 4 to the right side. The at argument controls the location of tick marks. There are several other arguments as well. Of particular note is the labels argument: If labels=TRUE (the default), then numerical labels are used for the tick marks; otherwise, labels takes a vector of character strings to provide tick labels (e.g., labels=c ("male", "female", "nonbinary") or labels=levels (gender) for the imaginary factor gender).

**Figure 9.6** Grid of horizontal and vertical lines created by the grid () function.



The `grid()` function can be used to add a grid of horizontal and vertical lines, typically at the default axis tick-mark positions (see help ("grid") for details). For example:

```
library("carData") # for data
plot(prestige ~ income, type="n", data=Prestige)
grid(lty="solid")
with(Prestige, points(income, prestige, pch=16, cex=1.5))
```

The resulting graph is shown in [Figure 9.6](#). In the call to `grid()`, we specify `lty="solid"` in preference to the default dotted lines. We are careful to plot the points *after* the grid, suppressing the points in the initial call to `plot()`. We invite the reader to see what happens if the points are plotted *before* the grid (hint: recall the ink-on-paper metaphor).

## **text () and locator ()**

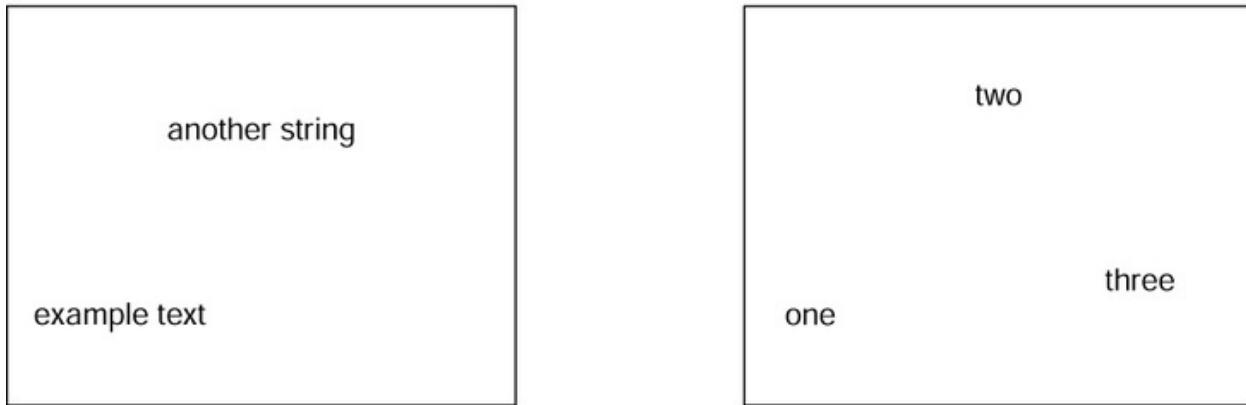
The `text()` function places character strings—as opposed to individual characters—on a plot; the function has several arguments that determine the position, size, and font that are used. For example, the following commands produce [Figure 9.7](#) (a):

```
plot(c(0, 1), c(0, 1), axes=FALSE, type="n", xlab="", ylab="",
      frame.plot=TRUE, main="(a)")
text(x=c(0.2, 0.5), y=c(0.2, 0.7),
      c("example text", "another string"))
```

**Figure 9.7** Plotting character strings with `text()`.

(a)

(b)



We sometimes find it helpful to use the `locator()` function along with `text()` to position text with the mouse; `locator()` returns a list with vectors of  $x$  and  $y$  coordinates corresponding to the position of the mouse cursor when the left button is clicked. [Figure 9.7](#) (b) is constructed as follows:

```
plot(c(0, 1), c(0, 1), axes=FALSE, type="n", xlab="", ylab="",
      frame.plot=TRUE, main="(b)")
text(locator(3), c("one", "two", "three"))
```

To position each of the three text strings, we move the mouse cursor to a point in the plot, and click the left button.

Called with no arguments, locator () returns pairs of coordinates corresponding to left-clicks until the right mouse button is pressed and *Stop* is selected from the resulting pop-up context menu (under Windows) or the esc key is pressed (under macOS). In RStudio, you can either use the Esc key or press the *Finish* button, which appears at the top of the RStudio *Plots* tab when the locator is active. Alternatively, you can indicate in advance the number of points to be returned as an argument to locator ()—locator (3) in the current example—in which case, control returns to the R command prompt after the specified number of left-clicks.

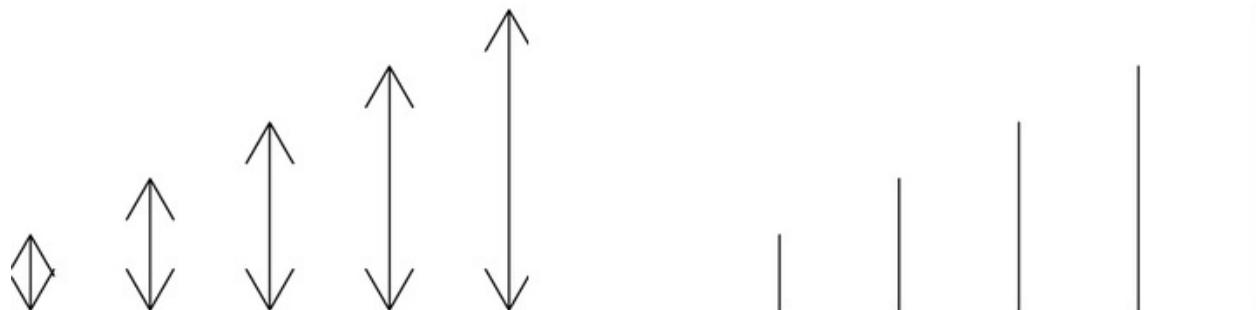
Another handy argument to text ()—not, however, used in these examples—is adj, which controls the horizontal justification of text: 0 specifies left-justification, 0.5 centering (the initial default, given by par ("adj")), and 1 right-justification. If two values are given, adj=c (x, y), then the second value controls vertical justification (0 = bottom, 0.5 = center, 1 = top).

Sometimes we want to add text outside the plotting area. The function mtext () can be used for this purpose; mtext () is similar to text (), except it writes in the margins of the plot. Alternatively, specifying the argument xpd=TRUE to text () or setting the global graphics option par (xpd=TRUE) also allows us to write outside of the normal plotting region.

**Figure 9.8** The arrows () and segments () functions.

(a) arrows

(b) segments



### arrows () and segments ()

As their names suggest, the arrows () and segments () functions may be used to add arrows and line segments to a plot. For example, the following commands produce [Figure 9.8](#):

```
plot(c(1, 5), c(0, 1), axes=FALSE, type="n",
      xlab="", ylab="", main="(a) arrows")
arrows(x0=1:5, y0=rep(0.1, 5),
       x1=1:5, y1=seq(0.3, 0.9, len=5), code=3)
```

```
plot(c(1, 5), c(0, 1), axes=FALSE, type="n",
      xlab="", ylab="", main="(b) segments")
segments(x0=1:5, y0=rep(0.1, 5),
         x1=1:5, y1=seq(0.3, 0.9, len=5))
```

The argument `code=3` to `arrows()` produces double-headed arrows.

The arrows drawn by the `arrows()` function are rather crude, and other packages provide more visually pleasing alternatives; see, for example, the `p.arrows()` function in the `sfsmisc` package (Maechler, 2018).

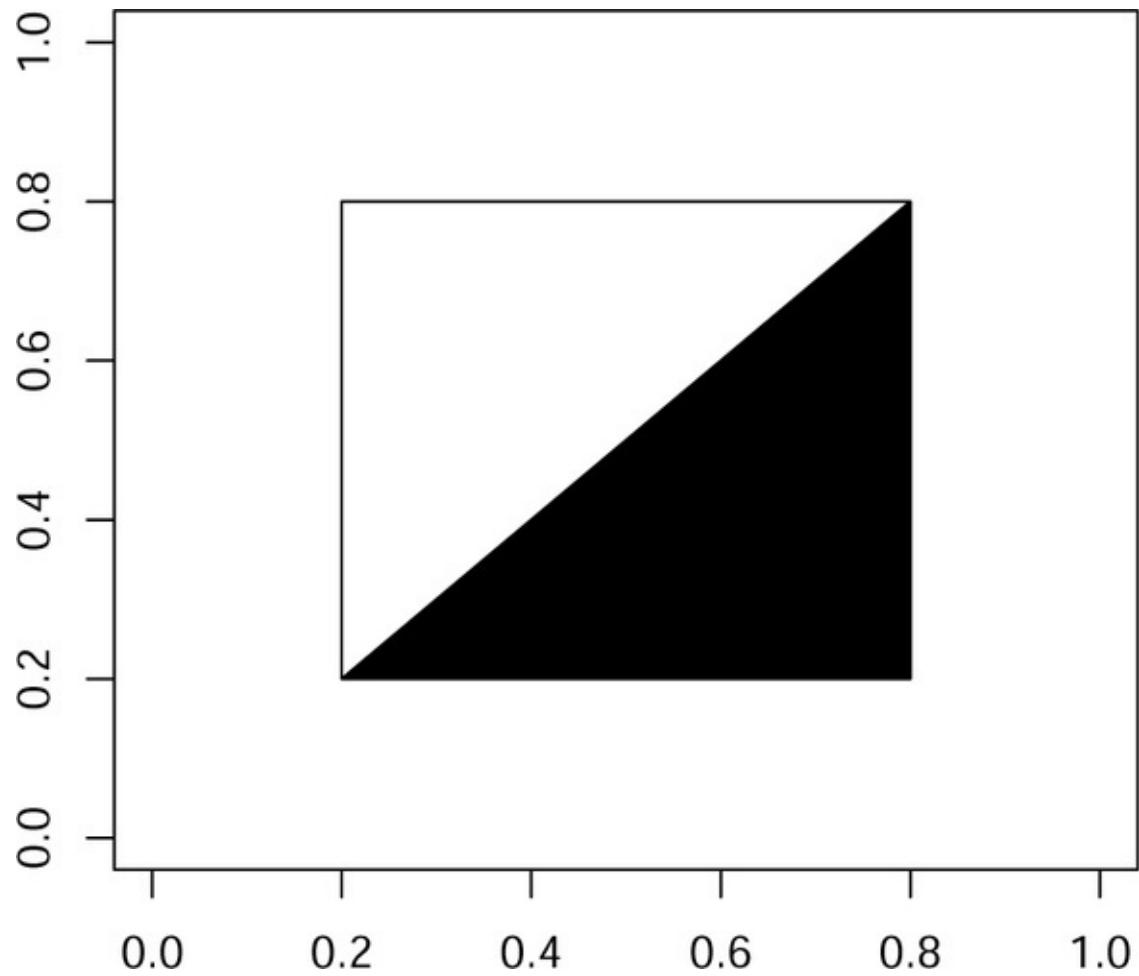
## **polygon ()**

Another self-descriptive function is `polygon()`, which takes as its first two arguments vectors defining the  $x$  and  $y$  coordinates of the vertices of a polygon; for example, to draw the two triangles (i.e., three-sided polygons) in [Figure 9.9](#):

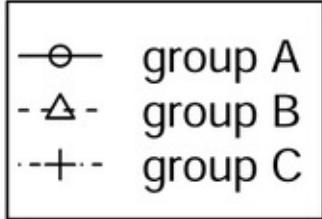
```
plot(c(0, 1), c(0, 1), type="n", xlab="", ylab="")
polygon(c(0.2, 0.8, 0.8), c(0.2, 0.2, 0.8), col="black")
polygon(c(0.2, 0.2, 0.8), c(0.2, 0.8, 0.8))
```

The `col` argument, if specified, gives the color to use in filling the polygon (see the discussion of colors in [Section 9.1.4](#)).

**Figure 9.9** Filled and unfilled triangles produced by `polygon()`.



**Figure 9.10** Using the legend () function.



## legend ()

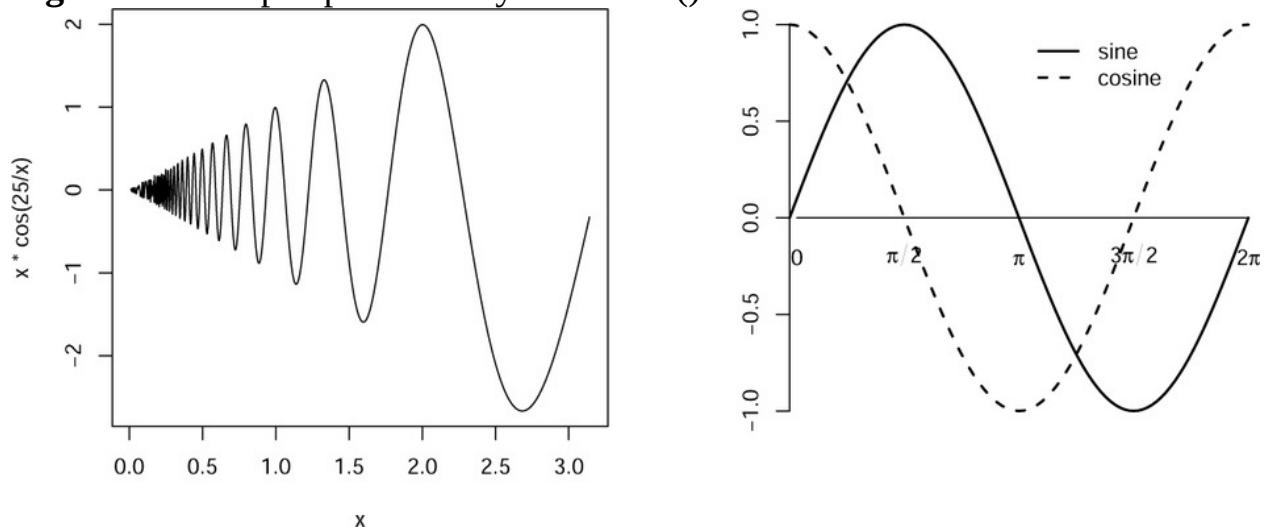
The `legend ()` function may be used to add a legend to a graph; an illustration appears in [Figure 9.10](#):

```
plot (c (1, 5), c (0, 1), axes=FALSE, type="n", xlab="", ylab="",
frame.plot=TRUE)
```

```
legend (locator (1), legend=c ("group A", "group B", "group C"), lty=c
(1, 2, 4), pch=1:3)
```

We use `locator ()` to position the legend: We find that this is sometimes easier than computing where the legend should be placed. Alternatively, we can place the legend by specifying the coordinates of its upper-left corner, or as one of "topleft", "topcenter", "topright", "bottomleft", "bottomcenter", or "bottom-right". If we put the legend in one of the corners, then the argument `inset=0.02` will inset the legend from the corner by 2% of the size of the plot.

**Figure 9.11** Graphs produced by the curve () function.



## curve ()

The curve () function can be used to graph an R function or expression, given as its first argument, or to add a curve representing a function or expression to an existing graph. The second and third arguments of curve (), from and to, define the domain over which the function is to be evaluated; the argument n, which defaults to 101, sets the number of points at which the function is to be evaluated; and the argument add, which defaults to FALSE, determines whether curve () produces a new plot or adds a curve to an existing plot.

If the first argument to curve () is a function, then it should be a function of one argument; if it is an expression, then the expression should be a function (in the mathematical sense) of a variable named x. For example, the left-hand panel of [Figure 9.11](#) is produced by the following command, in which the built-in variable pi is the mathematical constant  $\pi$ ,

```
curve (x*cos (25/x), 0.01, pi, n=1000)
```

while the next command, in which the expression is a function of a variable named y rather than x, produces an error,

```
curve (y*cos (25/y), 0.01, pi, n=1000)
```

Error in curve (y \* cos (25/y), 0.01, pi, n = 1000) :

'expr' must be a function or an expression containing 'x'

The graph in the right-hand panel of [Figure 9.11](#) results from the following commands:

```
rainbow(10)
```

```
[1] "#FF0000FF" "#FF9900FF" "#CCFF00FF" "#33FF00FF" "#00FF66FF"  
[6] "#00FFFFFF" "#0066FFFF" "#3300FFFF" "#CC00FFFF" "#FF0099FF"
```

The pos argument to the axis () function, set to 0 for both the horizontal and vertical axes, draws the axes through the origin of the coordinate space—that is, through the point (0, 0). The argument bty="n" to legend () suppresses the box that is normally drawn around a legend. We move the tick label for  $x = 0$  slightly to the right (at=0.1) so that it isn't overplotted by the vertical axis and suppress the tick marks by setting their line width to zero (lwd.ticks=0).

## 9.1.4 Specifying Colors

Using different colors can be an effective means of distinguishing graphical elements such as lines or points. Although we are limited to monochrome graphs in this book and in many print publications, the specification of colors in R graphs is nevertheless straightforward to describe. If you are producing graphics for others, keep in mind that some people have trouble distinguishing various colors, particularly red and green. Colors should be used for clarity and to convey information rather than to provide “eye candy.”

Plotting functions such as lines () and points () specify color via a col argument; this argument is vectorized, allowing you to select a separate color for each point. R provides three principal ways of specifying a color. The most basic, although rarely directly used, is by setting *RGB* (Red, Green, Blue) values.<sup>6</sup> For example, the rainbow () function creates a spectrum of RGB colors, in this case, a spectrum of 10 colors from pure red to deep purple:

```
gray(0:9/9)
```

```
[1] "#000000" "#1C1C1C" "#393939" "#555555" "#717171" "#8E8E8E"  
[7] "#AAAAAA" "#C6C6C6" "#E3E3E3" "#FFFFFF"
```

[6](#) In addition to the RGB color space, R supports specification of colors by *hue*, *saturation*, and *value* (the *HSV color space*) and by *hue*, *chroma*, and *luminance* (the *HCL color space*). The `hsv()` and `hcl()` functions translate the HSV and HCL color spaces to RGB values—see `help("rgb")`, `help("hsv")`, and `help("hcl")`.

Similarly, the `gray()` function creates gray levels from black (`gray(0)`) to white (`gray(1)`):

```
head(colors(), 10)
```

```
[1] "white"           "aliceblue"      "antiquewhite"  
[4] "antiquewhite1"  "antiquewhite2"  "antiquewhite3"  
[7] "antiquewhite4"  "aquamarine"    "aquamarine1"  
[10] "aquamarine2"
```

```
length(colors())
```

```
[1] 657
```

The color codes are represented as *hexadecimal* (base 16) numbers, of the form `"#RRGGBB"` or `"#RRGGBBTT"`, where each pair of hex digits *RR*, *GG*, and *BB* encodes the intensity of one of the three additive primary colors—from 00 (i.e., 0 in decimal) to FF (i.e., 255 in decimal).[7](#) The various shades of gray have equal R, G, and B components. The hex digits *TT*, if present, represent *transparency*, varying from 00, completely transparent, to FF, completely opaque; if the TT digits are absent, then the value FF is implied. Ignoring variations in transparency, there are over 16 million possible colors.

[7](#) Just as decimal digits run from 0 through 9, hexadecimal digits run from 0 through 9, A, B, C, D, E, F, representing the decimal numbers 0 through 15. Thus, the hex number `"#2B"` represents in decimal  $2 \times 16^1 + 11 \times 16^0 = 32 + 11 = 43$ , and `"#FF"` is  $15 \times 16 + 15 = 255$ .

Specifying colors by name is often more convenient, and the names that R

recognizes are returned by the colors () function:

```
palette()  
[1] "black"  "red"    "green3"  "blue"   "cyan"   "magenta" "yellow"  
[8] "gray"
```

We have shown only the first 10 of more than 600 prespecified color names: Enter the command colors () to see them all. The full set of color definitions appears in the editable file rgb.txt, which resides in the R etc subdirectory. All of the common color names are included, so it's safe to specify names like "pink", "orange", or "purple".

Yet another simple way of specifying a color is by number. What the numbers mean depends on the value returned by the palette () function:<sup>8</sup>

```
library("car")  
pie(rep(1, 8), col=carPalette())  
pie(rep(1, 100), col=rainbow(100), labels=rep("", 100))  
pie(rep(1, 100), col=rainbow_hcl(100), labels=rep("", 100))  
pie(rep(1, 100), col=gray(0:100/100), labels=rep("", 101))
```

[8](#) At one time, the eighth color in the standard R palette was "white". Why was that a bad idea?

Thus, col=c (4, 2, 1) would first use "blue", then "red", and finally "black". We can use the pie () function to see the colors in the default palette (the result of which is not shown because of our restriction to monochrome graphs):

**pie (rep (1, 8), col=1:8)**

R permits us to change the value returned by palette () and thus to change the meaning of the color numbers. For example, we used

**palette (rep ("black", 8))**

to write the *R Companion*, so that all plots are rendered in black and white.<sup>9</sup> If you prefer the colors produced by `rainbow()`, you could set

### ***palette (rainbow (10))***

<sup>9</sup> More correctly, all plots that refer to colors by number are black and white. We could still get other colors by referring to them by name or by their RGB values. Moreover, some graphics functions, including those in the **car** package (see below), select colors independently of the R color palette.

In this last example, we change both the palette colors and the number of colors. The choice

```
library ("colorspace")
```

```
palette (rainbow_hcl (10))
```

uses a palette suggested by Zeileis, Hornik, and Murrell (2009) and implemented in the **colorspace** package.

Changing the palette is session specific and is forgotten when we exit from R. If you want a custom palette to be used at the beginning of each R session, you can add a `palette()` command to your R profile (described in the Preface).

Graphics functions in the **car** package use the `carPalette()` function for color selection. By default, the colors returned by `carPalette()` are `c("black", "blue", "magenta", "cyan", "orange", "gray", "green3", "red")`, which is nearly a permutation of the colors in the standard R palette meant to minimize the use of red and green in the same graph and substituting the more visible color orange for yellow. You can also use `carPalette()` to customize the colors used by **car** graphics functions: See help ("`carPalette`").

To get a sense of how all this works, try each of the following commands:

```

UN <- na.omit(UN[, c("ppgdp", "infantMortality")])
  # remove missing data
gdp <- UN$ppgdp/1000
infant <- UN$infantMortality
ord <- order(gdp)      # order data by gdp
gdp <- gdp[ord]         # sort gdp
infant <- infant[ord]  # put infant into corresponding order

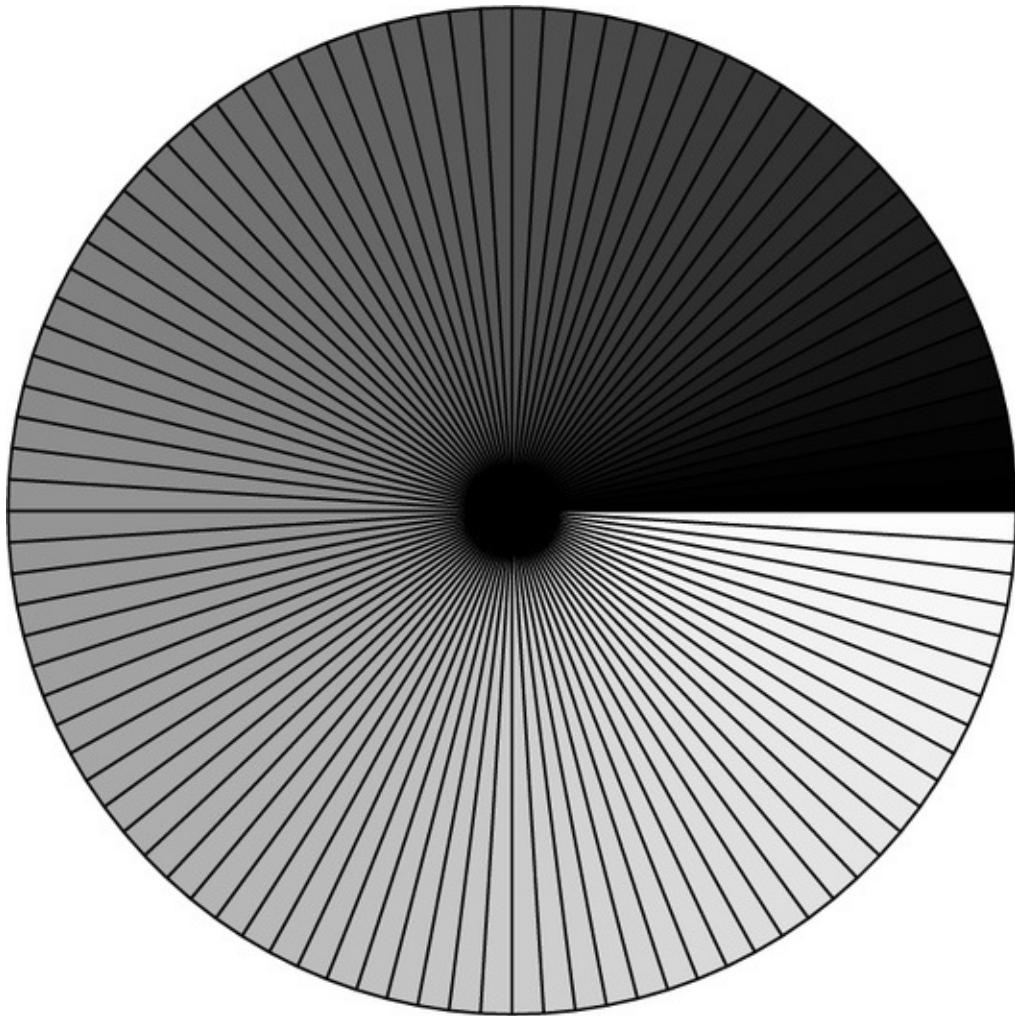
```

The graph produced by the last command appears in [Figure 9.12](#).

## 9.2 Putting It Together: Explaining Local Linear Regression

Most of the analytic graphs and many of the presentation graphs that you will want to create are easily produced in R, often with a single function call. The principal aim of this chapter is to show you how to construct the small proportion of graphs that require substantial customization. Such graphs are diverse by definition, and it would be futile to try to cover their construction exhaustively. Instead, we will develop an example that employs many of the functions introduced in the preceding section to illustrate how one can proceed.

**Figure 9.12** The 101 colors produced by gray (0:100/100), starting with gray (0) (black) and ending with gray (1) (white).



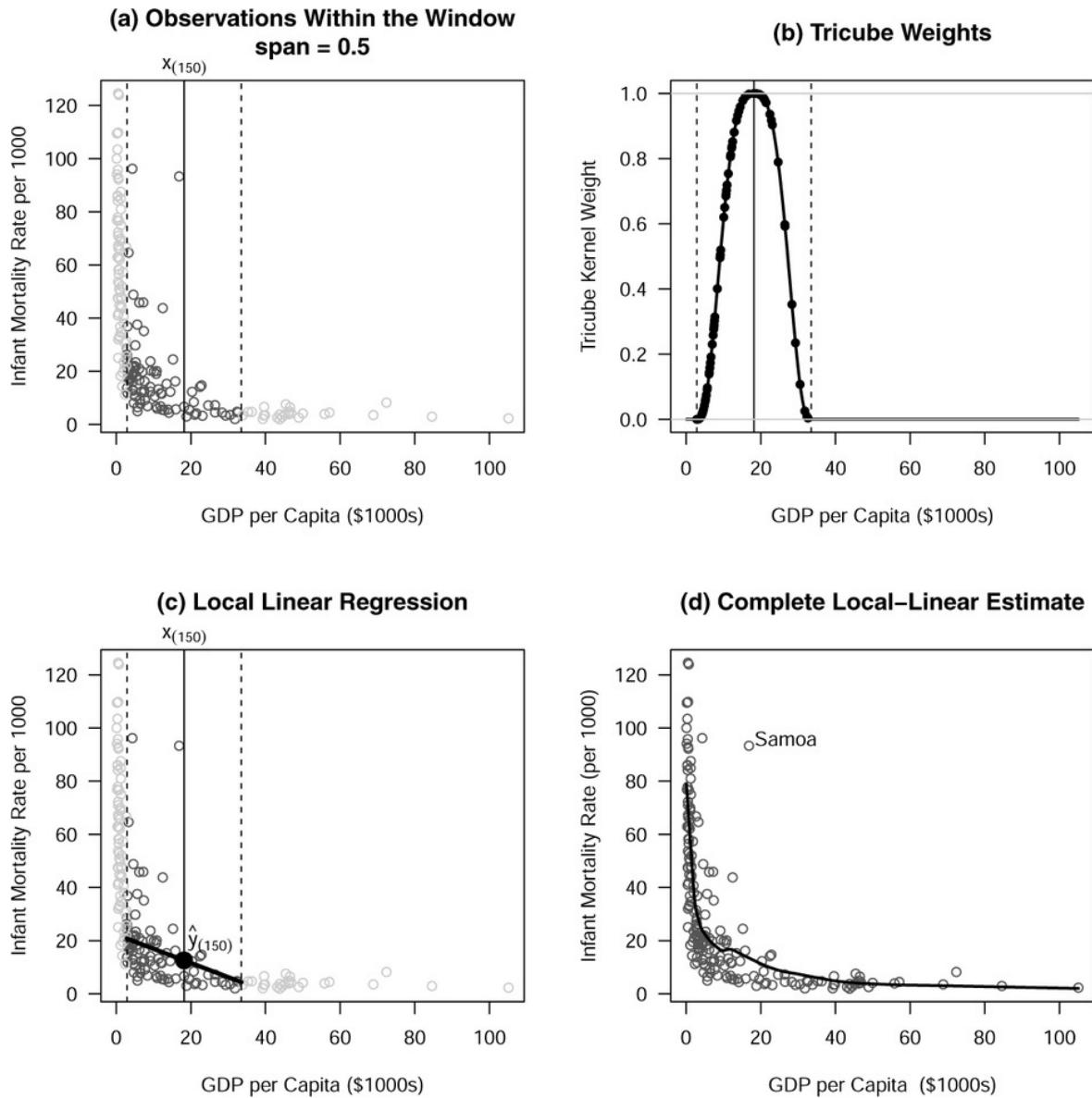
We describe step by step how to construct the diagram in [Figure 9.13](#), which is designed to provide a graphical explanation of *local linear regression*, a method of nonparametric regression. Local linear regression is very similar to the loess scatterplot smoother used extensively in the *R Companion*, for example, by the `scatterplot()` function in the **car** package. Loess, as implemented in the standard R `loess()` function, is a bit more general, however—for example, `loess()` by default computes a robust fit that discounts outliers.

The end product of local linear regression is the estimated regression function shown in [Figure 9.13](#) (d). In this graph,  $x$  is the GDP per capita and  $y$  the infant mortality rate of each of 193 nations of the world, from the UN data set in the **carData** package.<sup>10</sup> The estimated regression function is obtained as follows:

<sup>10</sup> We previously encountered the scatterplot for these two variables in [Figure 3.16](#) (page 152).

- Select the *grid* of points at which to estimate the regression function, either by selecting a number (say, 100) of equally spaced values that cover the range of  $x$  or by using the observed values of  $x$ , if, as in the UN data, they are not too numerous. We follow the latter course, and let  $x_0$  be a value from among  $x_1, x_2, \dots, x_n$ , at which we will compute the corresponding fitted value . The fitted regression simply joins the points , after arranging the  $x$ -values in ascending order.
- The estimate is computed as the fitted value at  $x_0$  from a weighted-least-squares (WLS) regression of the  $y_i$  corresponding to the  $m$  closest  $x_i$  to the *focal value*  $x_0$ , called the *nearest neighbors* of  $x_0$ . We set  $m = [n \times s]$  for a prespecified fraction  $s$  of the data, called the *span*, where the square brackets represent rounding to the nearest integer. The span is a *tuning parameter* that can be set by the user, with larger values producing a smoother estimate of the regression function. To draw [Figure 9.13](#), we set  $s = 0.5$ , and thus  $m = [0.5 \times 193] = 97$  points contribute to each local regression.<sup>11</sup>

**Figure 9.13** A four-panel diagram explaining local linear regression.



The identification of the  $m$  nearest neighbors of  $x_0$  is illustrated in [Figure 9.13](#) (a) for  $x_0 = x_{(150)}$ , the 150th ordered value of gdp, with the dashed vertical lines in the graph defining a *window* centered on  $x_{(150)}$  that includes its 97 nearest neighbors. Selecting  $x_0 = x_{(150)}$  for this example is entirely arbitrary, and we could have used any other  $x$ -value in the grid. The size of the window is potentially different for each choice of  $x_0$ , but it always includes the same fraction of the data. In contrast, *fixed-bandwidth local regression* fixes the size of the window but lets the *number of points* used in the local regressions vary.

[11](#) For values of  $x_0$  in the middle of the range of  $x$ , typically about half the nearest neighbors are smaller than  $x_0$  and about half are larger than  $x_0$ , but for  $x_0$  near the minimum (or maximum) of  $x$ , almost all the nearest neighbors will be larger (or smaller) than  $x_0$ , and this *edge effect* can introduce *boundary bias* into the estimated regression function. Fitting a local regression line rather than simply computing a locally weighted average of the  $y^i$  (called the *kernel regression estimator*) reduces bias near the boundaries.

- The scaled distances between each of the  $xs$  and the focal  $x_0$  are  $z_i = |x_i - x_0|/h_0$ , where  $h_0$  is the distance between  $x_0$  and the most remote of its  $m$  nearest neighbors. The weights  $w_i$  to be used in the local WLS regression depend on a *kernel function*, as in kernel density estimation (discussed in [Section 3.1.2](#)). We use the Tukey tricube kernel function, setting  $w_i = K_T(z_i)$ , where

$$K_T(z) = \begin{cases} (1 - z^3)^3 & \text{for } z < 1 \\ 0 & \text{for } z \geq 1 \end{cases}$$

The tricube weights, shown in [Figure 9.13](#) (b), take on the maximum value of 1 at the focal  $x_0$  in the center of the window and fall to zero at the boundaries of the window.

- The  $y$ -values associated with the  $m$  nearest neighbors of  $x_0$  are then regressed by WLS on the corresponding  $x$ -values, using the tricube weights, to obtain the fitted value , where  $a_0$  and  $b_0$  are respectively the intercept and slope from the locally weighted WLS regression. This step is illustrated in [Figure 9.13](#) (c), in which the local WLS regression line is drawn in the window surrounding  $x_{(150)}$  containing the points that enter the local WLS regression; the fitted value at  $x_{(150)}$  is shown as a large black dot on the local regression line at the center of the window.
- The whole process is repeated for all values of  $x$  on the selected grid, and the fitted points are joined to produce [Figure 9.13](#) (d), completing the local linear regression.

To draw [Figure 9.13](#), we first need to divide the graph into four panels:

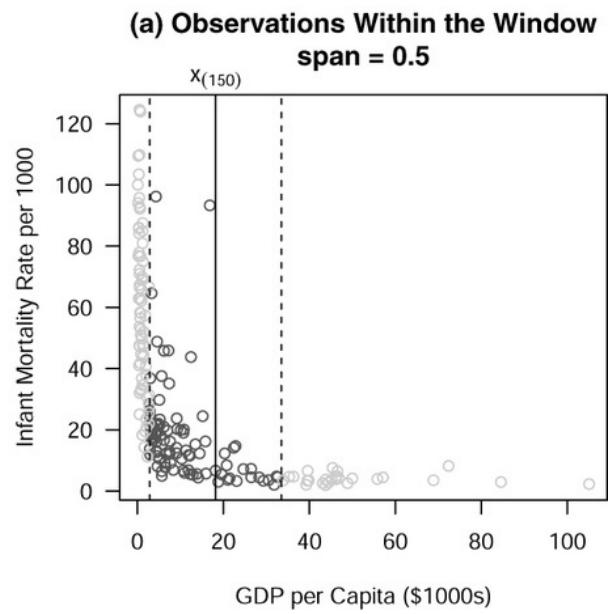
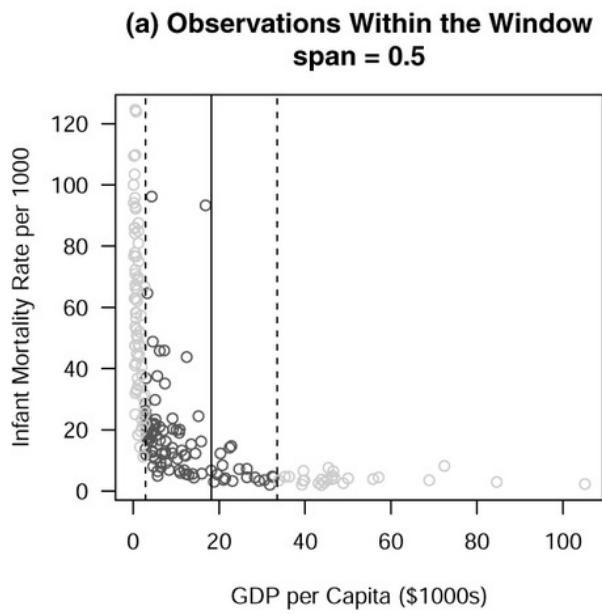
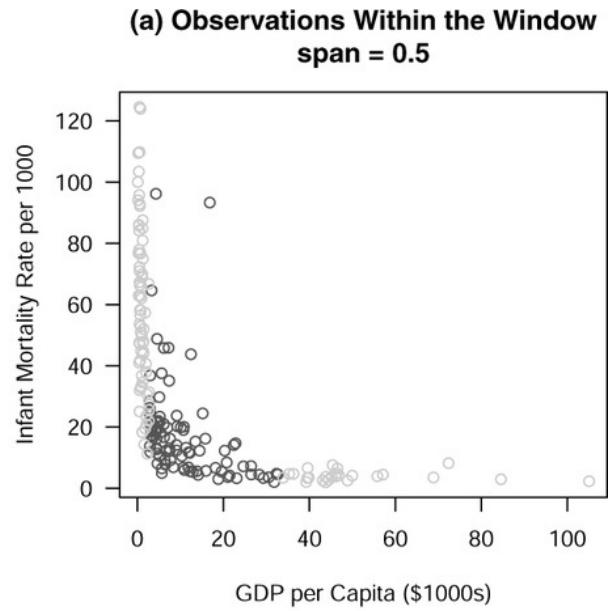
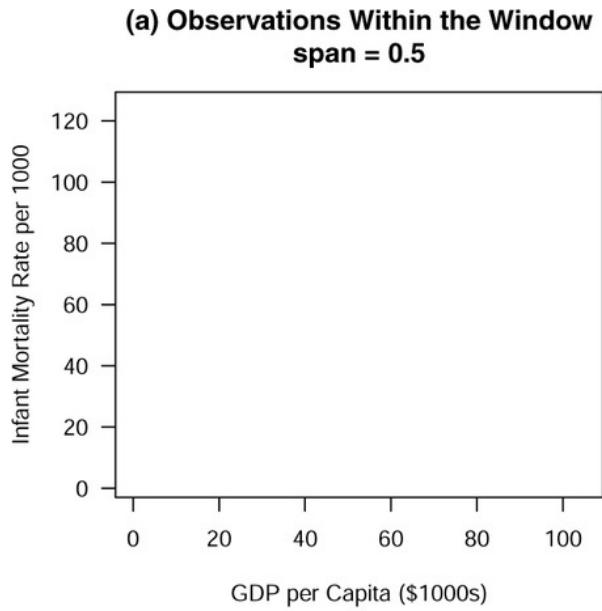
```
oldpar <- par(mfrow=c(2, 2), las=1) # 2-by-2 array of graphs
```

We are already familiar with using `par(mfrow=c(rows, cols))` and `par(mfcoll=c(rows, cols))`, defining an array of panels to be filled either row-wise, as in the current example, or column-wise. We also include the additional argument `las=1` to `par()`, which makes all axis tick labels, including those on the vertical axis, parallel to the horizontal axis, as is required in some journals. Finally, as suggested in [Section 9.1.1](#), we save the original value of `par()` and will restore it when we finish the graph.

After removing missing data from the two variables in the UN data frame in the **carData** package,<sup>12</sup> where the data for the graph reside, we order the data by *x*-values (i.e., `gdp`), being careful also to sort the *y*-values (`infant`) into the same order:

[12](#) We are careful here only to filter for missing values on the two variables used in the graph, so as not to discard cases unnecessarily that are missing data on variables we don't use.

**Figure 9.14** Building up panel (a) of [Figure 9.13](#) step by step.



```

x0 <- gdp[150]           # focal x = x_(150)
dist <- abs(gdp - x0)     # distance from focal x
h <- sort(dist)[97]       # bandwidth for span of 0.5 (n = 193)
pick <- dist <= h         # observations within window
  
```

We turn our attention to panel (a) of [Figure 9.13](#):

```

plot(gdp, infant,
      xlab="GDP per Capita ($1000s)",
      ylab="Infant Mortality Rate per 1000",
      type="n",
      main="(a) Observations Within the Window\nspan = 0.5")

```

Thus,  $x_0$  holds the focal  $x$ -value,  $x_{(150)}$ ;  $\text{dist}$  is the vector of distances between the  $xs$  and  $x_0$ ;  $h$  is the distance to the most remote point in the neighborhood for span 0.5 and  $n = 193$ ; and  $\text{pick}$  is a logical vector equal to TRUE for cases within the window and FALSE otherwise.

We draw the initial graph using the `plot()` function to define axes and a coordinate space:

```

plot(range(gdp), c(0, 1),
      xlab="GDP per Capita ($1000s)",
      ylab="Tricube Kernel Weight",
      type="n", main="(b) Tricube Weights")
abline(v=x0)

```

The character "\n" in the main argument produces a new line. The result of this command is shown in the upper-left panel of [Figure 9.14](#). In the upper-right panel, we add points to the plot, using dark gray (gray (0.25)) for points within the window and light gray (gray (0.75)) for those outside the window:

```
points(gdp[pick], infant[pick], col=gray(0.25))
```

```
points(gdp[!pick], infant[!pick], col=gray(0.75))
```

Next, in the lower-left panel, we add a solid vertical line at the focal  $x_0 = x_{(150)}$  and broken lines at the boundaries of the window:

```
abline(v=x0) # focal x
```

```
abline(v=c(x0 - h, x0 + h), lty=2) # window
```

Finally, in the lower-right panel, we use the `text()` function to display the focal value  $x_{(150)}$  at the top of the panel:

```
text(x0, par("usr")[4] + 5, expression(x[(150)]), xpd=TRUE)
```

The second argument to `text()`, giving the vertical coordinate, makes use of `par("usr")` to find the *user coordinates* of the boundaries of the plotting region. The command `par("usr")` returns a vector of the form  $c(x_1, x_2, y_1, y_2)$ , and here we pick the fourth element,  $y_2$ , which is the maximum vertical coordinate in the plotting region. Adding 5—one-quarter of the distance between vertical tick marks—to this value positions the text a bit above the plotting region, which is our aim.<sup>13</sup> The argument `xpd=TRUE` permits drawing outside of the normal plotting region. The text itself is given as an expression `()`, allowing us to incorporate mathematical notation in the graph, here the subscript  $(150)$ , to typeset the text as  $x_{(150)}$ .

[13](#) Typically, this kind of adjustment requires a bit of trial-and-error.

Panel (b) of [Figure 9.13](#) is also built up step by step. We begin by setting up the coordinate space and axes, drawing vertical lines at the focal  $x_0$  and at the boundaries of the window, and horizontal gray lines at zero and 1:

```
abline(v=c(x0 - h, x0 + h), lty=2)
abline(h=c(0, 1), col="gray")
tricube <- function(x, x0, h) {
  z <- abs(x - x0)/h
  ifelse(z < 1, (1 - z^3)^3, 0)
}
```

We then write a function to compute tricube weights.<sup>14</sup>

```

plot(gdp, infant,
      xlab="GDP per Capita ($1000s)",
      ylab="Infant Mortality Rate per 1000",
      type="n", main="(c) Local Linear Regression")
points(gdp[pick], infant[pick], col=gray(0.25))
points(gdp[!pick], infant[!pick], col=gray(0.75))
abline(v=x0)
abline(v=c(x0 - h, x0 + h), lty=2)
mod <- lm(infant ~ gdp,
           weights=tricube(gdp, x0, h)) # local WLS fit
new <- data.frame(gdp=c(x0 - h, x0, x0 + h))
fit <- predict(mod, newdata=new) # fits for line, y-hat at x0
lines(c(x0 - h, x0 + h), fit[c(1, 3)], lwd=3) # local reg line
points(x0, fit[2], pch=16, cex=2) # plot y-hat for x0
text(x0, par("usr")[4] + 5, expression(x[(150)]), xpd=TRUE)
text(x0 + 1, fit[2] + 2.5,
     expression(hat(y)[(150)]), adj=c(0, 0))

```

[14](#) The ifelse () command is described in [Section 10.4.1](#).

To complete panel (b), we draw the tricube weight function, showing points representing the weights for observations that fall within the window:

```

tc <- function (x) tricube (x, x0, h) # to use with curve curve (tc, min
(gdp), max (gdp), n=1000, lwd=2, add=TRUE) points (gdp[pick], tricube
(gdp, x0, h)[pick], pch=16)

```

The function tc () is needed for curve (), which requires a function of a single argument (see [Section 9.1.3](#)). The remaining two arguments to tricube () are set to the values of x0 and h that we created earlier in the global environment. The plotting character pch=16 is a filled circle.

Panel (c) of [Figure 9.13](#) is similar to panel (a), except that we draw the local regression line between the boundaries of the window surrounding the focal  $x_0 = x_{(150)}$ , along with the fitted value  $y_{(150)}$  at the center of the window:

```

plot(gdp, infant,
      xlab="GDP per Capita ($1000s)",
      ylab="Infant Mortality Rate (per 1000)",
      main="(d) Complete Local-Linear Estimate", col=gray(0.25))
yhat <- numeric(length(gdp)) # initialize to 0s
for (i in 1:length(gdp)){
  # fitted value at each x
  x0 <- gdp[i] # focal value
  dist <- abs(gdp - x0) # distances to focal value
  h <- sort(dist)[97] # distance to most remote point
                      # in neighborhood
  mod <- update(mod, weights=tricube(gdp, x0, h)) # WLS fit
  yhat[i] <- predict(mod, newdata=data.frame(gdp=x0))
}
lines(gdp, yhat, lwd=2) # plot smooth
text(gdp[149], infant[149], # label outlier
     paste0(" ", rownames(UN)[149]), adj=c(0, 0))
par(oldpar) # restore original value of par

```

We use the weights argument to lm () for the local WLS regression and predict () to get fitted values at the boundaries of the window to draw the regression line and, at its center, the fitted value .<sup>15</sup>

<sup>15</sup> Were we permitted to use color, we could, for example, emphasize the local regression line and the fitted value at by setting col="magenta" in the calls to lines () and points (). We could similarly use distinct colors rather than gray levels for the points inside and outside of the window surrounding  $x_{(150)}$ . We invite the reader to try this variation.

Finally, to draw panel (d) of [Figure 9.13](#), we repeat the calculation of for each of the  $n = 197$  ordered  $x$ -values, using a for () loop to set the focal  $x_0$  to each value of gdp in turn:<sup>16</sup>

```

plot(1:10, type="n", xlab="", ylab "", axes=FALSE, frame=TRUE)
box("outer")
text(5.5, 5.5, "plotting region", cex=1.5)
text(5.5, 4.5, "standard par() settings")
for (side in 1:4) {
  for (line in 0:4) {
    mtext(paste("line", line), side=side,
          line=line, adj=0.70)
  }
}
for (side in 1:4) {
  mtext(paste("margin, side", side), side=side, line=1,
        adj=0.25, cex=1.25)
}

```

[16](#) for () loops are described in [Section 10.4.2](#).

We also call the text () function to label the outlying point "Samoa" in the graph—the 149th ordered value in the data—pasting a blank onto the beginning of the country name and using the adj argument to fine-tune the placement of the text. It's clear both in panel (c) and in panel (d) of [Figure 9.13](#) that the outlier draws the fitted local regression toward it, creating a “bump” in the fit. We invite the reader to compare this to the robust fit produced by the loess () function.

### 9.2.1 Finer Control Over Plot Layout

[Figure 9.15](#) shows the standard layout of a plotting device in R and is created by the following commands, using the mtext () function to show the positions of lines in the margins of the plot:

```

par(mfrow=c(2, 2), oma=c(1, 1, 2, 1), mar=c(2, 2, 2, 1))
n.lines <- par("mar") - 1
for (plot in 1:4) {
  plot(1:10, type="n", xlab="", ylab="",
    axes=FALSE, frame=TRUE)
  box("figure")
  text(5.5, 5.5, paste("panel", plot), cex=1.5)
  if (plot == 1) {
    text(5.5, 3.5,
"par(mfrow=c(2, 2),\n  oma=c(1, 1, 2, 1),\n  mar=c(2, 2, 2, 1))",
      cex=1.25)
  }
  for (side in 1:4) {
    for (line in 0:n.lines[side]) {
      mtext(paste("inner margin line", line), side=side,
        line=line, cex=0.75)
    }
  }
}

```

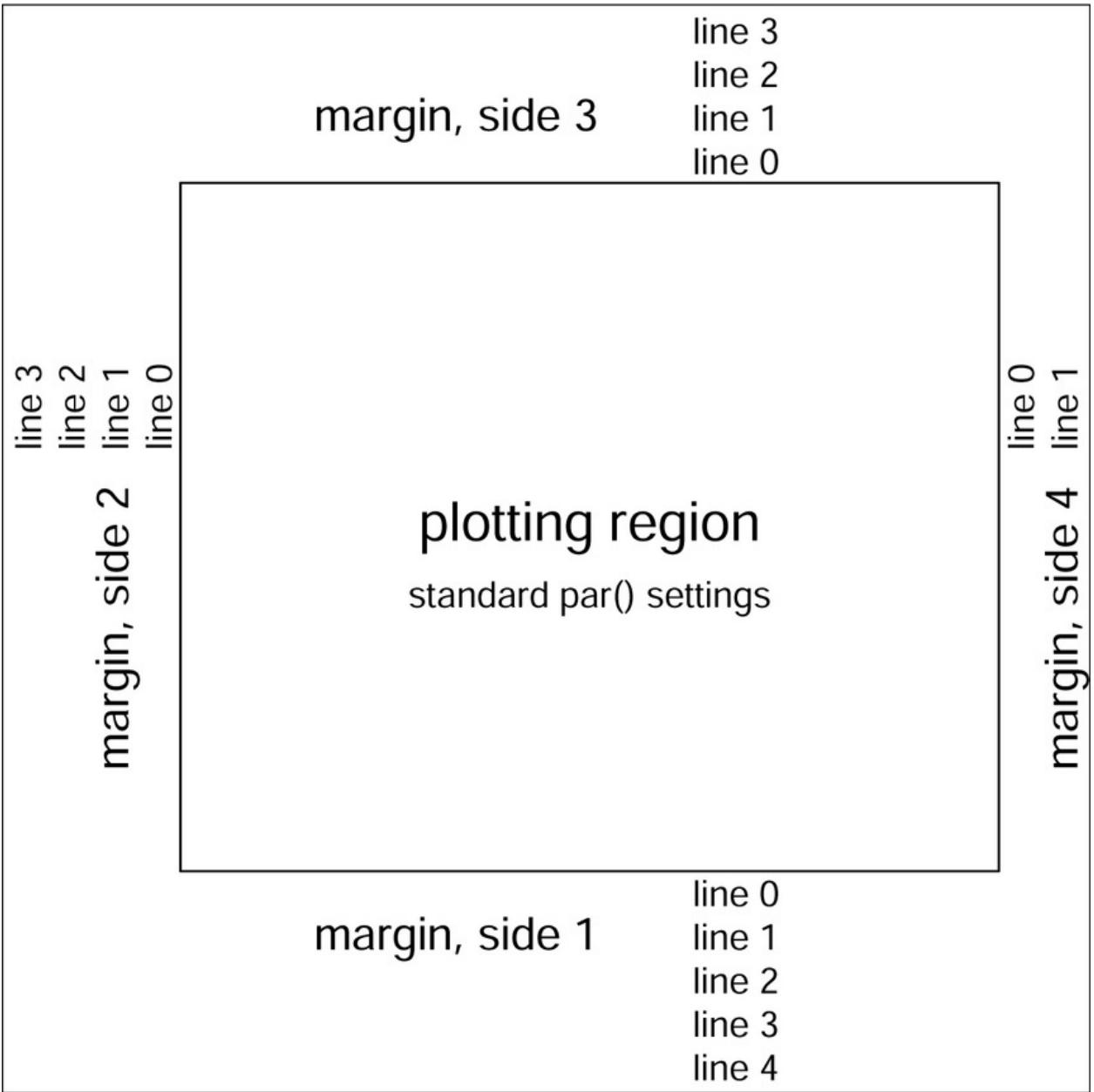
Similarly, [Figure 9.16](#) shows the layout of an R plot device divided into four subplots by mfrow=c (2, 2):

```

box("inner")
box("outer")
n.lines <- par("oma") - 1
for (side in 1:4) {
  for (line in 0:n.lines[side]) {
    mtext(paste("outer margin line", line),
      side=side, line=line, outer=TRUE)
  }
}

```

**Figure 9.15** Standard layout of an R graphics device.



```

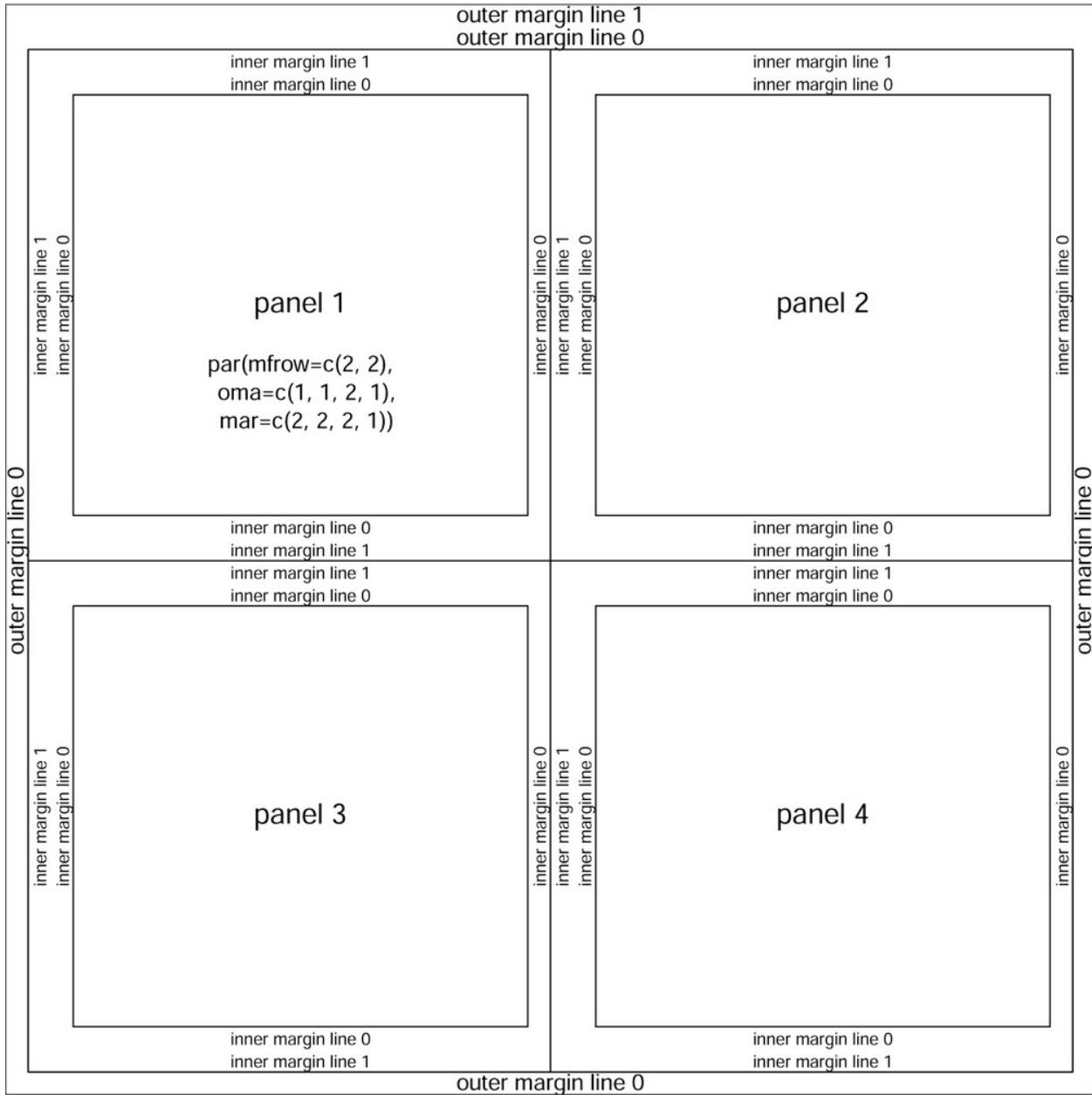
par(oma=c(0, 0, 1, 0), mar=c(2, 3, 3, 2))
  # expand top outer margin
par(fig=c(0, 0.5, 0.5, 1)) # top-left panel
x <- seq(0, 1, length=200)
Ey <- rev(1 - x^2) # expected value of y; reverse values
y <- Ey + 0.1*rnorm(200) # add errors
plot(x, y, axes=FALSE, frame=TRUE,
      main="(a) monotone, simple", cex.main=1,
      xlab="", ylab="", col="darkgray", cex=0.75)
lines(x, Ey, lwd=2)
mtext("x", side=1, adj=1) # text in bottom margin
mtext("y ", side=2, at=max(y), las=1) # text in left margin

par(fig=c(0.5, 1, 0.5, 1), new=TRUE) # top-right panel
x <- seq(0.02, 0.99, length=200)
Ey <- log(x/(1 - x))
y <- Ey + 0.5*rnorm(200)
plot(x, y, axes=FALSE, frame=TRUE,
      main="(b) monotone, not simple", cex.main=1,
      xlab="", ylab="", col="darkgray", cex=0.75)
lines(x, Ey, lwd=2)
mtext("x", side=1, adj=1)
mtext("y ", side=2, at=max(y), las=1)

```

The outer margins around the four subplots are set to `oma=c(1, 1, 2, 1)` lines and the inner margins for each subplot to `c(2, 2, 2, 1)` lines, in the order bottom, left, top, right. Again, we use `mtext()` to write in the plot margins.

**Figure 9.16** An R graphics device divided into an array of subplots by `par(mfrow=c(2, 2))`.

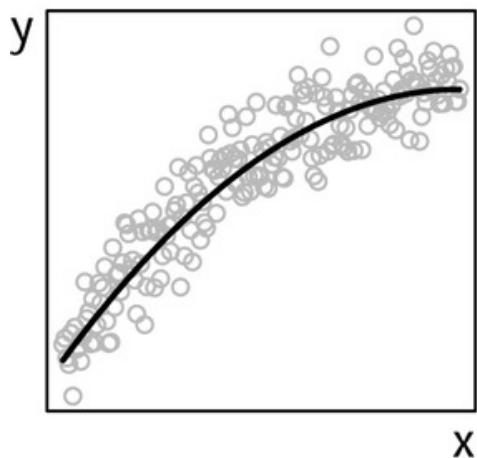


More complex arrangements than are possible with `mfrow` and `mfcol` can be defined using the `layout()` function or the `fig` argument to `par()`: See help ("layout") and help ("par"). We illustrate here with `fig`, producing [Figure 9.17](#), a graph meant to distinguish different kinds of nonlinearity:

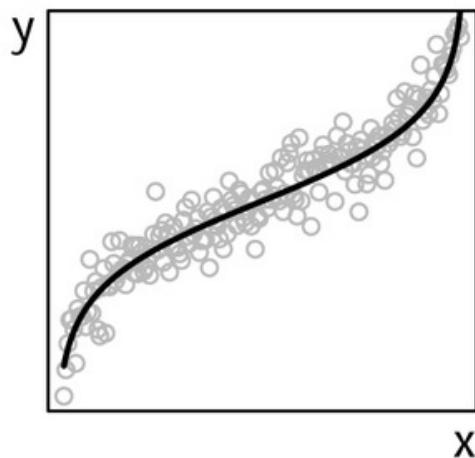
**Figure 9.17** Using the `fig` graphical parameter for finer control over plot layout.

# Nonlinear Relationships

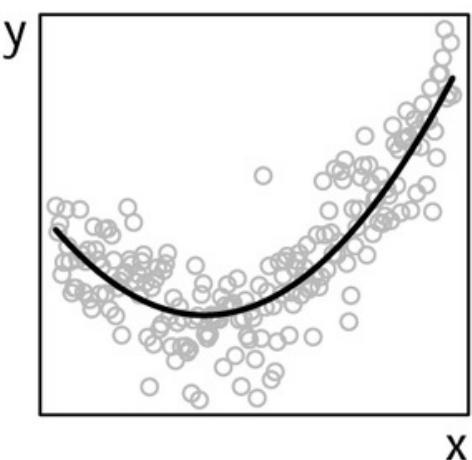
(a) monotone, simple



(b) monotone, not simple



(c) non-monotone, simple



```

par(fig=c(0.25, 0.75, 0, 0.5), new=TRUE) # bottom panel
x <- seq(0.2, 1, length=200)
Ey <- (x - 0.5)^2
y <- Ey + 0.04*rnorm(200)
plot(x, y, axes=FALSE, frame=TRUE,
      main="(c) non-monotone, simple", cex.main=1,
      xlab="", ylab="", col="darkgray", cex=0.75)
lines(x, Ey, lwd=2)
mtext("x", side=1, adj=1)
mtext("y ", side=2, at=max(y), las=1)
title("Nonlinear Relationships", outer=TRUE)
par(oma=c(0, 0, 1, 0), mar=c(2, 3, 3, 2))
n.lines <- par("mar") - 1

par(fig=c(0, 0.5, 0.5, 1)) # upper left
plot(0:1, 0:1, type="n", axes=FALSE, frame=TRUE)
box("figure")
text(0.5, 0.5, "panel 1")
for (side in 1:4){
  for (line in 0:n.lines[side]){
    mtext(paste("inner margin line", line), side=side,
          line=line, cex=0.75)
  }
}

par(fig=c(0.5, 1, 0.5, 1), new=TRUE) # upper right
plot(0:1, 0:1, type="n", axes=FALSE, frame=TRUE)
box("figure")
text(0.5, 0.5, "panel 2") # upper right
for (side in 1:4){

```

The first par () command leaves room in the top outer margin for the graph title, which is given in the title () command with outer=TRUE at the end, and establishes the margins for each panel. The order of margins both for oma (the outer margins) and for mar (the margins for each panel) are c (*bottom, left, top,*

*right*), and in each case, the units for the margins are lines of text. The `fig` argument to `par()` establishes the boundaries of each panel, expressed as fractions of the display region of the plotting device, in the order `c(x-minimum, x-maximum, y-minimum, y-maximum)`, measured from the bottom left of the device. Thus, the first panel, defined by the command `par(fig=c(0, 0.5, 0.5, 1))`, extends from the left margin to the horizontal middle and from the vertical middle to the top of the plotting device. Each subsequent panel begins with the command `par(new=TRUE)` so as not to clear the plotting device, which normally occurs when a high-level plotting function such as `plot()` is invoked. We use the `mtext()` command to position the axis labels just where we want them in the margins of each panel; in the `mtext()` commands, `side=1` refers to the bottom margin and `side=2` to the left margin of the current panel.

[Figure 9.18](#) shows the structure of the layout for [Figure 9.17](#):

```

for (line in 0:n.lines[side]){
  mtext(paste("inner margin line", line), side=side,
        line=line, cex=0.75)
}
}

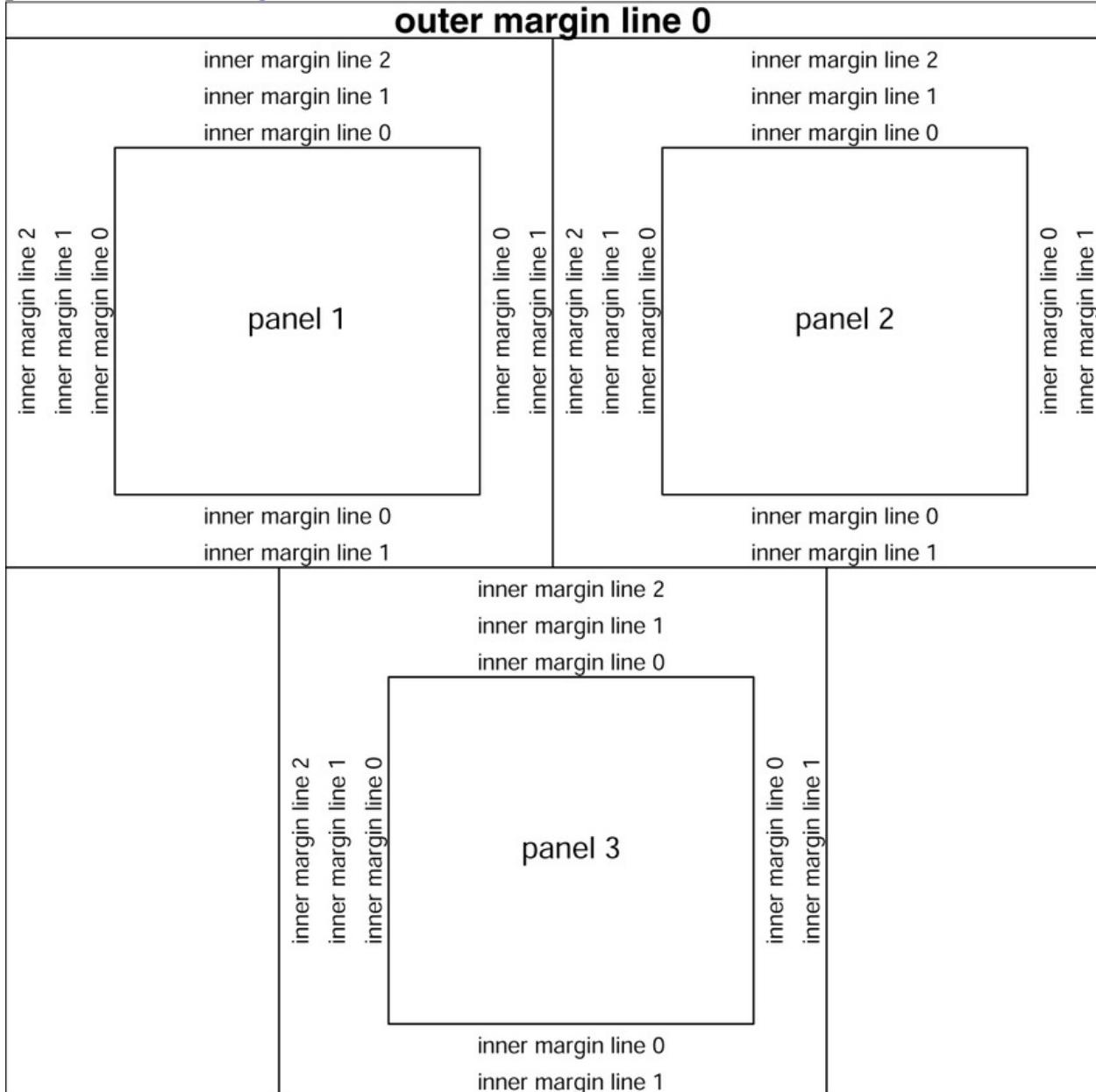
par(fig=c(0.25, 0.75, 0, 0.5), new=TRUE) # bottom center
plot(0:1, 0:1, type="n", axes=FALSE, frame=TRUE)
text(0.5, 0.5, "panel 3")
box("figure")
for (side in 1:4){
  for (line in 0:n.lines[side]){
    mtext(paste("inner margin line", line), side=side,
          line=line, cex=0.75)
  }
}

box("outer")
box("inner")
title("outer margin line 0", outer=TRUE)

```

```
(plt <- ggplot(Prestige, aes(x=income, y=prestige)))
(plt <- plt + labs(x="Average Income", y="Prestige Score"))
(plt <- plt + geom_point())
plt + geom_smooth()
```

**Figure 9.18** An R graphics device with panels defined by the `fig` graphical parameter, as in [Figure 9.17](#).



## 9.3 Other R Graphics Packages

To this point, we've emphasized the traditional R graphics system, along with its fundamental ink-on-paper metaphor. In addition to traditional graphics, the R distribution includes the powerful and general **grid** graphics package, which is extensively documented by Murrell (2011, Part II). Constructing **grid** graphs is sufficiently complicated that it rarely makes sense to use the **grid** package directly for individual plots or diagrams.

The **grid** package provides a basis for constructing object-oriented graphics systems in R, the two most important examples of which are the **lattice** (Sarkar, 2008) and **ggplot2** (Wickham, 2016) packages, introduced in the following sections. We subsequently briefly take up the **maps** and **ggmap** packages for drawing statistical maps and mention some other notable graphics packages for R.<sup>17</sup>

<sup>17</sup> Many of the thousands of packages on CRAN make graphs of various sorts, and it is certainly not our object to take on the Herculean task of cataloging what's available to R users in the realm of statistical graphics. A reasonable place to begin is the *CRAN Graphics Task View* at <http://cran.r-project.org/web/views/Graphics.html>.

### 9.3.1 The **lattice** Package

The **lattice** package, which is part of the standard R distribution, is a descendant of the **trellis** library in S, originally written by Richard Becker and William Cleveland. The term *trellis* derives from graphs that are composed of a rectangular array of similar panels—what Tufte (1983) terms *small multiples*. The implementation of the **lattice** package for R is based on **grid** graphics and is independent of the S original. The **lattice** package is documented extensively in Sarkar (2008).

We use **lattice** graphics without much comment elsewhere in the *R Companion*, for example, in effect plots, introduced in [Section 4.3](#), and in plotting data for individual clusters in [Chapter 7](#) on mixed-effects models. In [Section 10.7.1](#), we call the `xyplot()` function in the **lattice** package to create [Figure 10.6](#) (on page 518), generated by the following command:

```
library ("lattice")
```

```
xyplot (salary ~ yrs.since.phd | discipline:rank, groups=sex,
data=Salaries, type=c ("g", "p", "r"), auto.key=TRUE)
```

This is a typical **lattice** command:

- A formula determines the horizontal and vertical axes of each *panel* in the graph, here with salary on the vertical axis and yrs.since.phd on the horizontal axis.
- Each panel plots a subset of the data, determined by the values of the variables to the right of the vertical bar (|), which is read as *given*. In the example, we get a separate panel showing the scatterplot of salary versus yrs.since.phd for each combination of the factors discipline and rank.
- The groups argument specifies additional conditioning within a panel, using distinct colors, symbols, and lines for each level of the grouping factor—sex, in this example.
- The familiar data argument is used, as in statistical-modeling functions, to supply a data frame containing the data for the graph.
- The type argument specifies characteristics of each panel, in this case printing a background grid ("g"), showing the individual points ("p"), and displaying a least-squares regression line ("r") for each group in each panel of the plot. Other useful options for type include "smooth" for a loess smooth with default smoothing parameter and "l" (the letter "el") for joining the points with lines. The latter will produce surprising and probably undesirable results unless the data within groups are ordered according to the variable on the horizontal axis. The auto.key argument prints the legend at the top of the plot. If there is a grouping variable, then the regression lines and smooths are plotted separately for each level of the grouping variable; otherwise, they are plotted for all points in the panel.

The boxplots in [Figure 10.7](#) (page 519) were similarly generated using the **lattice** function bwplot () (for “box-and-whisker plot,” Tukey’s original term for boxplots):

```
bwplot (salary ~ discipline:sex | rank, data=Salaries, scales=list (rot=90),
layout=c (3, 1))
```

The panels are defined by the levels of the factor rank, and in each panel,

boxplots for salary are drawn for all combinations of levels of discipline and sex.

Graphics functions in the **lattice** package return objects of class "trellis", which can be printed by the print () function, called either explicitly or implicitly (if the object isn't assigned to an R variable), or can be modified by the user before printing. "Printing" a "trellis" object draws the graph on the current plotting device.

Consider the following variation on the boxplots produced for the Salaries data, shown at the top of [Figure 9.19](#):

```
(bwp <- bwplot (salary ~ sex | rank + discipline, data=Salaries, scales=list  
 (x=list (rot=45), y=list (log=10, rot=0))))
```

We assign the "trellis" object returned by bwplot () to bwp and use our usual trick of enclosing the command in parentheses so that the object is also printed (i.e., plotted). The call to bwplot () produces a graph with six panels for the combinations of rank and discipline; within each panel, boxplots are drawn for salary by sex. We specify the scales argument to use logs for the salary axis and to control the orientation of the tick labels for both axes.

Because we save the "trellis" object representing the graph in the R variable bwp, we can subsequently modify it. The **latticeExtra** package (Sarkar & Andrews, 2016) contains a variety of functions for manipulating "trellis" objects;<sup>18</sup> for example, we can call the useOuterStrips () function to move the panel labels:

```
library ("latticeExtra")  
  
useOuterStrips (bwp, strip.left=strip.custom (strip.names=TRUE,  
 var.name="Discipline"))
```

<sup>18</sup> Unlike the **lattice** package, which is part of the basic R distribution, the **latticeExtra** package, and the other packages discussed in this section (including **ggplot2**, **maps**, and **ggmap**), must be installed from CRAN.

In this case, the modified "trellis" object returned by `useOuterStrips()` is simply “printed” (and is shown at the bottom of [Figure 9.19](#)).

Graphs produced by **lattice** functions are based on a different approach than standard R graphics, in that a complex plot is usually specified in a single call to a graphics function, rather than by adding to a basic graph in a series of independently executed commands. As a result, the command to create an initial **lattice** graph can be complicated. The key arguments include those for *panel functions*, which determine what goes into each panel; *strip functions*, to determine what goes into the labeling strips at the top of the panels; and *scale functions*, which control the axis scales. Both the **lattice** and **latticeExtra** packages contain many prewritten panel functions likely to suit the needs of most users, or you can write your own panel functions. When, as in the example, a **lattice** graph is modified in subsequent commands, the object representing the graph is manipulated, rather than simply adding graphical elements by drawing directly on a graphics device as in traditional R graphics.

In addition to scatterplots produced by `xyplot()` and boxplots produced by `bwplot()`, the **lattice** package includes 13 other high-level plotting functions for dot plots, histograms, various three-dimensional plots, and more, and the **latticeExtra** package adds several more high-level graphics functions.

### 9.3.2 The **ggplot2** Package

The **ggplot2** package (Wickham, 2016) was inspired by Leland Wilkinson’s influential book on the design of statistical graphs, *The Grammar of Graphics* (L. Wilkinson, 2005). The **ggplot2** package provides a wide-ranging and flexible alternative graphics system for R, and it has become very popular, at least partly because of the considerable aesthetic appeal of **ggplot2** graphs.

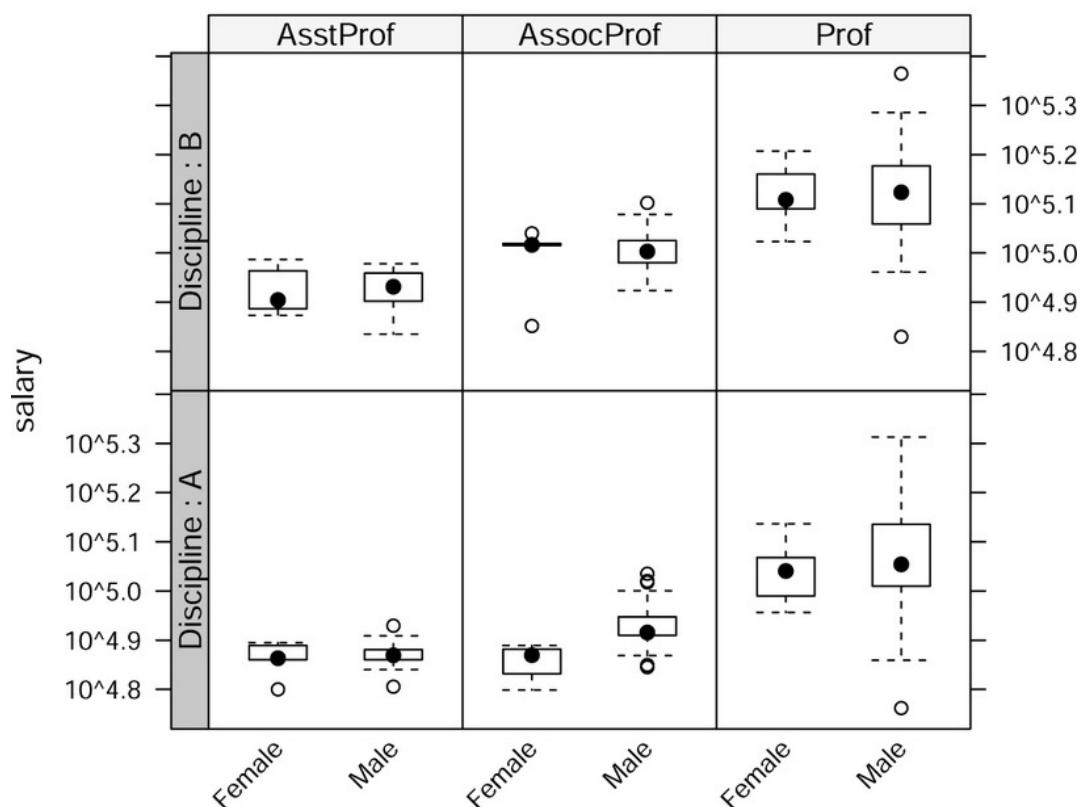
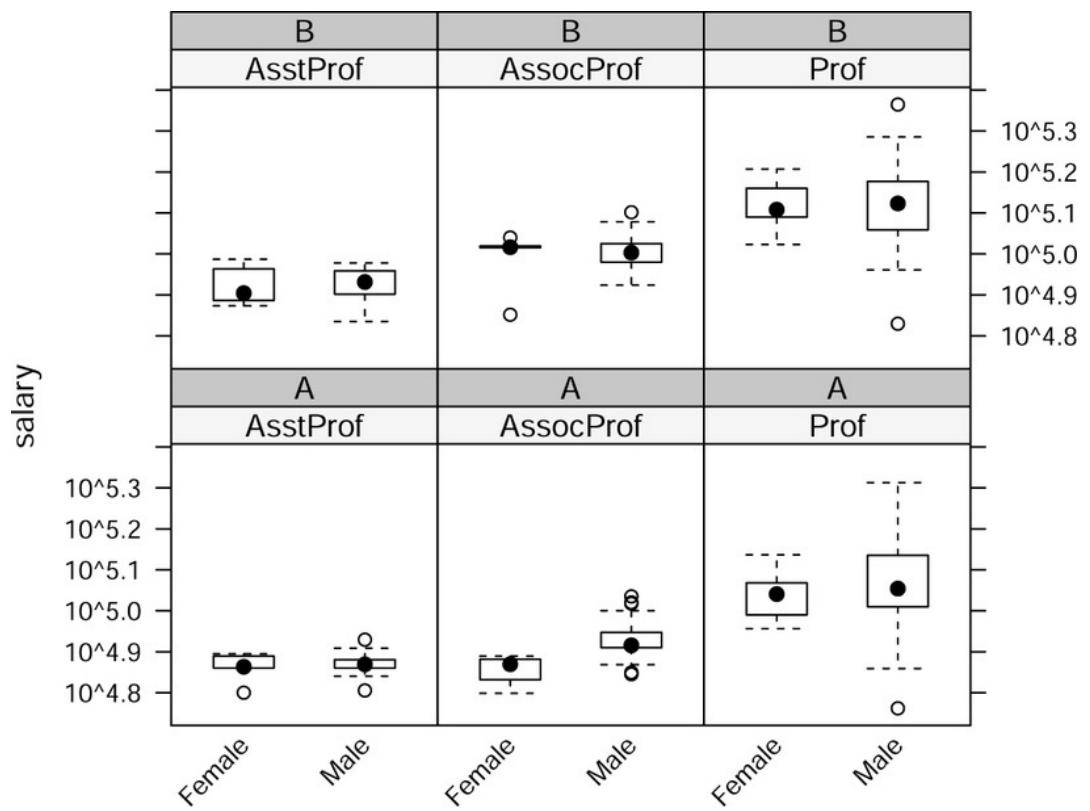
To illustrate, we use the `qplot()` (“quick plot”) function in **ggplot2** to draw a scatterplot ([Figure 9.20](#)) for income and prestige in the Canadian occupational-prestige data, residing in the `Prestige` data frame in the **carData** package.<sup>19</sup>

```
library("ggplot2")  
  
qplot(income, prestige, xlab="Average Income", ylab="Prestige Score",  
      geom=c("point", "smooth"), data=Prestige)
```

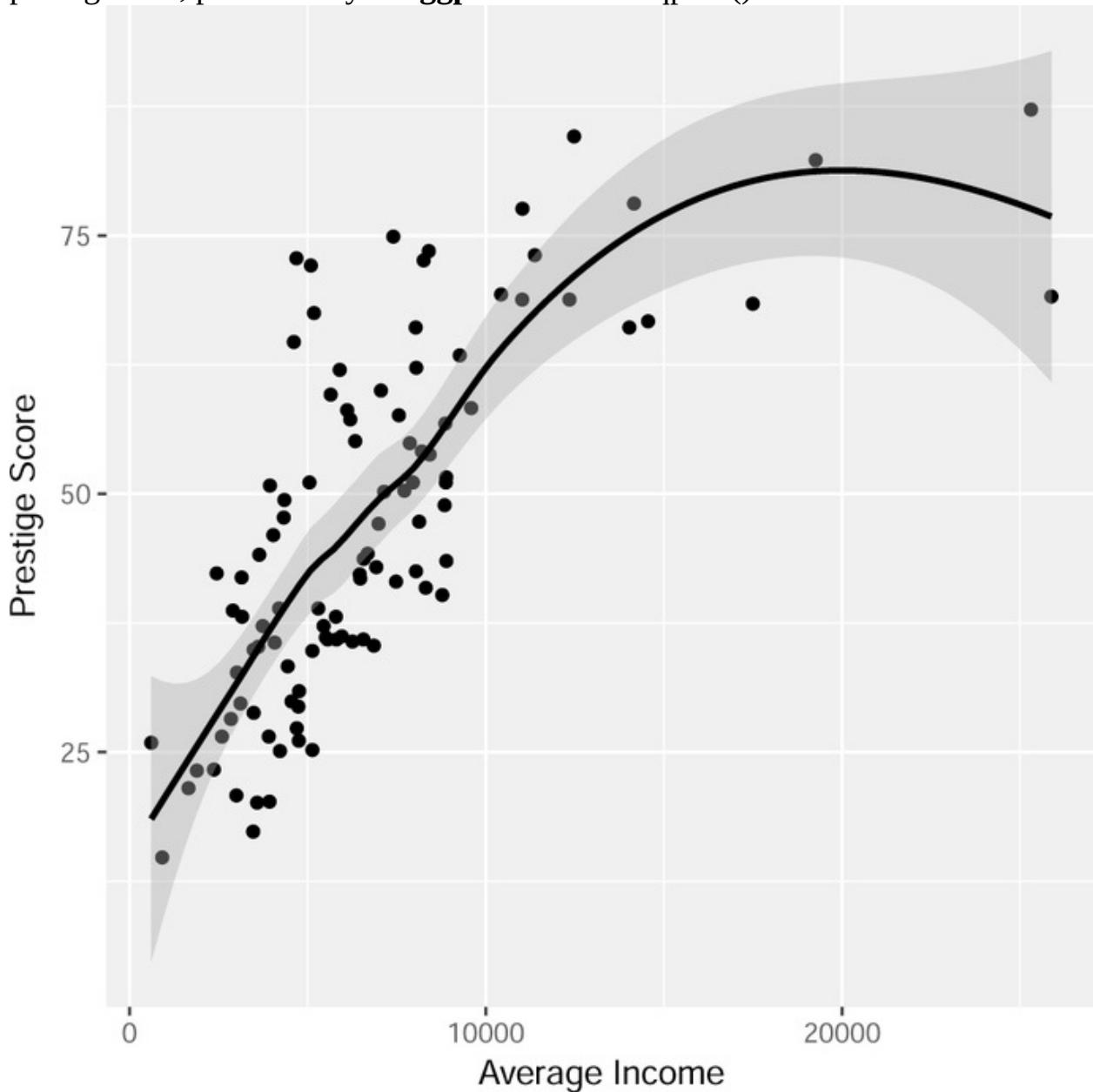
[19](#) We use the Prestige data set at several points in the book, for example, in [Section 4.2.2](#).

With the exception of the geom argument, for “geometry,” part of the extensive jargon from the *Grammar of Graphics*, the call to qplot () is largely self-explanatory. The geom argument determines the content of the graph, in this case, points whose coordinates are given by income and prestige, together with a loess smooth. The band around the smooth represents not conditional variation, as in the scatterplot () function, but a pointwise confidence envelope.

**Figure 9.19** Boxplot of log (salary) by rank, sex, and discipline: original graph produced by bwplot () (top) and modified graph (bottom).



**Figure 9.20** Scatterplot of prestige by income for the Canadian occupational-prestige data, produced by the **ggplot2** function `qplot()`.



This use of `qplot()` is not typical of **ggplot2** graphics. Like the **lattice** package, functions in **ggplot2** produce modifiable objects, of class "ggplot".<sup>20</sup> The `print()` method for "ggplot" objects is responsible for drawing the graph. More commonly, **ggplot2** graphs are built up in several steps, by modifying an initial object. For our example, we may proceed as follows, “printing” the graph at each stage to reveal the process more clearly (see [Figure 9.21](#)):

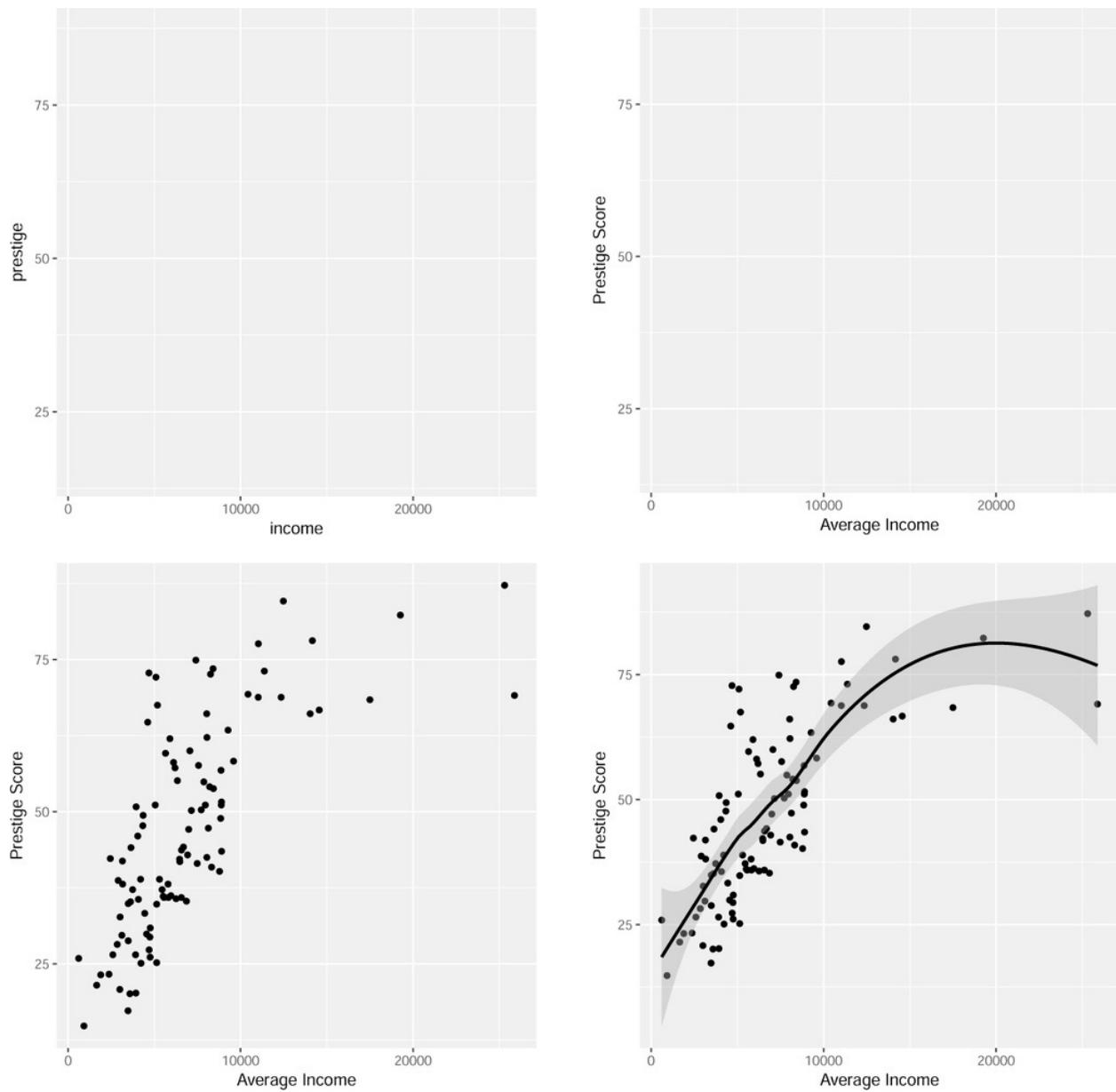
[20](#) Actually, the objects are of class "gg" inheriting from class "ggplot": See Sections 1.7 and 10.9 on object-oriented programming in R.

```
ggplot(Prestige, aes(x=income, y=prestige)) +  
  labs(x="Average Income", y="Prestige Score") +  
  geom_point() + geom_smooth()
```

The initial call to ggplot() sets up the (x, y) coordinate space for the graph, using the aes() ("aesthetic") function, returning a "ggplot" object, which we both save in the variable plt and "print." The **ggplot2** package provides an appropriate "ggplot" method for the R + (addition) operator, and this operator is typically used to modify or augment a "ggplot" object.[21](#) Thus, the second command adds axis labels, the third command adds points to the graph, and the fourth command adds the loess smooth (and confidence band), as shown in the subsequent panels of [Figure 9.21](#).

[21](#) In fact, the + method is for class "gg": See footnote 20.

**Figure 9.21** Scatterplot of prestige by income produced by successive **ggplot2** commands.



We wouldn't normally save and print the graph at each stage, however, and the final plot in [Figure 9.21](#) can be produced by the command

```
ggplot (Prestige, aes (x=income, y=prestige)) + labs (x="Average Income", y="Prestige Score") + geom_point () + geom_smooth ()
```

### 9.3.3 Maps

R has several packages for drawing maps, including the **maps** package (Becker, Wilks, Brownrigg, Minka, & Deckmyn, 2018). Predefined maps are available for the world and for several countries, including the United States. Viewing data on maps can often be illuminating. As a brief example, the data frame `Depredations` in the **carData** package contains data on incidents of wolves killing farm animals, called *depredations*, in Minnesota for the period 1979–1998, from Harper, Paul, Mech, and Weisberg (2008):

```
head(Depredations)
```

|   | longitude | latitude | number | early | late |
|---|-----------|----------|--------|-------|------|
| 1 | -94.5     | 46.1     | 1      | 0     | 1    |
| 2 | -93.0     | 46.6     | 2      | 0     | 2    |
| 3 | -94.6     | 48.5     | 1      | 1     | 0    |
| 4 | -92.9     | 46.6     | 2      | 0     | 2    |
| 5 | -95.9     | 48.8     | 1      | 0     | 1    |
| 6 | -92.7     | 47.1     | 1      | 0     | 1    |

The data include the longitude and latitude of the farms where depredations occurred, as well as the number of depredations at each farm for the whole period (1979–1998) and separately for the earlier period (1991 or before) and for the later period (after 1991). Management of wolf-livestock interactions is a significant public policy question, and maps can help us to understand the geographic distribution of the incidents.

```

library("maps")
par(mfrow=c(1, 2))
map("county", "minnesota", col=gray(0.4))
with(Depredations, points(longitude, latitude,
                           cex=sqrt(early), pch=20))
title("Depredations, 1976-1991", cex.main=1.5)
map("county", "minnesota", col=gray(0.4))
with(Depredations, points(longitude, latitude,
                           cex=sqrt(late), pch=20))
title("Depredations, 1992-1998", cex.main=1.5)

```

To draw separate maps for the early and late period in [Figure 9.22](#), we set up the graphics device with the `mfrow` graphics parameter. The `map()` function is used to draw the map, in this case a map of county boundaries in the state of Minnesota. The coordinates for the map are the usual longitude and latitude, and the `points()` function is employed to add points to the map, with areas proportional to the number of depredations at each location. We use `title()` to add a title to each panel, with the argument `cex.main` to increase the size of the title. The maps tell us where in the state the wolf-livestock interactions occur and where the farms with the largest number of depredations can be found. The range of depredations has expanded to the south and east between time periods. There is also an apparent outlier in the data—one depredation in the southeast of the state in the early period.

The **ggmap** package (Kahle & Wickham, 2013) allows the user to import a map from Google Maps, or from another source, and then to draw on the imported map. For example, using the data frame `Neigh.combined`, which we created in [Section 2.3.5](#), we draw the map shown in [Figure 9.23](#) that displays the number of police stops in Minneapolis neighborhoods:

```

head(Neigh.combined[,  

  c("neighborhood", "long", "lat", "nstops")])  

  neighborhood      long      lat nstops  

1      Armatage -93.312 44.898      77  

2 Audubon Park -93.244 45.018     554  

3      Bancroft -93.255 44.931     134  

4      Beltrami -93.243 44.995     211  

5      Bottineau -93.268 45.012     377  

6       Bryant -93.269 44.932      96  

nrow(Neigh.combined)  

[1] 84  

library("ggmap")  

MplsMap <- get_googlemap(center="Minneapolis, Minnesota",  

  maptype="roadmap", zoom=12, size=c(400, 700))  

ggmap(MplsMap) +  

  geom_point(aes(x=long, y=lat), data=Neigh.combined, alpha=0.5,  

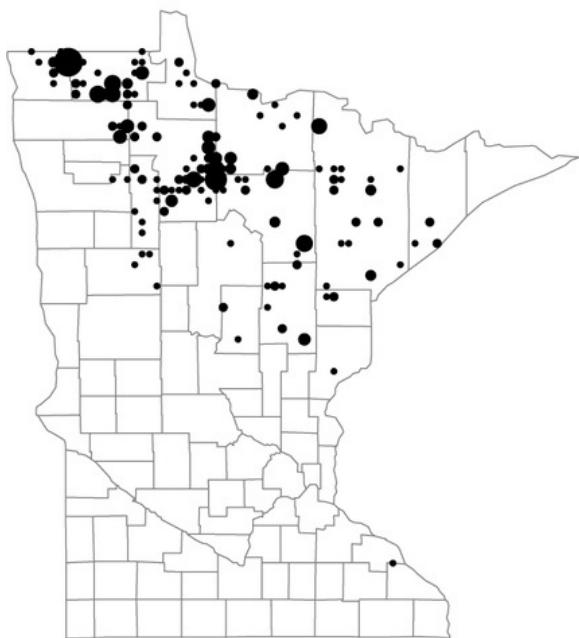
  color="darkred", size=sqrt(Neigh.combined$nstops/50)) +  

  labs(x="", y="")

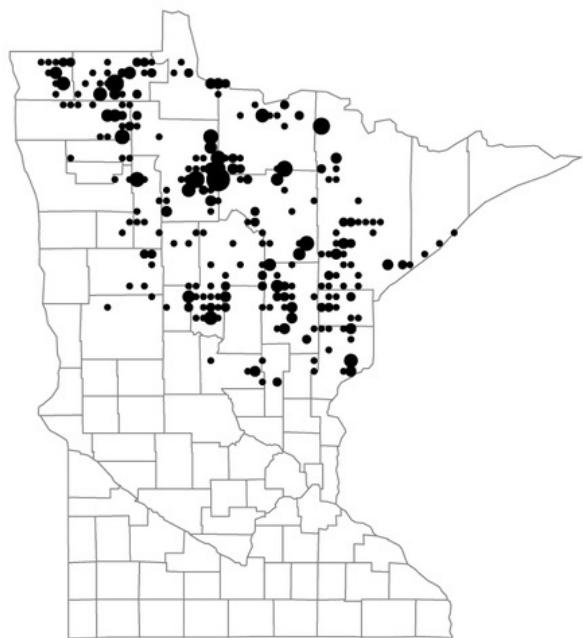
```

**Figure 9.22** Wolf depredations in Minnesota. The areas of the dots are proportional to the number of depredations.

**Depredations, 1976–1991**

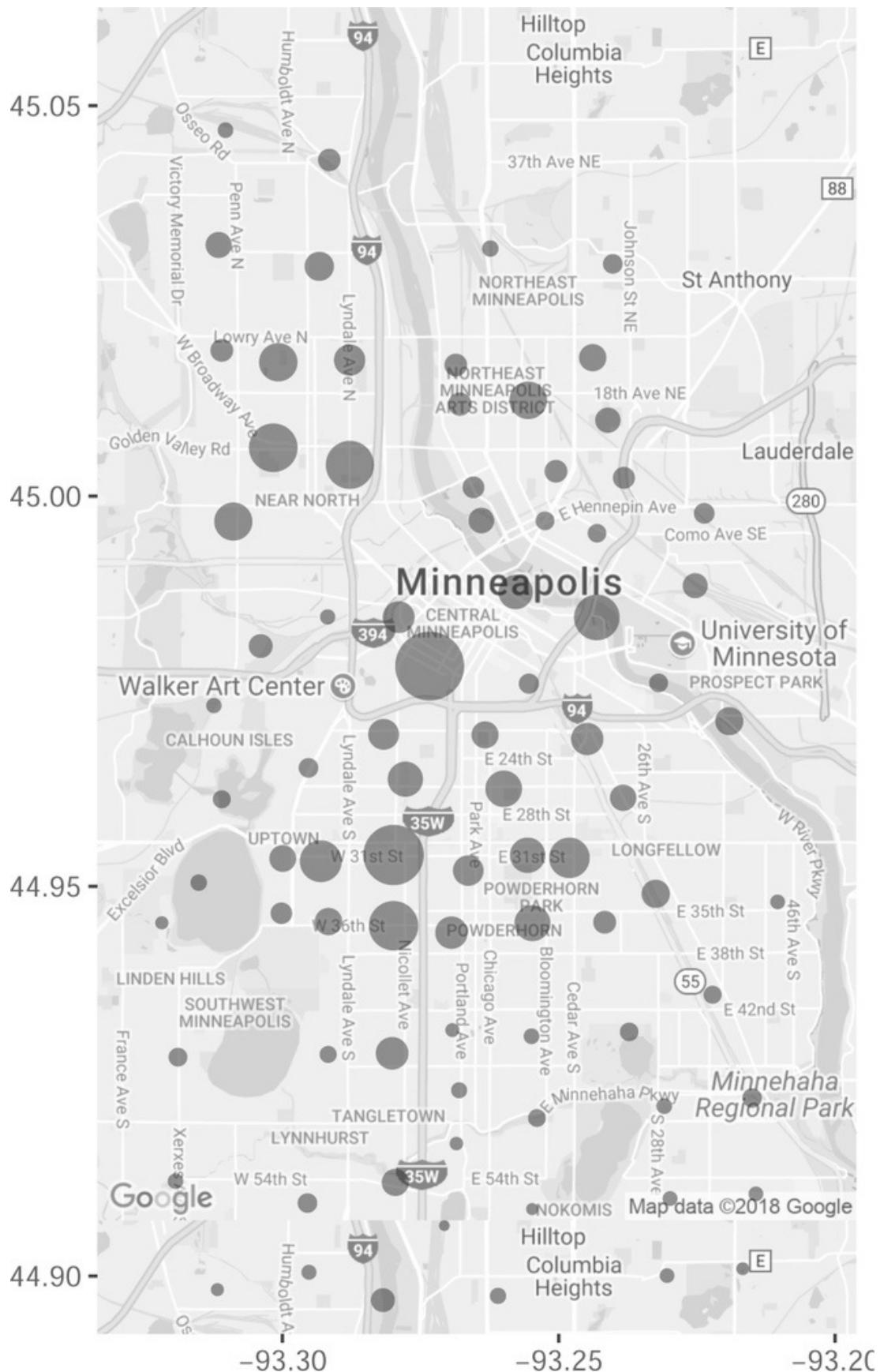


**Depredations, 1992–1998**



The Neigh.combined data set includes (among other variables) the latitude (lat), longitude (long), and the number of traffic and other stops of individuals by the police (nstops) in each of 84 Minneapolis neighborhoods during 2017. The `get_ggmap()` function retrieves a roadmap of Minneapolis over the internet from Google Maps. The arguments `zoom` and `size` set the magnification and size in pixels of the map image. The `ggmap()` function adds points to the map, one per neighborhood, with areas proportional to the number of police stops in the neighborhoods. The argument `alpha=0.5` sets the transparency of the points, and the `color` argument sets their color (although [Figure 9.23](#) is rendered in monochrome in this book). We call the `labs()` function from the **ggplot2** package to suppress the axis labels for the map, which would otherwise print as `long` and `lat`. The use of the `+` operator to modify a "ggmap" object, here the `MplsMap` image, is explained more generally for the **ggplot2** package in [Section 9.3.2](#).

**Figure 9.23** Number of 2017 police stops by Minneapolis neighborhood. The original map is in color.



### 9.3.4 Other Notable Graphics Packages

Written by the developers of RStudio, the **shiny** package (Chang, Cheng, Allaire, Xie, & McPherson, 2017) supports imbedding interactive R graphics in web pages, conveniently created from R Markdown documents. There is a website at <http://shiny.rstudio.com/> with extensive information about **shiny**.

The **rgl** package (Adler & Murdoch, 2017) interfaces R with the OpenGL 3D graphics library (<https://www.opengl.org>), providing a foundation for building 3D dynamic statistical graphs. The possibilities are literally impossible to convey adequately on the static pages of a book, especially without color. A monochrome picture of an **rgl** graph, created by the `scatter3d()` function in the **car** package, appears in [Figure 3.13](#) (on page 146).

The **plotrix** package (Lemon, 2017) provides a number of tools for conveniently adding features to traditional R graphs, produced, for example, by the `plot()` function. An example appears in [Figure 3.11](#) (b) (page 142), where we use the `plotCI()` function from the **plotrix** package to add error bars to a graph. Plots of coefficient estimates and their standard errors or confidence limits are easily created using this function. Enter `help(package="plotrix")` for an index of the functions that are available, and see the individual help pages, demos, and examples for information about the various functions in the package.

The **iplots** (Urbanek & Wichtrey, 2013) and **playwith** (Andrews, 2012) packages introduce interactive graphical capabilities to R, the former via the **RGtk2** package (Lawrence & Temple Lang, 2010), which links R to the GTK+ graphical-user-interface toolkit, and the latter via the **rJava** package (Urbanek, 2017), which links R to the Java computing platform.

The **rggobi** package (D. Cook & Swayne, 2009) provides a bridge from R to the GGobi system, which provides high-interaction dynamic graphics for visualizing multivariate data.

## 9.4 Complementary Reading and References

- Murrell (2011) is the definitive reference both for standard graphics in R and for the **grid** graphics system on which the **lattice** and **ggplot2** packages

are based. Sarkar (2008) provides an extensive reference for trellis graphics as implemented in the **lattice** package, and Wickham (2016) documents the **ggplot2** package.

- There is a very large general literature on statistical graphics. Some influential treatments of the subject include Tukey (1977), Tufte (1983), Cleveland (1993, 1994), and L. Wilkinson (2005).

# 10 An Introduction to R Programming

John Fox and Sanford Weisberg

This chapter shows you how to write simple programs in R.

- The first two sections of the chapter explain why you should learn to program in R, illustrating what you might hope to accomplish using simple examples.
- We then delve into the details, discussing matrix computations in [Section 10.3](#), presenting control structures in [Section 10.4](#), and showing you how to make computations simpler and potentially more efficient by vectorization in [Section 10.5](#).
- We apply these programming concepts to optimization problems in [Section 10.6](#) and to statistical simulations in [Section 10.7](#), which are common areas of application of basic R programming. Think of these illustrative applications as case studies, not as an exhaustive enumeration of what one can accomplish with basic R programming skills.
- We explain how to debug R code in [Section 10.8](#) and how to write object-oriented programs and statistical-modeling functions in R in Sections 10.9 and 10.10.
- Finally, in [Section 10.11](#), we offer some suggestions for organizing and maintaining your R programs.

The goal of this chapter is to provide you with the basics of writing R functions and scripts, primarily to meet immediate needs in routine and not-so-routine data analysis and data management. Although this brief introduction is probably not enough for you to write polished programs for general use, it is nevertheless worth cultivating good programming habits, and the line between programs written for one's own use and those written for others is often blurred.<sup>1</sup>

[1](#) Suggestions for further reading are given at the end of the chapter.

This would be a good time to review basic arithmetic operators and functions in R, discussed in [Section 1.2](#), and the logical and relational operators, in [Section 1.2.6](#).

## 10.1 Why Learn to Program in R?

We assume that most readers of this book are principally interested in using R to analyze their data, and consequently the focus of the book is on fitting regression models to data. We believe, however, that this interest in data analysis can be advanced by learning to write basic R programs. For example:

- Data sets are regularly encountered that must be modified before they can be used by statistical-modeling functions like `lm()` or `glm()`. You may need

to change values of -99 to the missing value indicator NA, or you may want to recode a variable with values zero and 1 to the more informative labels "Male" and "Female". You might be able to find existing functions, both in the standard R distribution and in contributed packages, to perform data management tasks, as discussed in [Chapter 2](#), but writing your own scripts and functions for these kinds of data management operations can be simpler and more efficient and can automate the process of preparing similar data sets. In the course of this book, we often use programming constructs for data management tasks related to statistical modeling.

- Output from existing R functions must often be rearranged for publication. Writing a function or script to format output automatically in the required form can avoid the tedious and error prone process of retyping it. Formatting output programmatically can be useful, for example, when you use R Markdown (discussed in [Section 1.4.2](#)) to produce a finished report.
- A graph is to be drawn that requires first performing several computations to produce the data to plot and then using a sequence of graphics commands to complete the plot. A function written once can automate these tasks, such as the “influence-plot” example in [Section 10.2.2](#).
- A random simulation is required to investigate estimator robustness, estimator error, or for some other purpose. Writing a flexible script or function to perform the simulation simplifies varying factors such as parameter values, estimators, and sample sizes (see [Section 10.7](#)).

This list is hardly exhaustive, but it does illustrate that writing R scripts and functions can be useful in a variety of situations. The ability to program is a fundamental advantage of using a statistical computing environment like R rather than a traditional statistical package.

## 10.2 Defining Functions: Preliminary Examples

R programming entails writing functions. We consider two preliminary motivating examples in this section: writing a function to lag time-series variables, and writing a function to produce an “influence plot,” simultaneously displaying Studentized residuals, hat-values, and Cook’s distances from a linear or generalized linear model. Both examples are real. The first example arose in response to a question posed in an R class. The second example eventually resulted in the `influencePlot()` function in the `car` package.

### 10.2.1 Lagging a Variable

In time-series regression, we may wish to *lag* a variable by one or more time periods, so that, for example, the value of a predictor in the previous time period influences the value of the response in the current time period.<sup>2</sup> To create a

simple example, we define an imaginary 10-period time-series variable `x` with the integers from 1 to 10:

```
(x <- 1:10)
[1] 1 2 3 4 5 6 7 8 9 10
```

[2](#) Because observations in time-series data are typically not independent, OLS regression is not generally appropriate. Time-series regression is the subject of an online appendix to the *R Companion*.

We can lag `x` one time period (i.e., one position), by appending an NA as the first value,

```
(x.lag <- c(NA, x))
[1] NA 1 2 3 4 5 6 7 8 9 10
```

The first element of `x` appears in the second time period, the second element in the third time period, and so on. Simply appending an NA at the start of the result creates a problem, however, because `x.lag` has 11 values, while, we're imagining, other time-series variables in our data have only 10 values. We can compensate by removing the last value of `x.lag`:

```
(x.lag <- x.lag[-11])
[1] NA 1 2 3 4 5 6 7 8 9
```

Removing the last value of the lagged variable makes sense because it pertains to the unobserved 11th time period. To lag `x` two time periods, we similarly add two NAs to the beginning of the vector and remove two values from the end, retaining the first eight values of `x`:

```
(x.lag2 <- c(rep(NA, 2), x[1:8]))
[1] NA NA 1 2 3 4 5 6 7 8
```

If we were to use `x.lag2` in a regression, for example, the first two time periods would be lost, which is as it should be because no lagged values of `x` were observed for these periods.

There is a `lag()` function in base R that is used with time-series objects, but it is not suitable for use with linear regression. We ignore that function and write our own function, `Lag()`, to perform the operation. We name our `Lag()` function with an uppercase “L” to avoid shadowing the standard `lag()` function; see [Section 2.3.1](#). Here's a simple definition of `Lag()` and a few examples of its use:

```

Lag <- function(x, lag=1) {
  c(rep(NA, lag), x[1:(length(x) - lag)])
}

Lag(x)
[1] NA 1 2 3 4 5 6 7 8 9

Lag(x, lag=2)
[1] NA NA 1 2 3 4 5 6 7 8

Lag(letters, 2)
[1] NA NA "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
[16] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x"

```

As explained in [Section 1.2.7](#), the *special form* function () returns a function as its result, which we assign to Lag. The *formal arguments* of the Lag () function are x and lag. Setting lag=1 in the function definition establishes 1 as the *default value* of the lag argument, the value used if the user doesn't specify the lag argument when Lag () is called. The argument x has no default, and so to invoke the Lag () function, a user must supply a value for x. Function arguments may be specified by position, by name, or both as illustrated in these examples.

The examples demonstrate that Lag () works with numeric vectors and with character vectors, such as the built-in variable letters, but the function fails when the argument x is a factor or if the lag is negative (called a *lead*):

```

set.seed(123) # for reproducibility
(fac <- factor(sample(c("a", "b", "c"), 20, replace=TRUE)))
[1] a c b c c a b c b b c b c b a c a a a c
Levels: a b c

Lag(fac, 2) # incorrect!
[1] NA NA 1 3 2 3 3 1 2 3 2 2 3 2 3 2 1 3 1 1

Lag(1:10, -1) # fails!
Error in rep(NA, lag): invalid 'times' argument

```

[Figure 10.1](#) displays an elaborated version of Lag () that works in both of these circumstances. Applying the new definition of Lag ():

**Figure 10.1** A more sophisticated version of the Lag () function.

```
Lag <- function(x, lag=1){  
  # lag a variable, padding with NAs  
  # x: variable to lag  
  # lag: number of observations to lag (negative = lead)  
  if (!(is.vector(x) || is.factor(x)))  
    stop ("x must be a vector or a factor")  
  if (!(length(lag) == 1 && floor(lag) == lag))  
    stop ("lag must be an integer")  
  length.x <- length(x)  
  if (abs(lag) > length.x) stop("|lag| > length of x")  
  result <- if (lag == 0) x  
           else if (lag > 0) c(rep(NA, lag),  
                           x[1:(length.x - lag)])  
           else {  
             lag <- abs(lag)  
             c(x[(lag + 1):length.x], rep(NA, lag))  
           }  
  if (is.factor(x)) result <- factor(levels(x)[result])  
  result  
}
```

```

Lag(1:10)      # works as before

[1] NA  1  2  3  4  5  6  7  8  9

Lag(1:10, 2)  # ditto

[1] NA NA  1  2  3  4  5  6  7  8

Lag(1:10, -2) # lead 2 periods, works

[1]  3  4  5  6  7  8  9 10 NA NA

Lag(1:10, 0)  # works

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

Lag(letters)  # works as before

[1] NA  "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
[16] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y"

Lag(fac, 2)   # lag a factor, works

[1] <NA> <NA> a     c     b     c     c     a     b     c     b     b
[13] c     b     c     b     a     c     a     a
Levels: a b c

Lag(1:10, 1.5) # error!

Error in Lag(1:10, 1.5): lag must be an integer

Lag(1:10, 11) # error!

Error in Lag(1:10, 11): |lag| > length of x
The function in Figure 10.1 includes several programming features:


- As is typical, the body of the function comprises several R commands, which are enclosed in curly braces, { }. The value returned by the function is the value of the last command executed, in this case the value of the local variable result. Although not used here, the last line of the function could have been return (result) to make the returned value explicit.
- We include comments at the top of the function to explain what the function does and what each argument represents. This simple form of documentation helps us to recall how the function works when we return to it after a period of time. Comments are never required in R but are good programming practice whenever the code isn't self-explanatory. On the other hand, if you employ sound programming style—such as using descriptive variable names, keeping functions brief by breaking a programming problem into its essential components, and avoiding “clever”

```

but opaque programming tricks—you’ll minimize the need for extensive comments.

- The *conditional if* is used to perform reality checks on the input (i.e., the arguments `x` and `lag`), to determine what to do when the `lag` is zero, positive, or negative, and to treat factors properly.
- The `stop()` function is called to signal an error and print an informative error message, for example, if the argument `x` isn’t a vector or a factor.
- The `floor()` function rounds `lag` to the next smallest whole number, and so `floor(lag) == lag` is `TRUE` if `lag` is a whole number (i.e., an “integer”<sup>3</sup>), as it should be, and `FALSE` otherwise.

<sup>3</sup>\* We use the term *integer* informally here, because in R, a number like 2 is represented as a floating-point number, not technically as an integer.

## 10.2.2 Creating an Influence Plot

Unusual-data diagnostics for linear models are discussed in [Section 8.3](#), using Duncan’s occupational-prestige regression as an example. Recall from this discussion that *hat-values* for a linear model measure the leverage of each case on the least-squares fit, *Studentized residuals* show the outlyingness of each case, and *Cook’s distances* measure the impact on the regression coefficients of omitting each case in turn. These three diagnostic statistics can be visualized in a single graph using the horizontal axis for hat-values, the vertical axis for Studentized residuals, and the size of the plotted points for Cook’s distances.<sup>4</sup>

<sup>4</sup> The `influencePlot()` function in the `car` package can construct this graph. The discussion in this section shows, in effect, how we developed the `influencePlot()` function, which is somewhat more flexible than the function that we present here. For example, `influencePlot()` tries to prevent point labels from extending beyond the boundaries of the plotting region, as one label does in [Figure 10.2](#).

```
library("carData") # for the Duncan data set
model <- lm(prestige ~ income + education, data=Duncan)
hat <- hatvalues(model) # horizontal axis variable
rstud <- rstudent(model) # vertical axis variable
```

```

cook <- sqrt(cooks.distance(model)) # size of plotted points
plot(hat, rstud, cex=10*cook,
      xlab="Hat-values", ylab="Studentized Residuals")
abline(h=c(-2, 0, 2), lty="dashed")
abline(v=c(2, 3)*length(coef(model))/length(rstud),
      lty="dashed")
noteworthy <- which(abs(rstud) > 2 |
      hat > 2*length(coef(model))/length(rstud))
text(hat[noteeworthy], rstud[noteeworthy],
     names(rstud)[noteeworthy])

```

The result of these commands is shown in [Figure 10.2](#). The code for the script is straightforward:

- We use the hatvalues (), rstudent (), and cooks.distance () functions to compute the various diagnostic quantities from the fitted model.
- We then call plot () to create a scatterplot of Studentized residuals versus hat-values, with the sizes of the points, specified via the cex (character expansion) argument, proportional to the square root of the Cook's distances, making the *areas* of the points proportional to Cook's distances. The factor 10 is arbitrary: Its purpose is to scale the circles so that they are large enough to discern size differences.
- The abline () function is used to draw horizontal and vertical reference lines on the graph, at  $\pm 2$  on the Studentized-residual scale, and at twice and three times the average hat-value, which is the number of coefficients in the model,  $k + 1$ , divided by the sample size,  $n$ .
- We define cases as noteworthy if their Studentized residuals are outside the range  $[-2, 2]$  or if their hat-values exceed twice the average hat-value, and we use the text () function to display labels for these points. The labels originate in the row names of the Duncan data set, which become the names of the vector of Studentized residuals as a consequence of setting data=Duncan in the call to lm ()�.

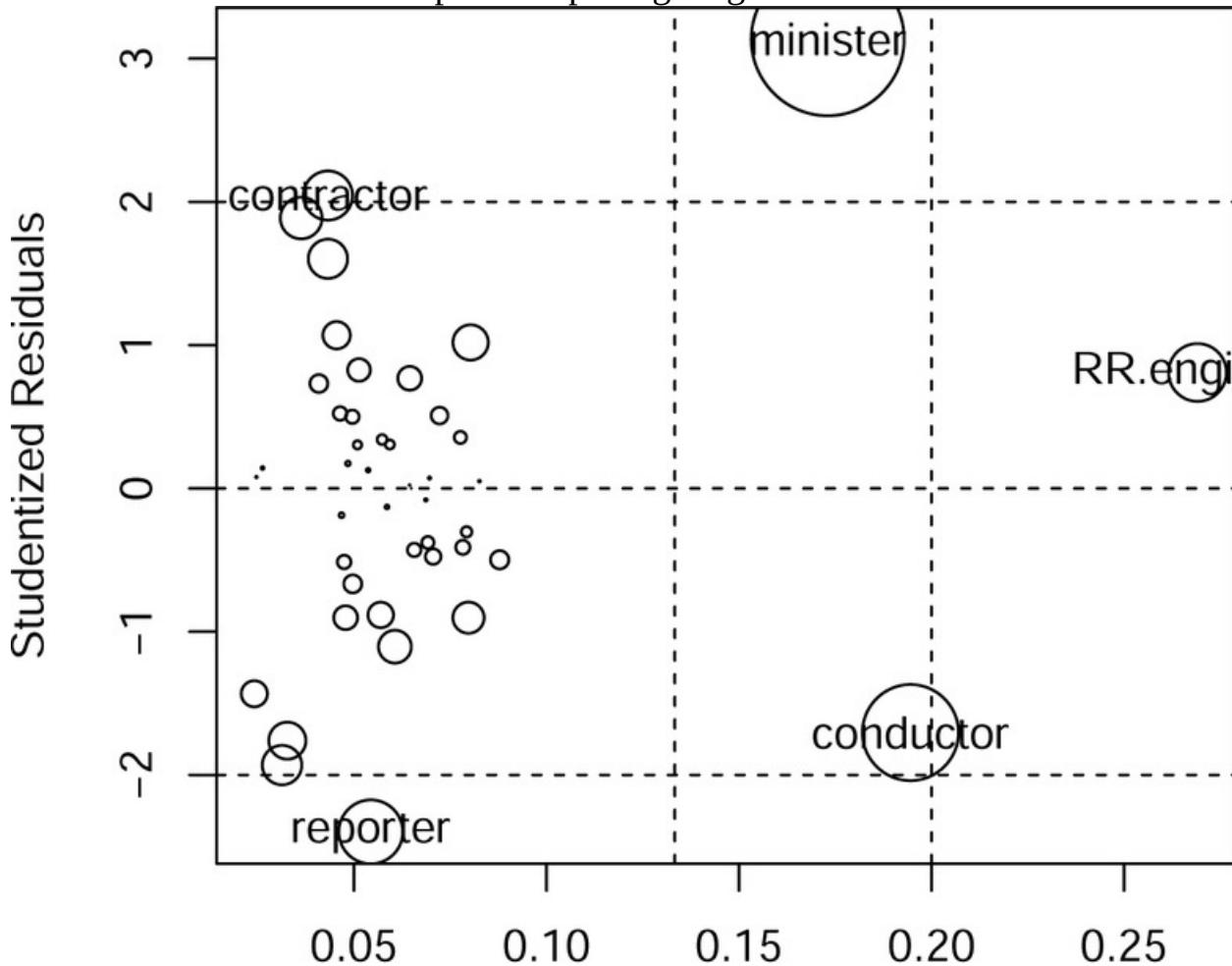
Making our script into a function is simplicity itself. The only input required is the linear-model object, and so our function needs only one argument. RStudio automates this process: Just use the mouse to block the relevant part of the script, from the hatvalues () command through the text () command; select *Code > Extract Function* from the RStudio menus; and, when asked, supply the name of the function to be created. We used the name inflPlot, producing the code in panel (a) of [Figure 10.3](#).

We could stop here, but instead we modify the function to make the scale factor

for the Cook's distances a second, optional, argument to the function, with the default value 10, and to return the names of any noteworthy cases (or, if there are no noteworthy cases, to invisibly return a NULL value).<sup>5</sup> In addition, we include the special ellipses argument ... discussed below and add a few comments explaining what the function does and what its arguments mean. The elaborated version of inflPlot () is shown in panel (b) of [Figure 10.3](#).

<sup>5</sup> Like all R functions, inflPlot () returns a result even if the result is an invisible NULL value, but its primary purpose is the *side effect* of creating a graph.

**Figure 10.2** Influence plot of hat-values, Studentized residuals, and Cook's distances for Duncan's occupational-prestige regression.



### Hat-values

The command inflPlot (model) creates the same graph as before but now returns the names of the noteworthy cases:

**inflPlot (model)**

```
[1] "minister" "reporter" "conductor" "contractor" [5] "RR.engineer"
```

The real argument `model` in this command, the object representing Duncan's regression, has the same name as the first dummy argument of `inflPlot()`, but names of real and dummy arguments are not generally the same, and we could use the `inflPlot()` function with any linear or generalized linear model object. The special ellipses argument ... allows the user to include other arguments to `inflPlot()` that are then passed to the `plot()` function called by `inflPlot()`. To see what additional arguments are available, look at `help("plot.default")`. For example,

**`inflPlot(model, xlim=c(0, 0.3), ylim=c(-2.5, 3.6))`**

passes the arguments `xlim` and `ylim` to `plot()`, expanding the range of the *x*- and *y*-axes to accommodate the circles and labels (try it!).

**Figure 10.3** The `inflPlot()` function: (a) initial version and (b) elaborated version.

(a)

```
inflPlot <- function(model) {  
  hat <- hatvalues(model)  
  rstud <- rstudent(model)  
  cook <- sqrt(cooks.distance(model))  
  plot(hat, rstud, cex=10*cook,  
    xlab="Hat-values", ylab="Studentized Residuals")  
  abline(h=c(-2, 0, 2), lty="dashed")  
  abline(v=c(2, 3)*length(coef(model))/length(rstud),  
    lty="dashed")  
  noteworthy <- which(abs(rstud) > 2 |  
    hat > 2*length(coef(model))/length(rstud))  
  text(hat[noteeworthy], rstud[noteeworthy],  
    names(rstud)[noteeworthy])  
}
```

(b)

```
inflPlot <- function(model, scale.factor=10, ...) {  
  # Influence Bubble Plot  
  # model: linear or generalized linear model  
  # scale factor: for circles representing Cook's Ds  
  # ...: arguments to pass to plot()  
  hat <- hatvalues(model)  
  rstud <- rstudent(model)  
  cook <- sqrt(cooks.distance(model))  
  plot(hat, rstud, cex=scale.factor*cook,  
    xlab="Hat-values", ylab="Studentized Residuals", ...)  
  abline(h=c(-2, 0, 2), lty="dashed")  
  abline(v=c(2, 3)*length(coef(model))/length(rstud),  
    lty="dashed")  
  noteworthy <- which(abs(rstud) > 2 |  
    hat > 2*length(coef(model))/length(rstud))  
  labels <- names(rstud)[noteeworthy]  
  text(hat[noteeworthy], rstud[noteeworthy], labels)  
  if (length(labels) == 0) invisible(NULL) else labels  
}
```

## 10.3 Working With Matrices\*

R incorporates extensive facilities for matrices and linear algebra, which can be very useful in statistical applications. This section focuses on basic matrix operations.

We begin by defining some matrices via the `matrix()` function, remembering that `matrix()` fills matrices by columns unless the argument `byrow` is set to `TRUE`:

```
(A <- matrix(c(1, 2, -4, 3, 5, 0), nrow=2, ncol=3))
```

```
 [,1] [,2] [,3]
[1,]    1   -4    5
[2,]    2    3    0
```

```
(B <- matrix(1:6, 2, 3))
```

```
 [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
(C <- matrix(c(2, -2, 0, 1, -1, 1, 4, 4, -4), 3, 3,
byrow=TRUE))
```

```
 [,1] [,2] [,3]
[1,]    2   -2    0
[2,]    1   -1    1
[3,]    4    4   -4
```

### 10.3.1 Basic Matrix Arithmetic

Matrix addition, subtraction, negation, and the product of a matrix and a single number (or *scalar*<sup>6</sup>) use the usual arithmetic operators `+`, `-`, and `*`; addition and subtraction require matrices of the same order:

```
A + B # addition
```

```
[,1] [,2] [,3]  
[1,] 2 -1 10  
[2,] 4 7 6
```

```
A - B # subtraction
```

```
[,1] [,2] [,3]  
[1,] 0 -7 0  
[2,] 0 -1 -6
```

```
A + C # error: A and C not of the same order!
```

Error in A + C: non-conformable arrays

```
2*A # scalar times a matrix
```

```
[,1] [,2] [,3]  
[1,] 2 -8 10  
[2,] 4 6 0
```

```
-A # negation
```

```
[,1] [,2] [,3]  
[1,] -1 4 -5  
[2,] -2 -3 0
```

[6](#) We use the term *scalar* in its mathematical sense here, because in R, there are no scalars: Single numbers are represented as one-element vectors.

Using \* to multiply two matrices of the same order forms their elementwise product. The standard *matrix product* is computed with the *inner-product operator*, %\*%, which requires that the matrices be conformable for multiplication:

***dim(A)***

```
[1] 2 3
```

***dim(C)***

```
[1] 3 3
```

**A %\*% C # matrix product**

```
 [,1] [,2] [,3]
[1,]    18   22  -24
[2,]     7   -7     3
```

***dim(B)***

```
[1] 2 3
```

**A %\*% B # fails!**

Error in A %\*% B: non-conformable arguments  
In matrix products, R vectors are treated as row or column vectors, as required:

```

(a <- rep(1, 3))

[1] 1 1 1

(b <- c(1, 5, 3))

[1] 1 5 3

C %*% a  # matrix times (column) vector

[,1]
[1,]    0
[2,]    1
[3,]    4

a %*% C  # (row) vector times matrix

[,1] [,2] [,3]
[1,]    7    1   -3

a %*% b  # inner product of two vectors

[,1]
[1,]    9

```

The last of these examples illustrates that the inner product of two vectors of the same length, a %\*% b, is a  $1 \times 1$  matrix in R. Multiplying a vector by a matrix produces a one-column or one-row matrix.

The *outer product* of two vectors may be obtained via the outer () function or the %o% operator:

```
outer(a, b) # outer product
```

```
 [,1] [,2] [,3]
[1,]    1    5    3
[2,]    1    5    3
[3,]    1    5    3
```

```
a %o% b # equivalent
```

```
 [,1] [,2] [,3]
[1,]    1    5    3
[2,]    1    5    3
[3,]    1    5    3
```

The `outer()` function may also be used with operations other than multiplication; an optional third argument, which defaults to "`*`"<sup>7</sup> specifies the function (or binary operator) to be applied to pairs of elements from the first two arguments; for example:

```
outer(1:3, 1:3, ">=")
```

```
 [,1] [,2] [,3]
[1,] TRUE FALSE FALSE
[2,] TRUE  TRUE FALSE
[3,] TRUE  TRUE  TRUE
```

<sup>7</sup> The `*` operator must be quoted using double quotes, single quotes, or back-ticks, when supplied as an argument because it isn't a standard R name; see [Section 1.2.4](#).

The function `t()` returns the *transpose* of a matrix, exchanging rows and columns:

```
t(B) # transpose
```

```
[,1] [,2]  
[1,] 1 2  
[2,] 3 4  
[3,] 5 6
```

### 10.3.2 Matrix Inversion and the Solution of Linear Simultaneous Equations

The solve () function computes the inverse of a square nonsingular matrix:

```
solve(C) # matrix inverse
```

```
[,1] [,2] [,3]  
[1,] 0.0 0.5 0.125  
[2,] -0.5 0.5 0.125  
[3,] -0.5 1.0 0.000
```

```
solve(C) %*% C # check
```

```
[,1] [,2] [,3]  
[1,] 1 0 0  
[2,] 0 1 0  
[3,] 0 0 1
```

The fractions () function in the MASS package (Venables & Ripley, 2002), part of the standard R distribution, may be used to display numbers as rational fractions, which is often convenient when working with simple matrix examples:

```

library("MASS")
fractions(solve(C))

 [,1] [,2] [,3]
[1,] 0 1/2 1/8
[2,] -1/2 1/2 1/8
[3,] -1/2 1 0

```

The `solve()` function may also be used more generally to solve systems of linear simultaneous equations—hence the name of the function. If  $\mathbf{C}$  is a known square nonsingular matrix,  $\mathbf{b}$  is a known conformable vector or matrix, and  $\mathbf{x}$  is a vector or matrix of unknowns, then the solution of the system of linear simultaneous equations  $\mathbf{Cx} = \mathbf{b}$  is  $\mathbf{x} = \mathbf{C}^{-1} \mathbf{b}$ , which is computed in R by

```
(x <- solve(C, b)) # solution of Cx = b
[1] 2.875 2.375 4.500
```

as we may easily verify:

**C %\*% x**

```

 [,1]
[1,] 1
[2,] 5
[3,] 3

```

**b # check**

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

### 10.3.3 Example: Linear Least-Squares Regression

To illustrate the application of basic matrix operations in R, we compute the least-squares coefficients for a linear model with model-matrix  $\mathbf{X}$  and response-vector  $\mathbf{y}$ , using Duncan's occupational-prestige data as an example:

```

x <- cbind(1, as.matrix(Duncan[ , 2:3]))
  # model matrix with constant
colnames(x) [1] <- "intercept"
y <- Duncan[ , "prestige"] # the response vector
head(x) # first 6 rows

```

|            | intercept | income | education |
|------------|-----------|--------|-----------|
| accountant | 1         | 62     | 86        |
| pilot      | 1         | 72     | 76        |
| architect  | 1         | 75     | 92        |
| author     | 1         | 55     | 90        |
| chemist    | 1         | 64     | 86        |
| minister   | 1         | 21     | 84        |

```
head(y)
```

```
[1] 82 83 90 76 90 87
```

Selecting the single column for the response prestige from the Duncan data frame produces a vector rather than a one-column matrix because R by default drops dimensions with extent one. We can circumvent this behavior by specifying drop=FALSE (see [Section 2.4.4](#) on indexing), which not only produces a matrix with one column but also retains the row labels:<sup>8</sup>

```
head(Duncan[ , "prestige", drop=FALSE])
```

|            | prestige |
|------------|----------|
| accountant | 82       |
| pilot      | 83       |
| architect  | 90       |
| author     | 76       |
| chemist    | 90       |
| minister   | 87       |

<sup>8</sup> Actually `Duncan[, "prestige", drop=FALSE]` produces a one-column data

frame, but the distinction isn't important in the current context. Although some R functions are fussy about the distinction between a vector and a single-column matrix, in the current example, either will do. The usual formula for the least-squares coefficients is  $\mathbf{b} = (\mathbf{X}'\mathbf{X})^{-1} \mathbf{X}'\mathbf{y}$ . It is simple to translate this formula directly into an R expression:

```
solve(t(X) %*% X) %*% t(X) %*% y
```

```
[,1]
```

```
intercept -6.06466
income      0.59873
education   0.54583
```

Forming and inverting  $\mathbf{X}'\mathbf{X}$  is inefficient and may in some cases lead to excessive rounding errors. The `lm()` function, for example, uses a mathematically equivalent but numerically superior approach to least-squares problems based on the *QR decomposition* (Chambers, 1992; Weisberg, 2014, Appendix A.9). In our example, however, the direct approach provides the same answer as `lm()` to at least five digits:

```
coef(lm(prestige ~ income + education, data=Duncan))
```

|             | income   | education |
|-------------|----------|-----------|
| (Intercept) | -6.06466 | 0.59873   |
|             |          | 0.54583   |

### 10.3.4 Eigenvalues and Eigenvectors

The `eigen()` function calculates *eigenvalues* and *eigenvectors* of square matrices, including asymmetric matrices, which may have complex eigenvalues and eigenvectors. For example, an eigen-analysis of the correlation matrix for the predictors in Duncan's occupational-prestige regression is provided by the following commands:

```

(R <- with(Duncan,
            cor(cbind(income, education)))) # correlations

           income education
income      1.00000   0.72451
education  0.72451   1.00000

eigen(R) # eigenvalues and eigenvectors

eigen() decomposition
$values
[1] 1.72451 0.27549

$vectors
[,1]      [,2]
[1,] 0.70711 -0.70711
[2,] 0.70711  0.70711

```

The eigenvectors are the columns of the list-component \$vectors returned by eigen(); each eigenvector is normalized to length 1 and therefore the eigenvectors are the *loadings* for a *principal-components analysis* based on the correlations, while the eigenvalues give the collective variation accounted for by each component. Principal-components analysis can also be performed by the standard R functions princomp() and prcomp().

### 10.3.5 Miscellaneous Matrix Computations

The *determinant* of a square matrix may be computed by the det() function; for example:

```

det(R)
[1] 0.47508

```

Depending on its argument, the diag() function may be used to extract or to set the *main diagonal* of a matrix, to create a *diagonal matrix* from a vector, or to create an *identity matrix* of specified order:

```

diag(R)          # extract diagonal

    income education
      1           1

diag(R) <- NA      # set diagonal

R

    income education
income           NA   0.72451
education  0.72451       NA

(D <- diag(1:3))  # make diagonal matrix

      [,1] [,2] [,3]
[1,]     1     0     0
[2,]     0     2     0
[3,]     0     0     3

(I3 <- diag(3))  # order-3 identity matrix

      [,1] [,2] [,3]
[1,]     1     0     0
[2,]     0     1     0
[3,]     0     0     1

```

Beyond eigenvalues and eigenvectors, matrix factorizations available in R include the *singular-value decomposition*, the *QR decomposition*, and the *Cholesky decomposition*: See help ("svd"), help ("qr"), and help ("chol") for further references and usage.

The **MASS** package supplies a function, `ginv()`, for computing *generalized inverses* of square and rectangular matrices. Advanced facilities for matrix computation, including for efficiently storing and manipulating large sparse matrices, are provided by the **Matrix** package (Bates & Maechler, 2018), which

is part of the standard R distribution, and various functions for studying and visualizing matrix and linear algebra are provided by the **matlib** package (Friendly & Fox, 2018).

## 10.4 Program Control With Conditionals, Loops, and Recursion

### 10.4.1 Conditionals

The basic construct for conditional evaluation in R is the if statement, which we've already encountered and takes one of the following two general forms:

1. if (*logical-condition*) *command*
2. if (*logical-condition*) *command* else *alternative-command*

If the first element of *logical-condition* evaluates to TRUE or to a nonzero number, then *command* is evaluated and its value is returned.

If *logical-condition* evaluates to FALSE or 0 in the first form, then NULL is returned.

If *logical-condition* evaluates to FALSE or 0 in the second form, then *alternative-command* is evaluated and its value returned.

In either case, *command* (or *alternative-command*) may be a compound R command, with the elementary commands that compose it enclosed in braces and separated by semicolons or new lines; when a compound command is evaluated, the value returned is the value of its last elementary command.

The following function returns the absolute value of a number:

```
abs1 <- function(x) {  
  if(x < 0) -x else x  
}
```

```
abs1(-5)
```

```
[1] 5
```

```
abs1(5)
```

```
[1] 5
```

The standard abs () function serves the same purpose, and abs1 () and other functions in this section are simply intended to illustrate how the various control structures in R work.

When `abs1()` is applied to a vector, it does not produce the result that we probably intended, because only the first element in the condition `x < 0` is used. In the illustration below, the first element of the vector argument is less than zero, and so the condition evaluates to TRUE; the value returned by the function is therefore the negative of the input vector. A warning message is also printed by R:

```
abs1(-3:3) # wrong! the first element, -3, controls the result
[1] 3 2 1 0 -1 -2 -3
Warning message:
In if (x < 0) -x else x :
  the condition has length > 1
  and only the first element will be used
```

The `ifelse()` function in R, discussed in [Section 2.3.3](#) in the context of recoding variables, provides a *vectorized* conditional, as required here:

```
abs2 <- function(x) {
  ifelse(x < 0, -x, x)
}
abs2(-3:3)
```

```
[1] 3 2 1 0 1 2 3
```

The general format of `ifelse()` is

`ifelse(vector-condition, true-vector, false-vector)`

The three arguments of `ifelse()` are all vectors of the same length, although a scalar value for *true-vector* or *false-vector* will be extended by repetition to the length of *vector-condition*. Wherever an element of *vector-condition* is TRUE, the corresponding element of *true-vector* is selected, and where *vector-condition* is FALSE, the corresponding element of *false-vector* is selected.

More complex conditionals can be handled by *cascading* if/else statements. For example, the following function returns -1, 0, or 1 according to the sign of a number, negative, zero, or positive:

```
sign1 <- function(x) {  
  if (x < 0) -1  
  else if (x > 0) 1  
  else 0  
}  
sign1(-5)
```

```
[1] -1
```

The indentation of the code reveals the logical structure of the function. R does not enforce rules for laying out functions in this manner, but programs are more readable if the program code is properly indented.<sup>9</sup>

[9](#) Among its other features, the code editor in RStudio automatically indents R code.

The same technique may be applied to the **ifelse** () function, for example, to provide a vector of signs:

```
sign2 <- function(x) {  
  ifelse (x < 0, -1,  
          ifelse(x > 0, 1, 0))  
}  
sign2(c(-5, 0, 10))
```

```
[1] -1 0 1
```

Once again, this is an artificial example, because the same functionality is provided by the standard **sign** () function.

Complex conditions can also be handled by the **switch** () function; for example:

```
convert2meters <- function(x,  
  units=c("inches", "feet", "yards", "miles")) {  
  units <- match.arg(units)  
  switch(units,  
    inches = x * 0.0254,
```

```

feet = x * 0.3048,
yards = x * 0.9144,
miles = x * 1609.344

)
}

```

The default value of the argument units seems to be a vector of values, c ("inches", "feet", "yards", "miles"). The match.arg () function in the body of convert2meters () selects one of these elements. If the user doesn't specify units, then the first element, in this case "inches", is returned as the default value of the argument. If the user provides one of the other elements in the list, or an abbreviation that uniquely determines the element, then units is set equal to that value. Finally, if the user specifies a value of units that matches *none* of the elements of the vector, then an error results, and so match.arg () serves to limit the permissible choices for a function argument to a predefined set.

The switch () function picks which branch to execute according to the value of the units argument; for example:

```

convert2meters(10)  # first value of units ("inches") default
[1] 0.254
convert2meters(10, units="inches")  # equivalent to the default
[1] 0.254
convert2meters(3, "feet")
[1] 0.9144
convert2meters(100, "y")  # abbreviation of "yards"
[1] 91.44
convert2meters(5, "miles")
[1] 8046.7
convert2meters(3, "fathoms")  # produces an error!
Error in match.arg(units) :
  'arg' should be one of "inches", "feet", "yards", "miles"

```

## 10.4.2 Iteration (Looping)

The for, while, and repeat statements are used to implement loops in R, which execute blocks of commands *iteratively* (repetitively).

Consider computing the factorial of a nonnegative integer,  $n$ , usually denoted  $n!$ :

(10.1)

$$n! = n \times (n - 1)!$$

$$= n \times (n - 1) \times \cdots \times 2 \times 1 \text{ for } n \geq 1$$

$$0! = 1$$

```
fact1 <- function(n) {
  if (n <= 1) return(1)
  f <- 1 # initialize
  for (i in 1:n) {
    f <- f*i # accumulate product
  }
  f # return result
}
fact1(5)
```

```
[1] 120
```

This, too, is an artificial problem because of the standard factorial () function. In fact1(), we initialize the local variable f to 1, then accumulate the factorial product in the for loop, and finally implicitly return the accumulated product as the result of the function. As we mentioned, it is also possible to return a result explicitly—for example, return (1) in the first line of the fact1 () function—which causes execution of the function to cease at that point. The body of the for loop, enclosed in curly braces, consists of only one command in this example, which is atypical. If there is more than one command in the body of the loop, the commands appear on separate lines (or may be separated by semicolons).<sup>10</sup>

<sup>10</sup> When, as here, the body of the loop comprises a single command, the curly braces may be omitted and the loop may appear on a single line of code: for (i in 1:n) f <- f\*i.

The function fact1 () does not verify that its argument is a nonnegative integer,

and so, for example,

```
fact1(5.2) # wrong!
[1] 120
```

returns  $5! = 120$  rather than reporting an error. This incorrect result occurs because `1:5.2` expands to the values `1, 2, 3, 4, 5`, effectively replacing the argument `5.2` to `:` by truncating it to `5`. To return an incorrect answer would be unacceptable in a general-purpose program. For a quick-and-dirty program written for our own use, however, this might be acceptable behavior. We would only need to ensure that the argument to the function is always a nonnegative integer.

Here is another version of the program, adding an error check for the input:

```
fact2 <- function(n) {
  if ((!is.numeric(n)) || (n != floor(n))
    || (n < 0) || (length(n) > 1))
    stop("n must be a nonnegative integer")
  if (n <= 1) return(1)
  f <- 1 # initialize
  for (i in 1:n) {
    f <- f*i # accumulate product
  }
  f # return result
}
fact2(5)
[1] 120
fact2(5.2)
```

Error in `fact2(5.2)`: `n` must be a nonnegative integer  
The `stop()` function ends execution of `fact2()` immediately and prints its argument as an error message.

The double-or operator `||` used in `fact2()` differs from `|` in two respects:

1. `|` applies elementwise to vectors, while `||` takes single-element logical operands.
2. `||` evaluates its right operand only if its left operand is FALSE. This process,

called *lazy evaluation*, can be exploited to prevent the evaluation of an expression that would otherwise cause an error. In the illustration, for example,  $x \neq \text{floor}(x)$  is not evaluated if  $x$  is not numeric; if  $x$  were nonnumeric, then evaluating  $\text{floor}(x)$  would produce an error.

Analogous comments apply to the unvectorized double-and operator `&&`: The right operand of `&&` is evaluated only if its left operand is TRUE.

The general format of the for statement is

`for (loop-variable in values) command`

In executing the loop, *loop-variable* successively takes on the values in the vector or list *values*;<sup>11</sup> *command* is usually (but not in the preceding example) a compound command enclosed in braces, { }, and is evaluated each time through the loop using the current value of *loop.variable*.

[11](#) It's worth emphasizing that the loop variable need not be a simple counter, as it is in this example. Because lists can contain any objects, including other lists, we can loop over a list of complex objects, such as a list of linear-model objects. This can be a very powerful programming technique.

In contrast, while loops iterate as long as a specified condition holds TRUE; for example:

```
fact3 <- function(n) {
  if ((!is.numeric(n)) || (n != floor(n))
    || (n < 0) || (length(n) > 1))
    stop("n must be a nonnegative integer")
  i <- f <- 1 # initialize
  while (i <= n) {
    f <- f*i      # accumulate product
    i <- i + 1    # increment counter
  }
  f # return result
}
fact3(5)
[1] 120
```

As before, the indentation of the lines in the function reflects its logical

structure, such as the scope of the while loop.

The general format of a while loop is

while (*logical-condition*) *command*

where *command*, which is typically a compound command and therefore enclosed in braces, is executed as long as *logical-condition* holds.

Finally, repeat loops iterate until a break command is executed, as illustrated in the function fact4():

```
fact4 <- function(n) {  
  if ((!is.numeric(n)) || (n != floor(n))  
    || (n < 0) || (length(n) > 1))  
    stop("n must be a nonnegative integer")  
  i <- f <- 1 # initialize  
  repeat {  
    f <- f*i # accumulate product  
    i <- i + 1 # increment counter  
    if (i > n) break # termination test  
  }  
  f # return result  
}  
  
fact4(5)
```

```
[1] 120
```

The general format of a repeat loop is simply

repeat { *command* }

The *command* must be compound, and one of its components must be a termination test that calls break, or else the repeat could continue forever.<sup>12</sup>

[12](#) The break command can also be used to exit from a for or while loop.

### 10.4.3 Recursion

*Recursive functions* call themselves. Recursion is permissible in R and can provide an elegant alternative to looping. Some problems that yield readily to the “divide-and-conquer” strategy of recursion are difficult to solve in other ways.

A recursive definition of the factorial function is very similar to the mathematical definition of the factorial (Equation 10.1 on page 495). For

simplicity, we forgo error checking:<sup>13</sup>

<sup>13</sup>\* As an alternative to calling `fact5(n - 1)` within the definition of `fact5()`, we can instead use the special `Recall()` function: `Recall(n - 1)`. Using `Recall()` is in a sense safer, because the name of a function in R isn't immutable. For example, if we assigned `fact <- fact5` and then erased `fact5()`, `fact()` (which calls `fact5()`) would no longer work, but had we used `Recall()` in the original definition of `fact5()`, `fact()` would still work in the absence of `fact5()`. Because this is an unusual circumstance, we typically prefer the greater clarity of an explicit recursive call to use of `Recall()`.

```
fact5 <- function(n) {  
  if (n <= 1) 1           # termination condition  
  else n * fact5(n - 1)  # recursive call  
}  
  
fact5(5)  
  
[1] 120
```

This recursive implementation of the factorial relies on the properties  $n! = n \times (n - 1)!$  for  $n > 1$ , and  $0! = 1! = 1$ . We can use the `trace()` function to report the recursive calls, followed by `untrace()` to turn off tracing:

```
trace(fact5)  
  
fact5(5)  
  
trace: fact5(5)  
trace: fact5  
trace: fact5  
trace: fact5  
trace: fact5  
  
[1] 120
```

```
untrace(fact5)
```

## 10.5 Avoiding Loops: `apply()` and Its Relatives

Avoiding loops and recursion can make R programs more compact, easier to

read, and sometimes more efficient. Matrix functions and operators, as well as vectorized functions, encourage us to write loopless expressions for tasks that might require loops in lower-level programming languages such as Fortran and C.

The `apply()` function, and its relatives `lapply()`, `sapply()`, `mapply()`, `tapply()`, and `Tapply()`,<sup>14</sup> can also help us to avoid loops or recursion. In addition, the `Vectorize()` function uses `mapply()` to create a vectorized version of a function that normally takes scalar arguments.

<sup>14</sup> For a complete list of functions in the `apply()` family, enter the command `apropos ("^apply")`. The `Tapply()` function is from the **car** package; the other functions in the `apply()` family are part of the standard R distribution.

The `apply()` function invokes or *applies* another function over specified coordinates of an array. Although this is a useful facility for manipulating higher-dimensional arrays such as multiway contingency tables, in most instances, the array in question is a matrix or data frame, with the latter treated as a matrix.

Consider the data frame `DavisThin` in the **carData** package,<sup>15</sup> representing the responses of 191 subjects to a 7-item *drive for thinness* scale, collected as part of a study of eating disorders. Each item is scored from zero to 3, and the scale is to be formed by adding the seven individual items (DT1 through DT7) for each subject:

```
head(DavisThin, 10) # first 10 rows
```

|    | DT1 | DT2 | DT3 | DT4 | DT5 | DT6 | DT7 |
|----|-----|-----|-----|-----|-----|-----|-----|
| 1  | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 2  | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 3  | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 4  | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 5  | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 6  | 0   | 1   | 0   | 0   | 0   | 0   | 0   |
| 7  | 0   | 2   | 2   | 0   | 2   | 2   | 0   |
| 8  | 2   | 3   | 3   | 2   | 3   | 3   | 3   |
| 9  | 0   | 0   | 0   | 0   | 3   | 0   | 0   |
| 10 | 3   | 3   | 2   | 1   | 3   | 3   | 0   |

```
dim(DavisThin)
```

```
[1] 191 7
```

[15](#) We're grateful to Caroline Davis of York University in Toronto, who contributed this data set.

We can calculate the scale score for each subject by applying the sum () function over the rows (the first coordinate) of the data frame:

```
DavisThin$thin.drive <- apply (DavisThin, 1, sum)
```

```
head (DavisThin$thin.drive, 10)
```

```
[1] 0 0 0 0 0 1 8 19 3 15
```

We have chosen to add a variable called thin.drive to the data frame, in a new eighth column of DavisThin.

Similarly, if we are interested in the column means of the data frame, they may be simply calculated as follows, by averaging over the second (column) coordinate:

```
apply(DavisThin, 2, mean) # variable (column) means
```

|         | DT1            | DT2     | DT3     | DT4     | DT5 |
|---------|----------------|---------|---------|---------|-----|
| 0.46597 | 1.02094        | 0.95812 | 0.34031 | 1.10995 |     |
| DT6     | DT7 thin.drive |         |         |         |     |
| 0.93194 | 0.56545        | 5.39267 |         |         |     |

In these two simple applications, we can more efficiently use the functions rowSums () and colMeans (); for example:

```
colMeans(DavisThin) # equivalent
```

|         | DT1            | DT2     | DT3     | DT4     | DT5 |
|---------|----------------|---------|---------|---------|-----|
| 0.46597 | 1.02094        | 0.95812 | 0.34031 | 1.10995 |     |
| DT6     | DT7 thin.drive |         |         |         |     |
| 0.93194 | 0.56545        | 5.39267 |         |         |     |

There are similar colSums () and rowMeans () functions.

To extend the example, imagine that some items composing the scale are missing for certain subjects; to simulate this situation, we will eliminate thin.drive from the data frame and arbitrarily replace some of the values with NAs:

```
DavisThin$thin.drive <- NULL # remove thin.drive  
DavisThin[1, 2] <- DavisThin[2, 4] <- DavisThin[10, 3] <- NA  
head(DavisThin, 10)
```

|    | DT1 | DT2 | DT3 | DT4 | DT5 | DT6 | DT7 |
|----|-----|-----|-----|-----|-----|-----|-----|
| 1  | 0   | NA  | 0   | 0   | 0   | 0   | 0   |
| 2  | 0   | 0   | 0   | NA  | 0   | 0   | 0   |
| 3  | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 4  | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 5  | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 6  | 0   | 1   | 0   | 0   | 0   | 0   | 0   |
| 7  | 0   | 2   | 2   | 0   | 2   | 2   | 0   |
| 8  | 2   | 3   | 3   | 2   | 3   | 3   | 3   |
| 9  | 0   | 0   | 0   | 0   | 3   | 0   | 0   |
| 10 | 3   | 3   | NA  | 1   | 3   | 3   | 0   |

If we simply apply sum () over the rows of the data frame, then the result will be missing for cases with any missing items, as we can readily verify:

```
head(apply(DavisThin, 1, sum), 10)
```

| 1  | 2  | 3 | 4 | 5 | 6 | 7 | 8  | 9 | 10 |
|----|----|---|---|---|---|---|----|---|----|
| NA | NA | 0 | 0 | 0 | 1 | 8 | 19 | 3 | NA |

A simple alternative is to average over the items that are present, multiplying the resulting mean by 7 to restore zero to 21 as the range of the scale. This procedure can be implemented by defining an *anonymous function* in the call to apply ():

```
head(apply(DavisThin, 1,
function(x) 7 * mean(x, na.rm=TRUE)), 10)
```

| 1      | 2     | 3     | 4     | 5     | 6     | 7     | 8      | 9     |
|--------|-------|-------|-------|-------|-------|-------|--------|-------|
| 0.000  | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 8.000 | 19.000 | 3.000 |
| 10     |       |       |       |       |       |       |        |       |
| 15.167 |       |       |       |       |       |       |        |       |

The anonymous function has a single arbitrarily named argument, x, which will correspond to a row of the data frame; thus, in this case, we compute the mean of the nonmissing values in a row and then multiply by 7, the number of items in the scale. The anonymous function disappears after apply () is executed. Suppose that we are willing to work with the average score if more than half of the seven items are valid but want the scale to be NA if there are four or more missing items:

```
DavisThin[1, 2:5] <- NA # create more missing data
head(DavisThin, 10)
```

|   | DT1 | DT2 | DT3 | DT4 | DT5 | DT6 | DT7 |
|---|-----|-----|-----|-----|-----|-----|-----|
| 1 | 0   | NA  | NA  | NA  | NA  | 0   | 0   |
| 2 | 0   | 0   | 0   | NA  | 0   | 0   | 0   |
| 3 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 4 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| 5 | 0   | 0   | 0   | 0   | 0   | 0   | 0   |

```

6     0     1     0     0     0     0     0
7     0     2     2     0     2     2     0
8     2     3     3     2     3     3     3
9     0     0     0     0     3     0     0
10    3     3    NA     1     3     3     0

makeScale <- function(items) {
  if (sum(is.na(items)) >= 4) NA
  else 7 * mean(items, na.rm=TRUE)
}

head(apply(DavisThin, 1, makeScale), 10)

  1      2      3      4      5      6      7      8      9
NA  0.000  0.000  0.000  0.000  1.000  8.000 19.000 3.000
10

```

15.167  
The lapply () and sapply () functions are similar to apply () but reference the successive elements of a list. To illustrate, we convert the data frame DavisThin to a list:

```

thin.list <- as.list(DavisThin)
str(thin.list) # structure of the result

List of 7
$ DT1: int [1:191] 0 0 0 0 0 0 0 2 0 3 ...
$ DT2: int [1:191] NA 0 0 0 0 1 2 3 0 3 ...
$ DT3: int [1:191] NA 0 0 0 0 0 2 3 0 NA ...
$ DT4: int [1:191] NA NA 0 0 0 0 0 2 0 1 ...
$ DT5: int [1:191] NA 0 0 0 0 0 2 3 3 3 ...
$ DT6: int [1:191] 0 0 0 0 0 0 2 3 0 3 ...
$ DT7: int [1:191] 0 0 0 0 0 0 0 3 0 0 ...

```

The list elements are the variables from the data frame. To calculate the mean of each list element eliminating missing data, we use

```
lapply(thin.list, mean, na.rm=TRUE)
```

```
$DT1
```

```
[1] 0.46597
```

```
$DT2
```

```
[1] 1.0263
```

```
$DT3
```

```
[1] 0.95767
```

```
$DT4
```

```
[1] 0.34392
```

```
$DT5
```

```
[1] 1.1158
```

```
$DT6
```

```
[1] 0.93194
```

```
$DT7
```

```
[1] 0.56545
```

In this example, and for use with apply () as well, the argument na.rm=TRUE is passed to the mean () function, so an equivalent (with the output suppressed) is

***lapply (thin.list, function (x) mean (x, na.rm=TRUE)) # equivalent***

The lapply () function returns a list as its result; sapply () works similarly, but tries to simplify the result, in this case returning a vector with named elements:

```
sapply(thin.list, mean, na.rm=TRUE) # simplified
```

| DT1     | DT2     | DT3     | DT4     | DT5     | DT6     | DT7     |
|---------|---------|---------|---------|---------|---------|---------|
| 0.46597 | 1.02632 | 0.95767 | 0.34392 | 1.11579 | 0.93194 | 0.56545 |

The mapply () function is similar to sapply () but is multivariate in the sense that it processes several vector arguments simultaneously. Consider the integrate () function, which approximates definite integrals numerically and evaluates an individual integral.<sup>16</sup> The function dnorm () computes the density function of a standard-normal random variable. To integrate this function between  $-1.96$  and  $1.96$ :

```
(result <- integrate(dnorm, lower=-1.96, upper=1.96))
```

0.95 with absolute error < 1e-11

<sup>16</sup> This example is adapted from Ligges and Fox (2008). For readers unfamiliar with calculus, integration finds areas under curves—in our example, areas under the standard-normal density curve, which are probabilities.

The printed representation of the result of this function is shown above. The class of the returned value result is "integrate" and it is a list with several elements:

```
names(result)
```

```
[1] "value"           "abs.error"      "subdivisions" "message"
[5] "call"
```

The element result\$value contains the approximate value of the integral. To compute areas under the standard-normal density for a number of intervals—such as the adjacent, nonoverlapping intervals  $(-\infty, -3)$ ,  $(-3, -2)$ ,  $(-2, -1)$ ,  $(-1, 0)$ ,  $(0, 1)$ ,  $(1, 2)$ ,  $(2, 3)$ , and  $(3, \infty)$ —we can *vectorize* the computation with mapply ():

```
(low <- c(-Inf, -3:3))
```

```
[1] -Inf    -3    -2    -1     0     1     2     3
```

```
(high <- c(-3:3, Inf))
```

```
[1] -3   -2   -1   0    1    2    3   Inf
(P <- mapply(function(lo, hi) integrate(dnorm, lo, hi)$value,
             lo=low, hi=high))
[1] 0.0013499 0.0214002 0.1359051 0.3413447 0.3413447 0.1359051
[7] 0.0214002 0.0013499
```

```
sum(P)
```

```
[1] 1
```

This is an artificial example because the vectorized pnorm () function is perfectly capable of producing the same result:

***pnorm (high) - pnorm (low)***

```
[1] 0.0013499 0.0214002 0.1359051 0.3413447 0.3413447 0.1359051 [7]  
0.0214002 0.0013499
```

The Vectorize () function employs mapply () to return a vectorized version of a function that otherwise would be capable of handling only scalar arguments. We have to tell Vectorize () which arguments of the function are to be vectorized. For example, to produce a vectorized version of integrate ():

```
Integrate <- Vectorize(  
  function(fn, lower, upper){  
    integrate(fn, lower, upper)$value  
  },  
  vectorize.args=c("lower", "upper")  
)  
Integrate(dnorm, lower=low, upper=high)  
  
[1] 0.0013499 0.0214002 0.1359051 0.3413447 0.3413447 0.1359051  
[7] 0.0214002 0.0013499
```

Finally, tapply () (table apply) applies a function to each cell of a *ragged array*, containing data for a variable cross-classified by one or more factors. We will use Fox and Guyer's data on anonymity and cooperation, introduced in [Section 2.1.2](#), to illustrate; the data are in the Guyer data set in the **carData** package, loaded earlier in the chapter:

***summary (Guyer)***

| cooperation   | condition      | sex          |
|---------------|----------------|--------------|
| Min. :27.0    | anonymous:10   | female:10    |
| 1st Qu.:38.5  | public     :10 | male     :10 |
| Median  :46.5 |                |              |
| Mean    :48.3 |                |              |
| 3rd Qu.:58.8  |                |              |
| Max.   :79.0  |                |              |

The factor condition has levels "anonymous" and "public", the factor sex has levels "female" and "male", and the response, cooperation, is a numeric variable.

We may, for example, use tapply () to calculate the mean cooperation within each combination of levels of condition and sex:

```
with(Guyer, tapply(cooperation,
      list(Condition=condition, Sex=sex), mean))
```

|           | Sex    |      |
|-----------|--------|------|
| Condition | female | male |
| anonymous | 40.2   | 41.6 |
| public    | 57.4   | 54.0 |

The factors by which to cross-classify the data are given in a list as the second argument to tapply (); the names Condition and Sex are supplied optionally for the list elements to label the output. The third argument, the mean () function, is applied to the values of cooperation within each combination of levels of condition and sex.

The Tapply () function in the **car** package works like tapply (), but includes a formula argument that is simpler to use when computing cell statistics as in the last example. Tapply () also automatically labels the factors in the table that it produces:

```
library("car")
Tapply(cooperation ~ condition + sex, mean, data=Guyer)

      sex
condition   female male
  anonymous    40.2 41.6
    public      57.4 54.0
```

### 10.5.1 To Loop or Not to Loop?

Because R code is executed under the control of the R interpreter rather than compiled into an independently executable program, loops that perform many iterations can consume a great deal of computing time.<sup>17</sup> It is not our object to encourage a neurotic avoidance of loops. The primary goal should be to produce clear, correct, and reasonably efficient R code. It is at times difficult to write vectorized code, and vectorized code may be opaque or may yield little or no speed advantage, possibly at the cost of a large increase in memory consumption.

[17](#) This statement is an oversimplification: R compiles loops into *byte code*, which can be interpreted more efficiently than the original R code. It is still true that R loops are less efficient than loops in a language such as C or Fortran that is compiled into machine code.

Moreover, our expectation is that you will be writing R programs primarily for your own use, and that most of these programs will be used once or a small number of times—for example, for data management tasks. Attending simply to the efficiency of your code can be myopic: If it takes you an additional hour to write a program that saves 2 minutes of computing time, you've lost 58 minutes. Computational efficiency is important primarily for programs that will be executed many times or when inefficient code produces intolerably long computing times. It *may* be worth spending an hour of programming time to save 10 hours of computing time, especially if the computation is to be repeated, but you may also be able to do other work while the computer grinds away for 10 hours—as long as the computation is successful!

When you use loops in your R programs, it is advisable to adhere to the following rules:[18](#)

[18](#) The material in this section is adapted from Ligges and Fox (2008), which was written when loops in R were less efficient than they are currently. The rules are still sound, however, as the various examples demonstrate.

## The Prime Directive: Initialize Objects to Full Length Before the Loop

If an element is to be assigned to an object in each iteration of a loop, and if the final length of that object is known in advance, then the object should be initialized to full length prior to the loop. Otherwise, memory has to be allocated and data copied unnecessarily at each iteration, potentially wasting a great deal of time.

To initialize objects, use functions such as logical(), integer(), numeric(), complex(), and character() for vectors of different modes, as well as the more general function vector(), along with the functions matrix() and array().

Consider the following example, in which we introduce three functions, time1(), time2(), and time3(), each assigning values elementwise into an object. For  $i = 1, \dots, 10^5$ , the value  $i^2$  is written into the  $i$ th element of the vector a. In time1(), the vector a is built up element by element in a loop:

```

time1 <- function(n) { # inefficient!
  a <- NULL
  for (i in 1:n) a <- c(a, i^2)
  a
}
system.time(time1(1e5))

```

| user  | system | elapsed |
|-------|--------|---------|
| 11.09 | 0.17   | 11.28   |

We use the `system.time()` function to measure execution time.<sup>19</sup> In `time2()`, the vector `a` is initialized to full length prior to the loop:

```

time2 <- function(n) { # better
  a <- numeric(n)
  for (i in 1:n) a[i] <- i^2
  a
}

```

<sup>19</sup> The `system.time()` function reports three numbers, all in seconds: *User time* is the time spent executing program instructions, *system time* is the time spent using operating-system services (such as file input/output), and *elapsed time* is clock time. Elapsed time is usually approximately equal to, or slightly greater than, the sum of the other two, but because of small measurement errors, this may not be the case.

This program proves to be so much faster than `time1()` that we'll make the problem 100 times larger, evaluating  $10^7$  rather than  $10^5$  squares:

```

system.time(time2(1e7))

```

| user | system | elapsed |
|------|--------|---------|
| 0.54 | 0.00   | 0.54    |

Finally, in `time3()`, `a` is created in a vectorized operation (i.e., without a loop), which is even faster:

```

time3 <- function(n) { # best
  a <- (1:n)^2
  a
}
system.time(time3(1e7))

```

| user | system | elapsed |
|------|--------|---------|
| 0.03 | 0.00   | 0.03    |

## Move Computations Outside the Loop

It is inefficient to perform a computation repeatedly that can be performed once, possibly in a vectorized fashion. Consider the following example in which we apply the trigonometric sin () function to the numbers  $i = 1, \dots, 10^6$  (interpreted as angles in radians; see help ("sin")), and multiply the results by  $2\pi$ :

```

time4 <- function(n) { # (slightly) inefficient!
  a <- numeric(n)
  for (i in 1:n) a[i] <- 2*pi*sin(i)
  a
}
system.time(time4(1e6))

```

| user | system | elapsed |
|------|--------|---------|
| 0.17 | 0.00   | 0.17    |

```

time5 <- function(n) { # better
  a <- numeric(n)
  for (i in 1:n) a[i] <- sin(i)
  2*pi*a # multiplication moved outside of the loop
}
system.time(time5(1e6))

```

| user | system | elapsed |
|------|--------|---------|
| 0.11 | 0.00   | 0.12    |

Reducing computation time by a few hundredths of a second is unimportant, but in functions with many steps, savings like this can add up, and, in this example, a fully vectorized version is even better:<sup>20</sup>

<sup>20</sup> This computation is too fast to time reliably with `system.time()`. That's OK in the current context where we just want to see which implementation is best. We could use the `microbenchmark()` function in the **microbenchmark** package (Mersmann, 2018) to obtain a more accurate timing; we illustrate the use of `microbenchmark()` in [Section 10.6](#).

```
time6 <- function(n) { # best (fully vectorized)
  2*pi*(1:n)
}
system.time(time6(1e6))

      user  system elapsed
      0       0       0
```

## Do Not Avoid Loops Simply for the Sake of Avoiding Loops

Some time ago, a question was posted to the R-help email list asking how to sum a large number of matrices in a list. To simulate this situation, we will use a list of 10,000  $100 \times 100$  matrices, generated randomly:

```
set.seed(12345) # for reproducibility
system.time({
  matrices <- vector(mode="list", length=10000) # preallocate!
  for (i in 1:10000) {
    matrices[[i]] <- matrix(rnorm(10000), 100, 100)
  }
})
      user  system elapsed
      8.30   0.09   8.39
```

One suggestion was to use a loop to sum the matrices as follows, producing simple, straightforward code:

```

system.time({
  S1 <- matrix(0, 100, 100)          # initialize
  for (M in matrices) S1 <- S1 + M  # accumulate sum
})

      user  system elapsed
    0.14    0.08   0.22

```

This solution took maybe a minute to figure out and takes less than a second to run.

In response, someone else suggested the following ostensibly cleverer loop-phobic solution:

```

system.time(S2 <- apply(array(unlist(matrices),
  dim=c(100, 100, 10000)), 1:2, sum)
)

      user  system elapsed
    4.81    0.45   5.27

```

This approach is not only considerably slower but also opaque, probably took much longer to figure out, and wastes a great deal of memory. It's comforting, however, that the two approaches produce the same answer:<sup>21</sup>

<sup>21</sup> We avoided testing for *exact* equality using ==, which is unlikely to hold when we perform theoretically equivalent floating-point arithmetic on a computer in two different ways. The all.equal () function tests for equality within the bounds of rounding error.

***all.equal (S1, S2)***

[1] TRUE

A final note on the problem:

```
all.equal(S1, S2)
```

```
[1] TRUE
```

A final note on the problem:

```
system.time(S3 <- rowSums(array(unlist(matrices),  
dim=c(100, 100, 10000)), dims=2)  
)  
  
user  system elapsed  
0.60    0.11   0.70
```

```
all.equal(S1, S3)
```

```
[1] TRUE
```

which substitutes `rowSums()` for `apply()`, is considerably faster than the previous version but is still slower than the loop, is opaque, and wastes memory. The lesson: Avoid loops in R when doing so produces clearer and possibly more efficient code, not simply to avoid loops.

## 10.6 Optimization Problems\*

You may find that you want to estimate a statistical model for which there are no existing R functions. Fitting a statistical model often implies an *optimization* problem, such as maximizing a likelihood or minimizing a sum of squares. There are several general optimizers in the standard R distribution and many others in a variety of CRAN packages.<sup>22</sup> We'll illustrate with the standard R `optim()` function, which is quite powerful and general, and which implements several optimization methods. General-purpose optimizers work by searching iteratively for the maximum or minimum of an *objective function*, such as a likelihood or least-squares function.

[22](#) See the CRAN optimization taskview, `carWeb ("taskviews")`.

### 10.6.1 Zero-Inflated Poisson Regression

In Sections 6.5 and 6.10.4, we fit Poisson, quasi-Poisson, and negative-binomial regression models to Ornstein's data on interlocking directorships among Canadian corporations, which are in the data set `Ornstein` in the **carData** package. As was apparent in our discussion of Ornstein's data, more

corporations maintained zero interlocks with other corporations than would be predicted using any of these models. *Zero-inflated Poisson and negative-binomial* count-data regression models (ZIP and ZINB models) are used for count data with “extra zeros.” We’ll write a function to fit the ZIP model here. This example is somewhat artificial because although the standard R distribution doesn’t include a function capable of fitting the ZIP model, the `zeroinfl()` function in the **pscl** package (Zeileis, Kleiber, & Jackman, 2008) will fit this model. The example nevertheless illustrates how to fit a regression model by maximizing a likelihood function.

The ZIP model is defined as follows (e.g., Fox, 2016, Section 15.2.1): We imagine that there are two latent classes of cases, those for which the response variable  $y$  is *necessarily* zero and those for which the response variable conditional on the regressors, the  $x$ s, is Poisson distributed and thus *may* be zero or a positive integer. The probability  $\pi_i$  that a particular case  $i$  is in the first, necessarily zero, latent class may depend on a set of potentially distinct regressors,  $z$ s, according to a binary logistic-regression model:

$$\log \left( \frac{\pi_i}{1 - \pi_i} \right) = \gamma_0 + \gamma_1 z_{i1} + \cdots + \gamma_p z_{ip}$$

For an individual  $i$  in the second latent class,  $y$  follows a Poisson regression model with log link,

$$\log \mu_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_k x_{ik}$$

where  $\mu_i = E(y_i | x_1, \dots, x_k)$ , and conditional distribution

(10.2)

$$p(y_i | x_1, \dots, x_k) = \frac{\mu_i^{y_i} e^{-\mu_i}}{y_i!} \text{ for } y_i = 0, 1, 2, \dots$$

The regressors in the model for  $\pi^i$  may or may not be different from the regressors in the model for  $\mu_i$ .

The probability of observing a zero count for case  $i$ , not knowing to which latent class the case belongs, is therefore

$$p(0) = \Pr(y_i = 0) = \pi_i + (1 - \pi_i)e^{-\mu_i}$$

and the probability of observing a particular nonzero count  $y_i > 0$  is

$$p(y_i) = (1 - \pi_i) \frac{\mu_i^{y_i} e^{-\mu_i}}{y_i!}$$

The log-likelihood for the ZIP model combines the two components, for  $y = 0$  and for  $y > 0$ :

$$\begin{aligned}\log(\beta, \gamma) &= \sum_{y_i=0} \log [\pi_i + (1 - \pi_i)e^{-\mu_i}] \\ &\quad + \sum_{y_i>0} \log \left[ (1 - \pi_i) \frac{\mu_i^{y_i} e^{-\mu_i}}{y_i!} \right]\end{aligned}$$

where  $\beta = (\beta_0, \beta_1, \dots, \beta_k)'$  is the vector of parameters from the Poisson regression component of the model on which the  $\mu_i$  depend, and  $\gamma = (\gamma_0, \gamma_1, \dots, \gamma_p)'$  is the vector of parameters from the logistic-regression component of the model on which the  $\pi_i$  depend.

[Figure 10.4](#) displays a function, `zipmod()`, that uses `optim()` to compute the maximum-likelihood estimator for the ZIP model.<sup>23</sup> Because by default `optim()` minimizes the objective function, we work with the *negative* of the log-likelihood. The `zipmod()` function takes two required arguments: `X`, the model matrix for the Poisson part of the model, possibly without a column of ones for the constant regressor, which may then be attached to `X` in the function, depending on the value of the argument `intercept.X` (defaulting to `TRUE`), and the response vector `y` of observed counts. The optional argument `Z` gives the model matrix for the logistic-regression part of the model; if `Z` is absent, it's assumed to be the same as `X`. The argument `intercept.Z` controls whether a column of ones is attached to `Z` (and also defaults to `TRUE`). The ellipses argument `...` is passed through to `optim()` and allows the user to fine-tune the optimization—for example, to change the maximum number of iterations performed: See help ("optim") for details.

<sup>23</sup> You should have downloaded the file `zipmod.R`, which includes this function, and the file `zipmod-generic.R`, used later in the chapter, when you set up your *R Companion* project directory in [Section 1.1](#).

**Figure 10.4** Function `zipmod()` for fitting the ZIP model.

```

zipmod <- function(X, y, Z=X, intercept.X=TRUE,
                    intercept.Z=TRUE, ...) {
  # ZIP model
  # X: model matrix for Poisson model
  # y: response vector
  # Z: model matrix for logit model
  # intercept.X: add column of 1s for intercept to X
  # intercept.Z: add column of 1s for intercept to Z
  # ...: arguments to be passed to optim()
  if (!is.matrix(X) || !is.numeric(X))
    stop("X must be a numeric matrix")
  if (!is.matrix(Z) || !is.numeric(Z))
    stop("Z must be a numeric matrix")
  if (!is.vector(y) || !is.numeric(y) || !all(y >= 0)
      || !all(y == round(y)))
    stop("y must be a vector of counts")
  if (nrow(X) != length(y))
    stop("number of rows in X must be the same as length of y")
  if (nrow(Z) != length(y))
    stop("number of rows in Z must be the same as length of y")
  if (intercept.X) {
    X <- cbind(1, X)
    colnames(X)[1] <- "intercept"
  }
  if (intercept.Z) {
    Z <- cbind(1, Z)
    colnames(Z)[1] <- "intercept"
  }
  n.x <- ncol(X)
  negLogL <- function(beta.gamma) {
    beta <- beta.gamma[1:n.x]
    gamma <- beta.gamma[-(1:n.x)]
    pi <- 1/(1 + exp(- Z %*% gamma))
    mu <- exp(X %*% beta)
    L1 <- sum(log(pi + (1 - pi)*dpois(y, mu))[y == 0])
    L2 <- sum((log((1 - pi)*dpois(y, mu)))[y > 0])
    -(L1 + L2)
  }
  initial.beta <- coef(glm(y ~ X - 1, family=poisson))
  initial.gamma <- coef(glm(y == 0 ~ Z - 1, family=binomial))
  result <- optim(c(initial.beta, initial.gamma), negLogL,
                  hessian=TRUE, method="BFGS", ...)
  beta.gamma <- result$par
  vcov <- solve(result$hessian)
  par.names <- c(paste0("beta.", colnames(X)), paste0("gamma.",
  colnames(Z)))
  names(beta.gamma) <- par.names
  rownames(vcov) <- colnames(vcov) <- par.names
  list(coefficients=beta.gamma, vcov=vcov,
       deviance=2*result$value,
       converged=result$convergence == 0)
}

```

We're familiar with *local variables* defined within a function. Here, `negLogL()` is a *local function* defined within `zipmod()`. Variables like `X`, `Z`, and `y`, which live in the environment of `zipmod()`, are visible to `negLogL()` and thus need not be passed as arguments.<sup>24</sup> As its name suggests, `negLogL()` computes the negative of the log-likelihood, and it takes the current estimated parameter vector `beta.gamma`, which includes both the  $\beta$  and  $\gamma$  parameters, as an argument, as is required by `optim()`. The computation of the log-likelihood is largely straightforward, but rather than writing the formula for Poisson probabilities, as in Equation 10.2 (on page 510), we more simply use the `dpois()` function to compute them directly.

<sup>24</sup> One can, however, pass additional arguments to an objective function by specifying them as named arguments to `optim()`.

The first argument to `optim()` is a vector of start values for the parameters, to begin the iterative optimization process; we get the start values by fitting preliminary Poisson and logistic-regression models. The second argument is the objective function to be minimized, in our case, the local function `negLogL`. Specifying `hessian=TRUE` causes `optim()` to return a numerical approximation of the *Hessian* at the solution, the matrix of second partial derivatives of the objective function with respect to the parameters. The Hessian is useful for computing the estimated covariance matrix of the estimated parameters. The argument `method="BFGS"` selects a standard optimization method due originally to Broyden, Fletcher, Goldfarb and Shanno (see `help("optim")` for details). Finally, the ellipses ... are arbitrary optional arguments passed through from `zipmod()` to `optim()`.

The `optim()` function returns a list with several elements: `par` is the vector of coefficients at the solution; `value` is the value of the objective function—in our case, the negative log-likelihood—at the solution; `convergence` is a code reporting whether the computation converged to a solution, with zero indicating convergence; and `hessian` is the value of the Hessian at the solution.<sup>25</sup> Our `zipmod()` function gathers this information to return a list with the estimated coefficients, their covariance matrix, the deviance under the model (which is twice the negative log-likelihood), and a convergence indicator. We take the names of the parameters from the column names of the `X` and `Z` model matrices.

<sup>25</sup> There are additional elements in the returned object; see `help("optim")` for details.

In maximizing the likelihood, it helps to have an expression for the vector of partial derivatives (the *gradient*) of the log-likelihood with respect to the estimated parameters and possibly of the matrix of second derivatives of the log-

likelihood (the *Hessian*). The gradient is useful in numerical optimization because at a (relative) maximum or minimum, it's equal to zero, and its steepness in the directions of the various parameters when nonzero provides a clue about how to improve the solution. The Hessian provides information about curvature of the objective function, and, for maximum-likelihood estimates, the inverse of the Hessian computed at the estimates is the covariance matrix of the estimated parameters. The optim () function has an optional argument for the gradient—if this argument isn't specified, the gradient is approximated numerically—but it always uses a numerical approximation for the Hessian. Our zipmod () function doesn't specify an analytic gradient.<sup>26</sup>

<sup>26</sup> As a general matter, it's desirable in numerical optimization to use an analytic gradient but not necessarily an analytic Hessian. We leave it as a nontrivial exercise for the reader to derive the gradient of the ZIP model and to use it to improve zipmod ().

As in Sections 6.5 and 6.10.4, we'll regress interlocks (number of interlocks) in Ornstein's data on  $\log_2$  (assets) and the factors nation (nation of control, with four levels) and sector (sector of economic activity, with 10 levels). Although it's common in ZIP models to use the same predictors in the two parts of the model, a little experimentation suggests that we can't get useful estimates for the logistic-regression component of the ZIP model with sector in the model, and so we specify different **X** and **Z** model matrices:<sup>27</sup>

```
X <- model.matrix(~ log2(assets) + nation + sector,
  data=Ornstein) [, -1] # removing the constant column 1
Z <- model.matrix(~ log2(assets) + nation, data=Ornstein) [, -1]
head(Z) # X is similar, but with additional columns for sector
```

|   | log2(assets) | nationOTH | nationUK | nationUS |
|---|--------------|-----------|----------|----------|
| 1 | 17.172       | 0         | 0        | 0        |
| 2 | 17.021       | 0         | 0        | 0        |
| 3 | 16.789       | 0         | 0        | 0        |
| 4 | 16.382       | 0         | 0        | 0        |
| 5 | 16.204       | 0         | 0        | 0        |
| 6 | 15.314       | 0         | 0        | 0        |

<sup>27</sup> With sector in the logistic-regression component of the model, the covariance matrix of coefficients is numerically singular—the analog of (near-)perfect collinearity—reflecting the flatness of the likelihood surface at the maximum. As a consequence, we get very similar estimates of the  $\pi$ s whether or not sector is included in the logistic-regression part of the ZIP model.

Then we fit the model as

```
ornstein.zip <- zipmod(X, Ornstein$interlocks, Z)
ornstein.zip$coefficients
```

|                    |                   |                 |
|--------------------|-------------------|-----------------|
| beta.intercept     | beta.log2(assets) | beta.nationOTH  |
| -0.352181          | 0.272374          | -0.057190       |
| beta.nationUK      | beta.nationUS     | beta.sectorBNK  |
| -0.410996          | -0.689423         | 0.057060        |
| beta.sectorCON     | beta.sectorFIN    | beta.sectorHLD  |
| -0.597324          | -0.052950         | 0.048333        |
| beta.sectorMAN     | beta.sectorMER    | beta.sectorMIN  |
| 0.222193           | 0.015237          | 0.249536        |
| beta.sectorTRN     | beta.sectorWOD    | gamma.intercept |
| 0.168535           | 0.492357          | 3.025253        |
| gamma.log2(assets) | gamma.nationOTH   | gamma.nationUK  |
| -0.591274          | 1.003065          | -0.249375       |

```
gamma.nationUS  
1.312324
```

```
sqrt(diag(ornstein.zip$vcov)) # standard errors
```

|                    |                   |                 |
|--------------------|-------------------|-----------------|
| beta.intercept     | beta.log2(assets) | beta.nationOTH  |
| 0.138845           | 0.011961          | 0.075249        |
| beta.nationUK      | beta.nationUS     | beta.sectorBNK  |
| 0.089546           | 0.049329          | 0.097462        |
| beta.sectorCON     | beta.sectorFIN    | beta.sectorHLD  |
| 0.213224           | 0.075698          | 0.119499        |
| beta.sectorMAN     | beta.sectorMER    | beta.sectorMIN  |
| 0.076674           | 0.086998          | 0.068924        |
| beta.sectorTRN     | beta.sectorWOD    | gamma.intercept |
| 0.078598           | 0.075888          | 1.550477        |
| gamma.log2(assets) | gamma.nationOTH   | gamma.nationUK  |
| 0.160233           | 0.893853          | 1.138977        |
| gamma.nationUS     |                   |                 |
| 0.516149           |                   |                 |

```
ornstein.zip$converged
```

```
[1] TRUE
```

The expected value of the response under the ZIP model is

$$E(y|x_1, \dots, x_p, z_1, \dots, z_k) = (1 - \pi)\mu$$

and so we can use the estimated coefficients to compute the estimated expected response as a function of the predictors. For example, we proceed to construct an effect plot of estimated expected interlocks as a partial function of assets, fixing the dummy regressors for nation and sector to their proportions in the data set by averaging the corresponding dummy-regressor columns of the model matrices  $\mathbf{X}$  and  $\mathbf{Z}$ :

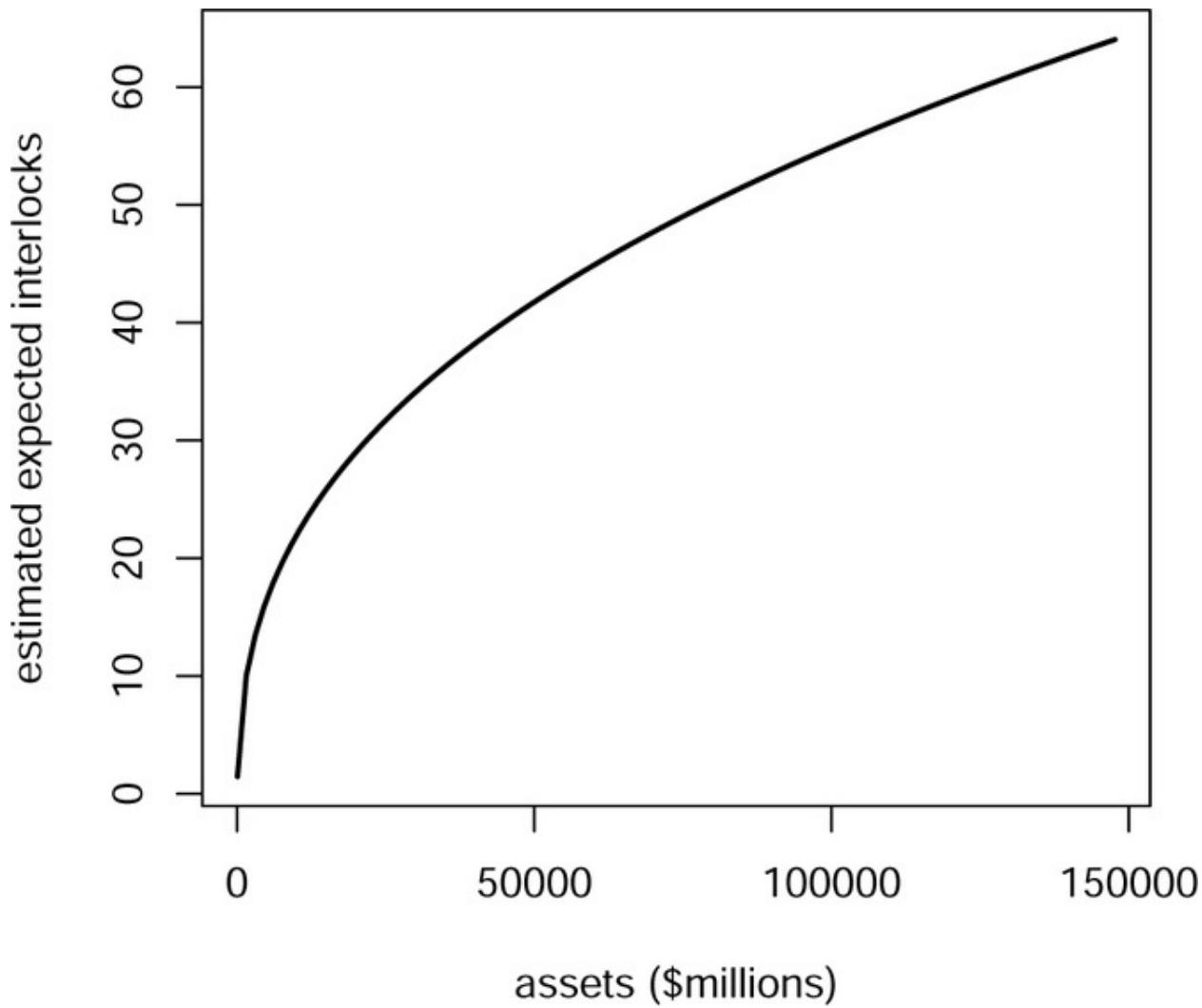
```

beta <- ornstein.zip$coefficients[1:14]
gamma <- ornstein.zip$coefficients[-(1:14)]
x.beta.fixed <- as.vector(c(1, colMeans(X[, 2:13])) %*%
                           beta[c(1, 3:14)])
z.gamma.fixed <- as.vector(c(1, colMeans(Z[, 2:4])) %*%
                           gamma[c(1, 3:5)])
assets <- with(Ornstein,
                 seq(min(assets), max(assets), length=100))
pi <- 1/(1 + exp(-(log2(assets)*gamma[2] + z.gamma.fixed)))
e.interlocks <- (1 - pi) *
  exp(log2(assets)*beta[2] + x.beta.fixed)
plot(assets, e.interlocks, xlab="assets ($millions)",
      ylab="estimated expected interlocks", type="l", lwd=2)

```

The effect plot is shown in [Figure 10.5](#).

**Figure 10.5** Effect plot for assets in the ZIP model fit to Ornstein's interlocking directorate data.



## 10.7 Monte-Carlo Simulations\*

Monte-Carlo, or random, simulations allow us to use brute-force computing to solve statistical problems for which theoretical results are either not available or are difficult to apply. For example, bootstrap confidence intervals and hypothesis tests are based on randomly resampling from the data.<sup>28</sup> Similarly, modern Bayesian methods use simulation (so-called *Markov Chain Monte Carlo*, or *MCMC*, methods) to sample from distributions too complex to deal with analytically.<sup>29</sup> Random simulations are also often used to explore the properties of statistical estimators.

<sup>28</sup> See [Section 5.1.3](#) and an online appendix to the *R Companion* for discussions of bootstrapping.

<sup>29</sup> See the online appendix to the *R Companion* on Bayesian inference for regression models.

Unless equipped with special hardware that taps into a truly random process

such as radioactive decay, computers can't generate random numbers. After all, if an algorithm is provided with the same input, it always produces the same output—the result, therefore, is *deterministic*, not random. There are, however, algorithms that generate sequences of numbers that behave in various respects *as if* they were random, so-called *pseudo-random numbers*. Although they produce pseudo-random rather than truly random numbers, it's typical to refer to these algorithms in abbreviated form as random-number generators.

The algorithms that R uses to generate pseudo-random numbers are generally of very high quality. Unless you are interested in the theory of pseudo-random numbers, you can simply use R's built-in functions for simulations without worrying about how they work. The complements to this chapter provide some references. R's programmability combined with its facility for generating random numbers makes it a natural tool for developing statistical simulations.

### 10.7.1 Testing Regression Models Using Simulation

The Salaries data set in the **carData** package provides the 2008–2009 nine-month academic salaries for assistant professors, associate professors, and professors in a U.S. college:

```
head(Salaries)
```

|   | rank      | discipline | yrs.since.phd | yrs.service | sex | salary      |
|---|-----------|------------|---------------|-------------|-----|-------------|
| 1 | Prof      | B          |               | 19          | 18  | Male 139750 |
| 2 | Prof      | B          |               | 20          | 16  | Male 173200 |
| 3 | AsstProf  | B          |               | 4           | 3   | Male 79750  |
| 4 | Prof      | B          |               | 45          | 39  | Male 115000 |
| 5 | Prof      | B          |               | 40          | 41  | Male 141500 |
| 6 | AssocProf | B          |               | 6           | 6   | Male 97000  |

```
nrow(Salaries)
```

```
[1] 397
```

Along with salary, we have data on each faculty member's academic rank, yrs.since.phd, and sex. In addition, each individual's academic department is classified as either "A" or "B", roughly corresponding respectively to "theoretical" disciplines and "applied" disciplines. The data were collected in this form as part of the ongoing effort of the college's administration to monitor salary differences between female and male faculty members.<sup>30</sup>

<sup>30</sup> When we originally saved the Salaries data, we ordered the levels of the factor rank in their natural ordering, "AsstProf", "AssocProf", "Prof", rather than in the default alphabetical ordering. Had we not done this, we could now reorder

the levels of rank by the command `Salaries$rank <- factor (Salaries$rank, levels=c ("AsstProf", "AssocProf", "Prof"))`.

Before looking at salaries, we examine the numbers of male and female faculty in the college by discipline and rank:

```
ftable(x1 <- xtabs(~ discipline + rank + sex, data=Salaries))
```

|   |           | sex Female Male |      |  |
|---|-----------|-----------------|------|--|
|   |           | discipline      | rank |  |
| A | AsstProf  | 6               | 18   |  |
|   | AssocProf | 4               | 22   |  |
|   | Prof      | 8               | 123  |  |
| B | AsstProf  | 5               | 38   |  |
|   | AssocProf | 6               | 32   |  |
|   | Prof      | 10              | 125  |  |

Here, we use the `xtabs()` function to construct a three-dimensional cross-classification by discipline, rank, and sex and then employ the `ftable()` function to “flatten” the three-dimensional array to a table for printing. It is clear that the faculty is mostly male, but it is harder to see if the fraction of females varies by rank and discipline. We therefore transform the counts to percentages of males and females in combinations of rank and discipline:

```
round(100*ftable(prop.table(x1, margin=c(1, 2))), 1)
```

|   |           | sex Female Male |      |  |
|---|-----------|-----------------|------|--|
|   |           | discipline      | rank |  |
| A | AsstProf  | 25.0            | 75.0 |  |
|   | AssocProf | 15.4            | 84.6 |  |
|   | Prof      | 6.1             | 93.9 |  |
| B | AsstProf  | 11.6            | 88.4 |  |
|   | AssocProf | 15.8            | 84.2 |  |
|   | Prof      | 7.4             | 92.6 |  |

### **# % Male and Female**

The `prop.table()` function computes proportions, conditioning on margins 1 (discipline) and 2 (rank) of the three-way table; we multiply the proportions by 100 to obtain percentages, rounding to one digit to the right of the decimal point. Assistant professors in discipline "A" are 25% female. In all other combinations

of rank and discipline, females comprise 6% to 16% of the faculty. Treating salary as the response, the data set has three factors and one numeric predictor. It is possible to view all of the data in a single very helpful graph. We use the xyplot () function in the **lattice** package to produce the graph in [Figure 10.6](#):<sup>31</sup>

```
library("lattice")
xyplot(salary ~ yrs.since.phd | discipline:rank, groups=sex,
       data=Salaries, type=c("g", "p", "r"),
       key=simpleKey(text=c("Female", "Male"),
                     points=TRUE, lines=TRUE))
```

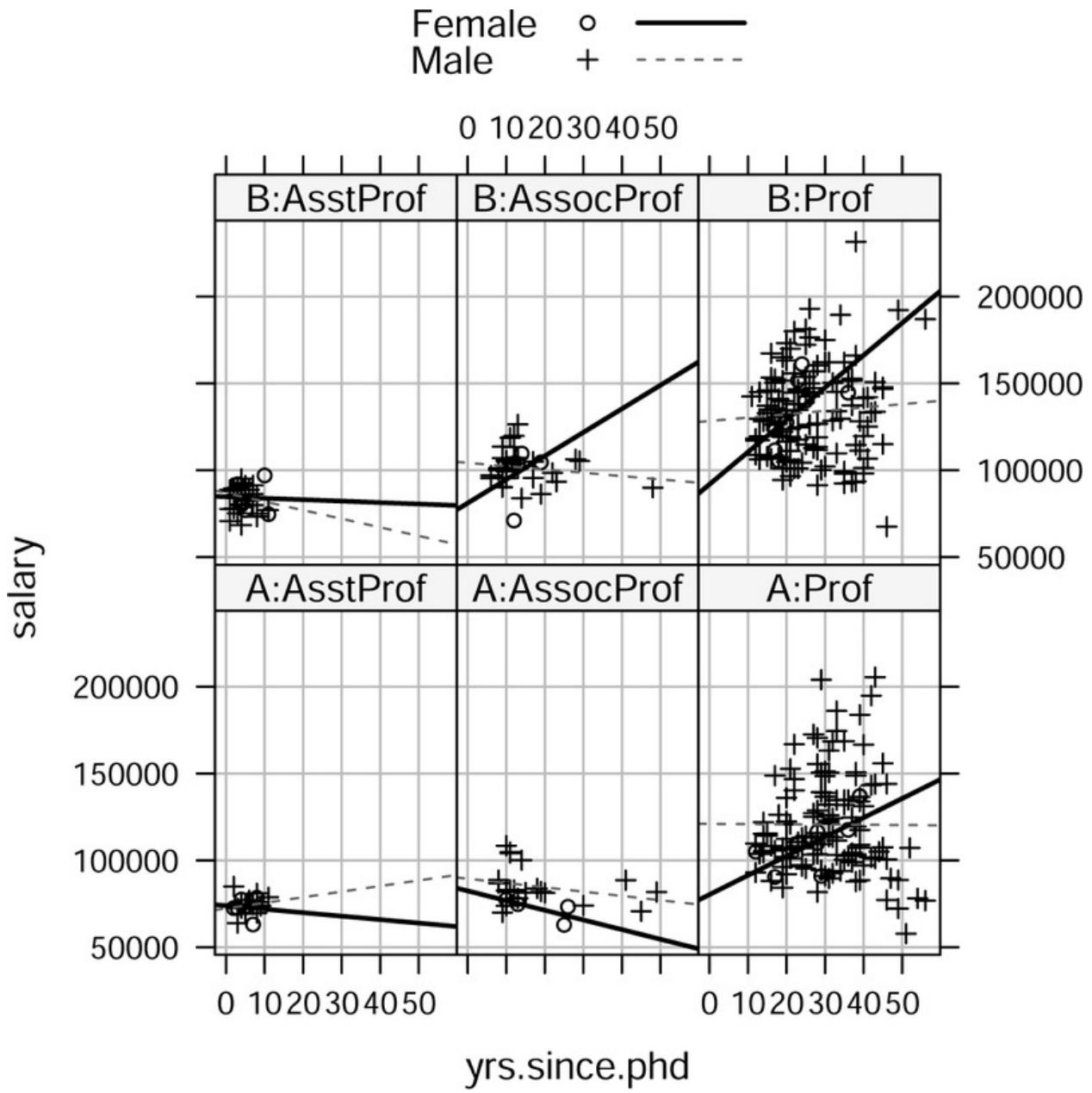
[31](#) The **lattice** package is discussed more generally in [Section 9.3.1](#).

A formula determines the horizontal and vertical axes of the graph. The variables to the right of the | (vertical bar) define the different panels of the plot, and so we have six panels, one for each combination of discipline and rank. The groups argument defines a grouping variable—sex—within each panel. The type argument specifies printing a grid ("g"), showing the individual points ("p"), and displaying a least-squares regression line ("r"). Points in each group (i.e., sex) are displayed with a distinct plotting symbol, and the regression line within each group is drawn with a distinct line type. The key argument invokes the **lattice** function simpleKey () to generate the legend at the top of the graph.

Surprisingly, after controlling for discipline and rank, salary appears to be more or less independent of yrs.since.phd. In some cases, the fitted lines even have *negative* slopes, implying that the larger the value of yrs.since.phd, the *lower* the salary. There are clear rank effects, however, with professors earning more on average than others, and professors' salaries are also much more variable.

Differences by discipline are harder to discern because to examine these differences, we have to compare panels in different rows, a more difficult visual task than comparing panels in the same row.

**Figure 10.6** Salary by years since PhD for combinations of rank and discipline for the college salary data.



Because the numeric variable `years.since.phd` doesn't seem to matter much, we can usefully summarize the data with parallel boxplots, as in [Figure 10.7](#), using the `bwplot()` (box-and-whisker plot) function in the **lattice** package:

```
bwplot (salary ~ discipline:sex | rank, data=Salaries, scales=list (rot=90),
layout=c (3, 1))
```

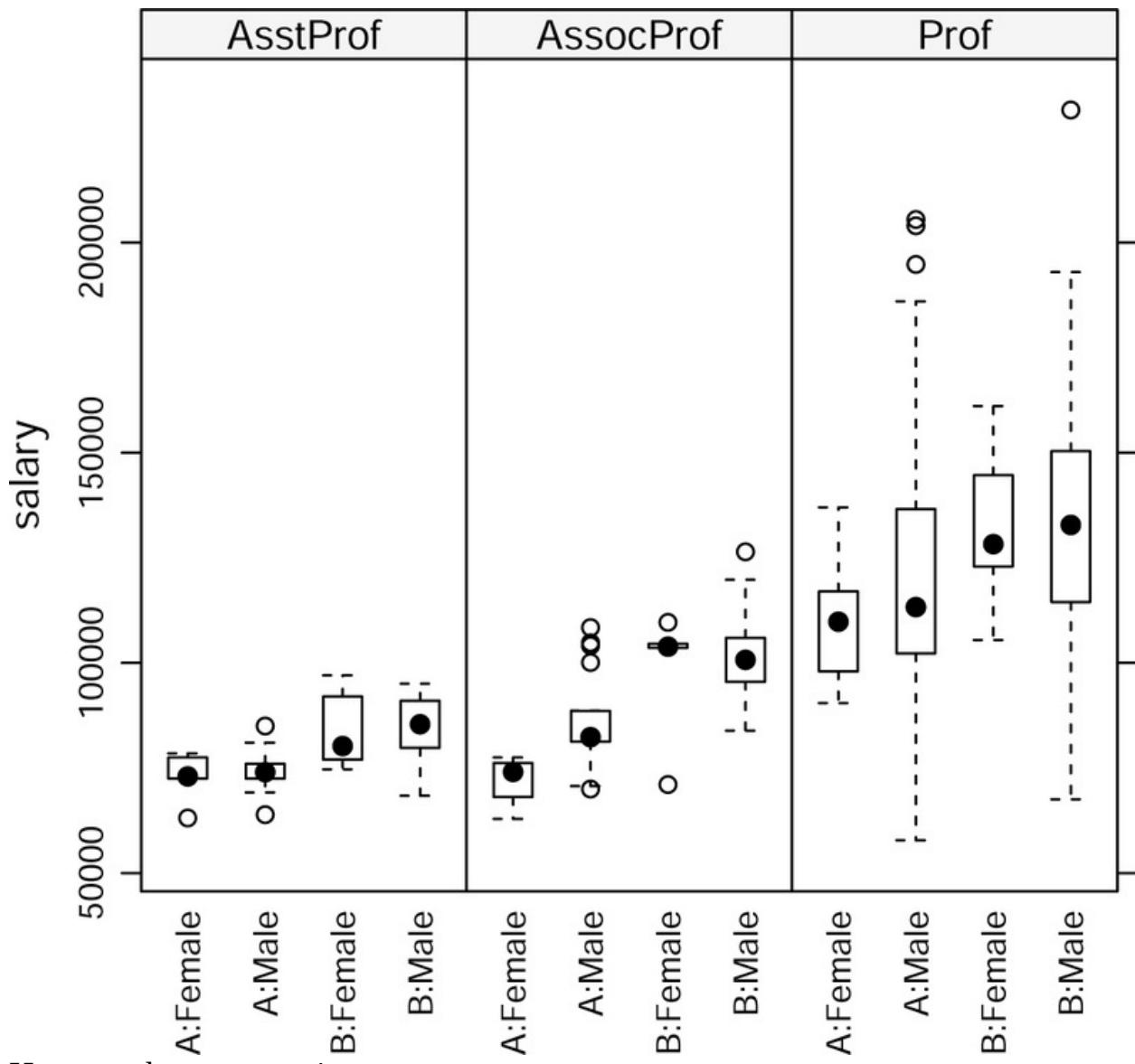
The specification `rot=90` in the `scales` argument rotates the tick-mark labels for the horizontal and vertical axes by 90°. The panels in the graph are organized so that the comparisons of interest, between males and females in the same discipline and rank, are closest to each other. Relatively small differences between males and females are apparent in the graphs, with males generally a bit

higher in salary than comparable females. Discipline effects are clearer in this graph, as is the relatively large variation in male professors' salaries for both disciplines.

We now turn to the problem of assessing the difference between males' and females' salaries. In the area of wage discrimination, it is traditional to assess differences using the following paradigm:

1. Fit a model, usually the linear regression model, to the majority group, in this case the male faculty members. This model allows us to predict salary for male faculty members given their rank, discipline, and yrs.since.phd.
2. Predict salaries for the minority group, in this case the female faculty members, based on the model fit to the majority group. If no real distinction exists between the two groups, then predictions based on the majority group should be unbiased estimates of the actual salaries of the minority group.
3. Compute the average difference between the actual and predicted minority salaries.

**Figure 10.7** Boxplots of salary by rank, discipline, and sex for the Salaries data.



Here are the computations:

```
fselector <- Salaries$sex == "Female" # TRUE for females
salmod <- lm(salary ~ rank*discipline + yrs.since.phd,
             data=Salaries, subset=!fselector) # regression for males
S(salmod)
```

```
Call: lm(formula = salary ~ rank * discipline + yrs.since.phd,
        data = Salaries, subset = !fselector)
```

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t ) |
|-------------|----------|------------|---------|----------|
| (Intercept) | 74095.1  | 5593.7     | 13.25   | < 2e-16  |

|                           |         |        |      |         |
|---------------------------|---------|--------|------|---------|
| rankAssocProf             | 10433.3 | 7632.2 | 1.37 | 0.17    |
| rankProf                  | 45621.4 | 6807.5 | 6.70 | 8.2e-11 |
| disciplineB               | 10418.3 | 6722.9 | 1.55 | 0.12    |
| yrs.since.phd             | 29.4    | 135.0  | 0.22 | 0.83    |
| rankAssocProf:disciplineB | 6267.4  | 9358.7 | 0.67 | 0.50    |
| rankProf:disciplineB      | 2604.0  | 7362.3 | 0.35 | 0.72    |

Residual standard deviation: 23500 on 351 degrees of freedom

Multiple R-squared: 0.415

F-statistic: 41.4 on 6 and 351 DF, p-value: <2e-16

| AIC    | BIC    |
|--------|--------|
| 8230.9 | 8261.9 |

**# predictions for females:**

```
femalePreds <- predict(salmod, newdata=Salaries[fselector, ])
(meanDiff <- mean(Salaries$salary[fselector] - femalePreds))
```

[1] -4550.5

We first define a *selector variable*, fselector, which has the value TRUE for females and FALSE for males. We then fit a regression to the males only, allowing for interactions between Discipline and Rank and an additive YrsSincePhD effect, which we expect will be small in light of the earlier graphical analysis. We use the generic function predict () to get predictions from the fitted regression. The newdata argument tells the function to obtain predictions only for the females. If we had omitted the second argument, then predictions would have been returned for the data used in fitting the model, in this case the males only—not what we want. We finally compute the mean difference between observed and predicted salaries for the females. The average observed female salary is therefore \$4,551 less than the amount predicted from the fitted model for males.

A question of interest is whether or not this difference of -\$4,551 is due to chance alone. Many of the usual ideas of significance testing are not obviously applicable here: First, the data form a complete population, not a sample, and so we are not inferring from a sample to a larger population. Nevertheless, because we are interested in drawing conclusions about the *social process* that generated the data, statistical inference is at least arguably sensible. Second, a sex effect can be added to the model in many ways and these could all lead to different conclusions. Third, the assumptions of the linear regression model are unlikely to hold; for example, we noted that professors are more variable than others in salary.

Here is a way to judge “statistical significance” based on simulation: We will compare the mean difference for the real minority group, the 39 females, to the mean difference we would have obtained had we nominated 39 of the faculty selected at random to be the “minority group.” We compute the mean difference between the actual and predicted salaries for these 39 “random females” and then repeat the process a large number of times—say 999 times:

```
(seed <- sample(1e7, 1)) # generate and note seed  
8141976  
  
set.seed(seed) # for reproducibility  
fnumber <- sum(fselector) # number of females  
n <- length(fselector) # number of observations  
B <- 999 # number of replications  
simDiff <- numeric(B) # initialize vector with B entries  
for (j in 1:B) {  
  # random sample of nominated 'females':  
  sel <- sample(n, fnumber)  
  # refit regression model to simulated 'males':  
  m2 <- update(salmod, subset=-sel)  
  # simulated salary difference in means:  
  simDiff[j] <- mean(Salaries$salary[sel]  
    - predict(m2, newdata=Salaries[sel, ]))  
}  
}
```

Explicitly setting the seed for R’s pseudo-random-number generator makes our simulation reproducible, which is generally a sound practice. We generated the seed, 8141976, pseudo-randomly with the command (seed <- sample (1e7, 1)). The important point is that we know the seed that was used.

We first compute the number of females, fnumber, and the number of faculty members, n. We use a for loop ([Section 10.4](#)) to perform the calculation repeatedly, B = 999 times. Within the loop, the sample () function is used to assign fnumber randomly selected faculty members to be simulated “females,” the regression is updated without the randomly chosen “females,” the predictions are obtained for the simulated “females” based on this fitted model, and the average difference between actual and predicted salary is saved.

There are many ways to summarize the simulation; we use a histogram (in

[Figure 10.8](#)):

```
(frac <- round(sum(meanDiff > simDiff)/(1 + B), 3))  
[1] 0.102  
  
hist(simDiff,  
      main=paste(  
        "Histogram of Simulated Mean Differences\np-value =",  
        frac),  
      xlab="Dollars")
```

The variable `frac` is the fraction of replications in which the simulated difference is less than the observed difference and gives the estimated *p*-value for the null hypothesis of no systematic difference between female and male average salaries against a one-sided alternative. The escaped character "\n" in the histogram title, specified by the `main` argument, is the new-line character. We draw the histogram with a broken vertical line at the observed mean difference.<sup>32</sup>

<sup>32</sup> See [Chapter 9](#) for more on customizing graphs.

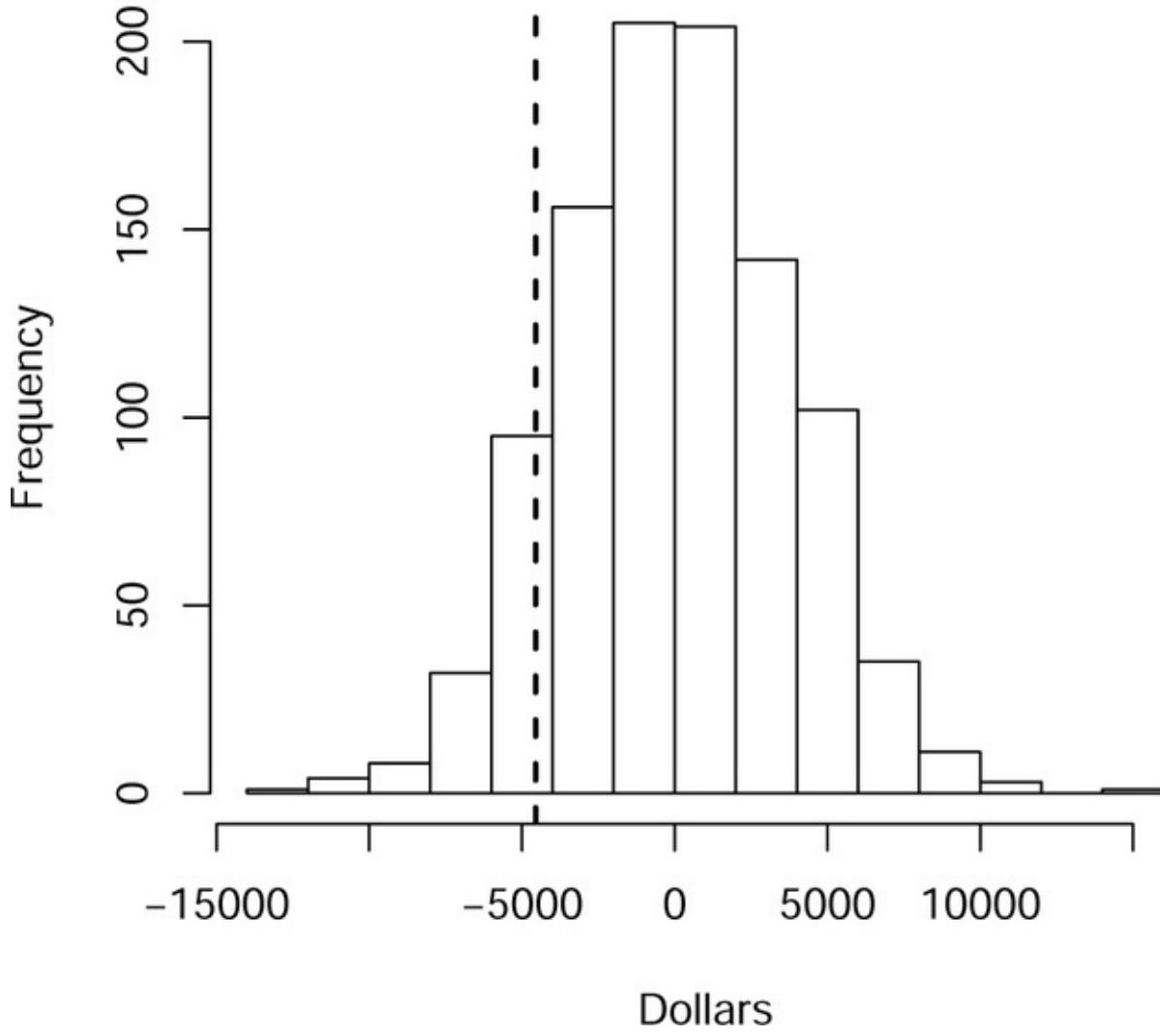
The histogram of simulated mean differences looks more or less symmetric and is centered close to zero. The simulated *p*-value is .102, suggesting only weak evidence against the hypothesis that salaries are determined by a process unrelated to sex.

This procedure can be criticized on several grounds, not the least of which is the manner in which “females” were randomly created. Most male faculty are professors, while the female faculty are more evenly spread among the ranks. Thus, the randomly generated “females” tend to be professors, which does not reflect the true state of affairs in the college. The reader is invited to repeat this analysis, changing the random mechanism used to generate “females” to ensure that each set of randomly generated “females” has the same number in each rank as do the actual females. Of course, the process of promotion to professor could itself be subject to gender discrimination. In addition, salaries are typically generally compared in log-scale (see [Section 3.4.1](#)), which can help to correct for the excess variation in the professor rank. In log-scale, the predictions from the model fit to males could be exponentiated and then compared to salaries for female faculty, both in the actual data and in the simulation. This, too, is left as an exercise for the interested reader.

**Figure 10.8** Histogram of simulated mean sex differences for the college salary data. The broken vertical line is drawn at the observed mean difference of -4,551.

## Histogram of Simulated Mean Differences

p-value = 0.102



### 10.8 Debugging R Code\*

Computer programs are written by people, and people tend not to be perfect.<sup>33</sup> It's unusual to write an R program of any complexity that works correctly the first time, and so learning how to detect and correct bugs efficiently is an important part of learning to program competently in R.

<sup>33</sup> Lest you think us cynical, we extrapolate here from our own experience. R offers several effective facilities for debugging functions using the `browser()` command. The `browser()` command halts the execution of an R function and allows the programmer to examine the values of local variables, to execute expressions in the environment of the function, and to step through the function

line by line from the point at which it was paused. The `browser()` function can be invoked directly, by inserting calls to `browser()` in the function code, or indirectly via the `debug()` and `debugonce()` commands.<sup>34</sup> These commands interact productively with RStudio, which also allows you to set break-points in a function.<sup>35</sup> In this section, we'll illustrate the use of the `browser()` command. <sup>34</sup> The `debug()` and `debugonce()` functions enter browser mode when the target function is called. For example, try the command `debugonce(zipmodBugged)` for the example developed in this section. Once in browser mode, you can step through the target function line by line, examining its state as you go. We generally find it more efficient to set a *break-point* before the known location of an error, however. The distinction between `debug()` and `debugonce()` is clear in their names: The command `debug(function-name)` will pause the function *function-name()* each time it's executed, until the function is redefined or the command `undebug(function-name)` is entered. In contrast, `debugonce(function-name)` will pause the function only the *first* time it's called.

<sup>35</sup> See the RStudio *Debug* menu, in particular the menu item for *Debugging Help*, and the discussion later in this section.

To provide an example, we introduce a bugged version of our ZIP model program, `zipmodBugged()`, in [Figure 10.9](#) and in the file `zipmodBugged.R`. This is a real example, in that the bug we'll encounter was present in our initial attempt to write the program. When we try to use the program, it produces an error:

```
ornstein.bugged <- zipmodBugged(X, Ornstein$interlocks, Z)

Error in optim(c(initial.beta, initial.gamma),
               negLogL, hessian = TRUE, :
  objective function in optim evaluates to length 220 not 1
In addition: Warning message:
In L1 + L2 :
  longer object length is not a multiple of shorter object length

traceback()

2: optim(c(initial.beta, initial.gamma), negLogL, hessian = TRUE,
         method = "BFGS", ...) at zipmodBugged.R#23
1: zipmodBugged(X, Ornstein$interlocks, Z)
```

The traceback, produced by calling the `traceback()` function after the error,<sup>36</sup> shows the *stack* of function calls to the point of the error and is often—though not here—helpful in localizing a bug. The error message, however, points clearly to a problem in `optim()`, called in line 23 of the program (actually lines 23 and

24) and produced by the objective function, negLogL (): The objective function should return a single value (the negative of the log-likelihood at the current parameter estimates) but apparently it returns a vector of 220 values! We've now located the source of the problem and can turn our attention to fixing it.

[36](#) Working in RStudio, we could equivalently press the *Show Traceback* link that appears in the *Console* tab after an error.

**Figure 10.9** A bugged version of the zipmod () ZIP model function (cf. zipmod () in [Figure 10.4](#), page 511).

```

zipmodBugged <- function(X, y, Z=X, intercept.X=TRUE,
                           intercept.Z=TRUE, ...) { # bugged!
  if (intercept.X) {
    X <- cbind(1, X)
    colnames(X)[1] <- "intercept"
  }
  if (intercept.Z) {
    Z <- cbind(1, Z)
    colnames(Z)[1] <- "intercept"
  }
  n.x <- ncol(X)
  negLogL <- function(beta.gamma) {
    beta <- beta.gamma[1:n.x]
    gamma <- beta.gamma[-(1:n.x)]
    pi <- 1/(1 + exp(- Z %*% gamma))
    mu <- exp(X %*% beta)
    L1 <- sum(log(pi + (1 - pi)*dpois(y, mu)))[y == 0]
    L2 <- sum((log((1 - pi)*dpois(y, mu))))[y > 0]
    -(L1 + L2)
  }
  initial.beta <- coef(glm(y ~ X - 1, family=poisson))
  initial.gamma <- coef(glm(y == 0 ~ Z - 1, family=binomial))
  result <- optim(c(initial.beta, initial.gamma), negLogL,
                  hessian=TRUE, method="BFGS", ...)
  beta.gamma <- result$par
  vcov <- solve(result$hessian)
  par.names <- c(paste0("beta.", colnames(X)), paste0("gamma.",
    colnames(Z)))
  names(beta.gamma) <- par.names
  rownames(vcov) <- colnames(vcov) <- par.names
  list(coefficients=beta.gamma, vcov=vcov,
       deviance=2*result$value,
       converged=result$convergence == 0)
}

```

To figure out what went wrong, we insert a call to `browser()` immediately after the two components of the log-likelihood are computed but before the negative of the log-likelihood is returned, “outdenting” the `browser()` command to make it more visible in the code (see [Figure 10.10](#)). Then, when we execute `zipmodBugged()`, it stops at the call to `browser()` and the `>` command prompt in the *Console* changes to `Browse[1]>`, allowing us to execute commands to examine the local state of `negLogL()` prior to the occurrence of the error:

```
> ornstein.bugged <- zipmodBugged(X, Ornstein$interlocks, Z)
```

```
Called from: fn(par, ...)
```

```
Browse[1]> str(L1)
```

```
num [1:28] NA ...
```

```
Browse[1]> str(L2)
```

**Figure 10.10** zipmodBugged () with browser () command inserted before the error.

```

zipmodBugged <- function(X, y, Z=X, intercept.X=TRUE,
                           intercept.Z=TRUE, ...) { # bugged!
  if (intercept.X) {
    X <- cbind(1, X)
    colnames(X)[1] <- "intercept"
  }
  if (intercept.Z) {
    Z <- cbind(1, Z)
    colnames(Z)[1] <- "intercept"
  }
  n.x <- ncol(X)
  negLogL <- function(beta.gamma) {
    beta <- beta.gamma[1:n.x]
    gamma <- beta.gamma[-(1:n.x)]
    pi <- 1/(1 + exp(- Z %*% gamma))
    mu <- exp(X %*% beta)
    L1 <- sum(log(pi + (1 - pi)*dpois(y, mu)))[y == 0]
    L2 <- sum((log((1 - pi)*dpois(y, mu))))[y > 0]
    browser() # interrupt execution here!
    -(L1 + L2)
  }
  initial.beta <- coef(glm(y ~ X - 1, family=poisson))
  initial.gamma <- coef(glm(y == 0 ~ Z - 1, family=binomial))
  result <- optim(c(initial.beta, initial.gamma), negLogL,
                  hessian=TRUE, method="BFGS", ...)
  beta.gamma <- result$par
  vcov <- solve(result$hessian)
  par.names <- c(paste0("beta.", colnames(X)), paste0("gamma.",
    colnames(Z)))
  names(beta.gamma) <- par.names
  rownames(vcov) <- colnames(vcov) <- par.names
  list(coefficients=beta.gamma, vcov=vcov,
       deviance=2*result$value, converged=result$convergence == 0)
}

```

num [1:220] -1255 NA NA NA NA ...

**Browse[1]> str(L1 + L2)**

num [1:220] NA ...

Warning message:

In L1 + L2 :

longer object length is not a multiple of shorter object length  
 Clearly something is very wrong with the computation of the two components of

the log-likelihood, L1 and L2: Both should be single numbers, but they are vectors of different lengths, which accounts for both the error and the warning message. Looking carefully at the expressions for L1 and L2 in the code, we see that the indexing operations,  $[y == 0]$  and  $[y > 0]$ , are in the wrong place—we should index *before* we sum. We can test out this solution by entering corrected commands at the browser prompt:

```
Browse[1]> str(L1 <- sum(log(pi + (1 - pi)*dpois(y, mu)) [y == 0]))  
num -46.2  
  
Browse[1]> str(L2 <- sum((log((1 - pi)*dpois(y, mu))) [y > 0]))  
num -1084  
  
Browse[1]> -(L1 + L2)  
[1] 1129.944
```

**Browse[1]> Q**

Now L1 and L2 are single numbers, as they should be, and the negative of their sum is also a single number: We've apparently isolated and learned how to correct the source of the error.

The Q command to the browser exits from browser mode.<sup>37</sup> Fixing zipmodBugged() by putting the indexing expressions in the correct places returns us to the original (correct) definition of the zipmod() function (given in [Figure 10.4](#) on page 511).

[37](#) For more information on the browser, including a list of available commands, see help ("browser"). This help page is partially shown in the lower-right pane of [Figure 10.11](#).

The browser() command interacts helpfully with RStudio (see [Figure 10.11](#)):

- While you're in browser mode, RStudio displays the interrupted function in the editor pane, showing where the function is paused—see the arrow to the left of the highlighted line in the upper-left pane of [Figure 10.11](#). This feature works best when the source code for the function is saved in a file (zipmodBugged.R in the example) that's “sourced” into the R *Console* by pressing the *Source* button at the top of the tab displaying the file.
- Debugging controls appear at the top of the *Console* (at the lower-left of [Figure 10.11](#)), allowing you to step through the paused function line by line, continue its execution, stop the function, and so on—hover the mouse over each button to see an explanation of what it does.
- You can type commands in the *Console* that are executed in the environment of the paused function, as illustrated in the *Console* at the

lower-left of [Figure 10.11](#).

- The *Environment* tab (at the upper-right of [Figure 10.11](#)) displays local variables in the paused function—in our example, the variables L1 and L2 (but not the variables X, Z, and y, which are in the environment of `zipmodBugged()`, not directly in the environment of `negLogL()`). The *Environment* tab also shows a traceback to the point where execution is paused.
- As an alternative to inserting a `browser()` command into a function to suspend execution, you can set a break-point in RStudio by clicking to the left of a line number in the source tab for the function. A red dot appears to the left of the line number, indicating that the break-point is set at that line. Left-clicking again unsets the break-point.

**Figure 10.11** Debugging `zipmodBugged()` in RStudio.

```

zipmod <- function(X, ...){
  UseMethod("zipmod")
}

zipmod.default <- function(X, y, Z=X, intercept.X=TRUE,
  intercept.Z=TRUE, ...) {
  if (!is.matrix(X) || !is.numeric(X))
    stop("X must be a numeric matrix")
  if (!is.matrix(Z) || !is.numeric(Z))
    stop("Z must be a numeric matrix")
  if (!is.vector(y) || !is.numeric(y) || !all(y >= 0)
    || !all(y == round(y)))
    stop("y must be a vector of counts")
  if (nrow(X) != length(y))
    stop("number of rows in X must be the same as length of y")
  if (nrow(Z) != length(y))
    stop("number of rows in Z must be the same as length of y")
  if (intercept.X) {
    X <- cbind(1, X)
    colnames(X)[1] <- "intercept"
  }
  if (intercept.Z) {
    Z <- cbind(1, Z)
    colnames(Z)[1] <- "intercept"
  }
  n.x <- ncol(X)
  negLogL <- function(beta.gamma) {
    beta <- beta.gamma[1:n.x]
    gamma <- beta.gamma[-(1:n.x)]
    pi <- 1/(1 + exp(- Z %*% gamma))
    mu <- exp(X %*% beta)
    L1 <- sum(log(pi + (1 - pi)*dpois(y, mu))[y == 0])
    L2 <- sum((log((1 - pi)*dpois(y, mu)))[y > 0])
    -(L1 + L2)
  }
  initial.beta <- coef(glm(y ~ X - 1, family=poisson))
  initial.gamma <- coef(glm(y == 0 ~ Z - 1, family=binomial))
  result <- optim(c(initial.beta, initial.gamma), negLogL,
    hessian=TRUE, method="BFGS", ...)
  beta.gamma <- result$par
  vcov <- solve(result$hessian)
  par.names <- c(paste0("beta.", colnames(X)),
    paste0("gamma.", colnames(Z)))
  names(beta.gamma) <- par.names
  rownames(vcov) <- colnames(vcov) <- par.names
  result <- list(coefficients=beta.gamma, vcov=vcov,
    npar=c(beta=ncol(X), gamma=ncol(Z)),
    deviance=2*result$value,
    converged=result$convergence == 0)
  class(result) <- "zipmod"
  result
}

```

## 10.9 Object-Oriented Programming in R\*

Quick-and-dirty programming, which is the principal focus of this chapter, generally does not require writing object-oriented functions, but understanding how object-oriented programming works in R is often useful even to quick-and-dirty R programmers. We described the basics of the S3 object system in [Section 1.7](#), which you may wish to reread now. In the current section, we explain how to write S3 generic functions and methods.

[Figure 10.12](#) shows an S3 object-oriented version of our ZIP model program, a modification of the `zipmod()` function (in [Figure 10.4](#) on page 511).<sup>38</sup>

Anticipating an extension of this example, we write `zipmod()` as an S3 *generic function*, for which `zipmod.default()` is the *default method*.

[38](#) The functions discussed in this section are in the file `zipmod-generic.R`. Almost all S3 generic functions have the same general form; here, for example, are the `summary()` and `print()` generics:

### ***summary***

```
function (object, ...)
UseMethod("summary")
<bytecode: 0x000000001d03ba78>
<environment: namespace:base>
```

### ***print***

```
function (x, ...)
UseMethod("print")
<bytecode: 0x0000000014a266f0>
<environment: namespace:base>
```

By convention, method functions have the same arguments as the corresponding generic function. The generic `summary()` function has arguments `object` and ..., `print()` has arguments `x` and ..., and our generic `zipmod()` function in [Figure 10.12](#) has arguments `X` and .... The ... (ellipses) argument allows additional arguments that are not in the generic function to be defined for specific methods. For example, `zipmod.default()` in [Figure 10.12](#) has four additional arguments. Three of the four arguments have default values and are therefore not required.

The argument `y` has no default, and so the user must supply this argument. The `...` argument to `zipmod.default()` collects any additional arguments that the user provides. By examining the definition of the `zipmod.default()` function in [Figure 10.12](#), you can see that these arguments are passed to the `optim()` function, potentially to modify the optimization process.

The function `zipmod.default()` returns a list with coefficients, the covariance matrix of the coefficients, the numbers of parameters in the Poisson and logit parts of the model, the deviance under the fitted model, and a convergence indicator. The returned list is assigned the class "zipmod".

We apply `zipmod()` to Ornstein's data, defining `X` and `Z` model matrices as before (see page 513) for the Poisson and logit parts of the model:

```
X <- model.matrix(~ log2(assets) + nation + sector,
  data=Ornstein) [, -1] # removing the constant column 1
Z <- model.matrix(~ log2(assets) + nation, data=Ornstein) [, -1]
ornstein.zip.2 <- zipmod(X, Ornstein$interlocks, Z)
class(ornstein.zip.2)

[1] "zipmod"

str(ornstein.zip.2, strict.width="cut")
List of 5
 $ coefficients: Named num [1:19] -0.3522 0.2724 -0.0572 -0.41...
 ..- attr(*, "names")= chr [1:19] "beta.intercept" "beta.log"...
 $ vcov       : num [1:19, 1:19] 0.019278 -0.001539 -0.001321...
 ..- attr(*, "dimnames")=List of 2
 ...$ : chr [1:19] "beta.intercept" "beta.log2(assets)" "b"...
 ...$ : chr [1:19] "beta.intercept" "beta.log2(assets)" "b"...
 $ npar       : Named int [1:2] 14 5
 ..- attr(*, "names")= chr [1:2] "beta" "gamma"
 $ deviance   : num 2227
 $ converged  : logi TRUE
 - attr(*, "class")= chr "zipmod"

# to avoid printing the entire object
```

**Figure 10.12** An object-oriented version of our ZIP model program, with `zipmod()` as an S3 generic function and `zipmod.default()` as its default method.

```

vcov.zipmod <- function(object, separate=FALSE, ...) {
  if (!separate) return(object$vcov)
  else{
    vcov <- object$vcov
    npar <- object$npar
    index <- 1:npar["beta"]
    vcov.beta <- vcov[index, index]
    vcov.gamma <- vcov[-index, -index]
    names.beta <- rownames(beta)
    names.gamma <- rownames(gamma)
    rownames(vcov.beta) <- colnames(vcov.beta) <-
      sub("^beta\\\\.\\.", "", names.beta)
    rownames(vcov.gamma) <- colnames(vcov.gamma) <-
      sub("^gamma\\\\.\\.", "", names.gamma)
    return(list(beta=vcov.beta, gamma=vcov.gamma))
  }
}

print.zipmod <- function(x, ...) {
  coef <- coef(x)
  npar <- x$npar
  beta <- coef[1:npar["beta"]]
  gamma <- coef[-(1:npar["beta"])]
  names.beta <- names(beta)
  names.gamma <- names(gamma)
  names(beta) <- sub("^beta\\\\.\\.", "", names.beta)
  names(gamma) <- sub("^gamma\\\\.\\.", "", names.gamma)
  cat("beta coefficients:\n")
  print(beta)
  cat("\ngamma coefficients:\n")
  print(gamma)
  if (!x$converged) warning("estimates did not converge")
  invisible(x)
}

```

We may now write "zipmod" methods for standard generic functions, such as `vcov()` and `print()`, shown in [Figure 10.13](#), and `summary()`, shown in [Figure 10.14](#). The default `coef()` method works with `print.zipmod()`, so no special method needs to be written.



```
ornstein.zip.2 # invokes the print() method
```

beta coefficients:

|           |              |           |           |
|-----------|--------------|-----------|-----------|
| intercept | log2(assets) | nationOTH | nationUK  |
| -0.352181 | 0.272374     | -0.057190 | -0.410996 |
| nationUS  | sectorBNK    | sectorCON | sectorFIN |
| -0.689423 | 0.057060     | -0.597324 | -0.052950 |
| sectorHLD | sectorMAN    | sectorMER | sectorMIN |
| 0.048333  | 0.222193     | 0.015237  | 0.249536  |
| sectorTRN | sectorWOD    |           |           |
| 0.168535  | 0.492357     |           |           |

gamma coefficients:

|           |              |           |          |
|-----------|--------------|-----------|----------|
| intercept | log2(assets) | nationOTH | nationUK |
| 3.02525   | -0.59127     | 1.00307   | -0.24937 |
| nationUS  |              |           |          |
| 1.31232   |              |           |          |

```
summary(ornstein.zip.2) # invokes the summary() method
```

beta coefficients:

|              | Estimate | Std.Err | z value | Pr(> z ) |
|--------------|----------|---------|---------|----------|
| intercept    | -0.3522  | 0.1388  | -2.54   | 0.01120  |
| log2(assets) | 0.2724   | 0.0120  | 22.77   | < 2e-16  |
| nationOTH    | -0.0572  | 0.0752  | -0.76   | 0.44725  |
| nationUK     | -0.4110  | 0.0895  | -4.59   | 4.4e-06  |
| nationUS     | -0.6894  | 0.0493  | -13.98  | < 2e-16  |
| sectorBNK    | 0.0571   | 0.0975  | 0.59    | 0.55824  |
| sectorCON    | -0.5973  | 0.2132  | -2.80   | 0.00509  |
| sectorFIN    | -0.0529  | 0.0757  | -0.70   | 0.48425  |
| sectorHLD    | 0.0483   | 0.1195  | 0.40    | 0.68587  |
| sectorMAN    | 0.2222   | 0.0767  | 2.90    | 0.00376  |
| sectorMER    | 0.0152   | 0.0870  | 0.18    | 0.86097  |
| sectorMIN    | 0.2495   | 0.0689  | 3.62    | 0.00029  |
| sectorTRN    | 0.1685   | 0.0786  | 2.14    | 0.03201  |
| sectorWOD    | 0.4924   | 0.0759  | 6.49    | 8.7e-11  |

gamma coefficients:

|              |        |       |       |         |
|--------------|--------|-------|-------|---------|
| intercept    | 3.025  | 1.550 | 1.95  | 0.05104 |
| log2(assets) | -0.591 | 0.160 | -3.69 | 0.00022 |
| nationOTH    | 1.003  | 0.894 | 1.12  | 0.26179 |
| nationUK     | -0.249 | 1.139 | -0.22 | 0.82669 |
| nationUS     | 1.312  | 0.516 | 2.54  | 0.01101 |

Deviance = 2227.4

**Figure 10.13** vcov () and print () methods for objects of class "zipmod".

```

summary.zipmod <- function(object, ...) {
  coef <- coef(object)
  npar <- object$npar
  beta <- coef[1:npar["beta"]]
  gamma <- coef[-(1:npar["beta"])]
  names.beta <- names(beta)
  names.gamma <- names(gamma)
  names(beta) <- sub("^\beta\backslash\.", "", names.beta)
  names(gamma) <- sub("^\gamma\backslash\.", "", names.gamma)
  vcov <- vcov(object, separate=TRUE)
  se.beta <- sqrt(diag(vcov[["beta"]]))
  z.beta <- beta/se.beta
  table.beta <- cbind(beta, se.beta, z.beta,
    2*(pnorm(abs(z.beta), lower.tail=FALSE)))
  colnames(table.beta) <- c("Estimate", "Std.Err",
    "z value", "Pr(>|z|)")
  rownames(table.beta) <- names(beta)
  se.gamma <- sqrt(diag(vcov[["gamma"]]))
  z.gamma <- gamma/se.gamma
  table.gamma <- cbind(gamma, se.gamma, z.gamma,
    2*(pnorm(abs(z.gamma), lower.tail=FALSE)))
  colnames(table.gamma) <- c("Estimate", "Std.Err",
    "z value", "Pr(>|z|)")
  rownames(table.gamma) <- names(gamma)
  result <- list(coef.beta=table.beta,
    coef.gamma=table.gamma, deviance=object$deviance,
    converged=object$converged)
  class(result) <- "summary.zipmod"
  result
}

print.summary.zipmod <- function(x, ...) {
  cat("beta coefficients:\n")
  printCoefmat(x$coef.beta, signif.legend=FALSE, ...)
  cat("\ngamma coefficients:\n")
  printCoefmat(x$coef.gamma, ...)
  cat("\nDeviance =", x$deviance, "\n")
  if (!x$converged) warning("estimates did not converge")
  invisible(x)
}

```

Our `print.zipmod()` method prints a brief report, while the output produced by `summary.zipmod()` is more extensive. The functions `cat()`, `print()`, and `printCoefmat()` are used to produce the printed output. We are already familiar with the generic `print()` function. The `cat()` function may also be used for output to the R console. Each *new-line character* ("`\n`") in the argument to `cat()` causes output to resume at the start of the next line. The `printCoefmat()` function prints coefficient matrices in a pleasing form.

**Figure 10.14** `summary()` method for objects of class "zipmod". The `summary()` method returns an object of class "summary.zipmod", which is printed by `print.summary.zipmod()`.

The screenshot shows the RStudio interface with the following components:

- Code Editor:** Displays the file `zipmodBugged.R` containing R code for a `zipmodBugged` function.
- Environment Browser:** Shows the current environment with variables `mu`, `pi`, `beta`, `beta.gamma`, `gamma`, `L1`, and `L2`.
- Console:** Displays the output of running the R script, including the definition of `zipmodBugged`, its execution, and the resulting values for `L1` and `L2`.

As mentioned, it is conventional for the arguments of a method to be the same as the arguments of the corresponding generic function, and thus we have arguments `x` and `...` for `print.zipmod()` and `object` and `...` for `summary.zipmod()`. It is also conventional for `print()` methods to return their first argument as an invisible result and for `summary()` methods to create and return objects to be

printed by a corresponding print () method. According to this scheme, summary.zipmod () returns an object of class "summary.zipmod", to be printed by the print () method print.summary.zipmod (). This approach produces summary () objects that can be used in further computations. For example, summary (ornstein.zip.2)\$coef.beta[, 3] returns the column of z-values from the coefficient table for the Poisson part of the model.

## 10.10 Writing Statistical-Modeling Functions in R\*

The ZIP model function that we developed in the preceding section ([Figure 10.12](#) on page 529) places the burden on the user to form the response vector  $y$  of counts; to construct the model matrices  $X$  and  $Z$  for the Poisson and logit parts of the model—including generating dummy variables from factors, creating interaction regressors, and so on; to select cases to use in a particular analysis; and to deal with missing data. Adding a formula method for our zipmod () generic function handles all of these tasks more conveniently.

We earlier introduced a default method, zipmod.default () (also in [Figure 10.12](#)), which takes a numeric model matrix  $X$  as its first argument. [Figure 10.15](#) shows a formula method, zipmod.formula (), for the zipmod () generic, the first argument of which is an R model formula.<sup>39</sup>

[39](#) Even though the first argument of the S3 generic function zipmod () is  $X$ , it is permissible to use the name formula for the first argument of the formula method zipmod.formula (), an exception to the rule that generic and method argument names should match. The other argument of the generic, ..., also appears in zipmod.formula (), at it should, and is passed through in the call to zipmod.default ()�.

After the definition of the local function combineFormulas () (explained below), the first few lines of zipmod.formula () handle the standard arguments for modeling functions: formula, data, subset, na.action, model, and contrasts (see [Section 4.9.1](#)). Comments in the code briefly indicate the purpose of each of the special commands for manipulating statistical models. Although you can simply regard these commands as an incantation to be invoked in writing a statistical-modeling function in R, an effective way to see more concretely how the function works is to enter the command debugonce (zipmod.formula) (see [Section 10.8](#) on debugging) and then to step through the example below line by line, examining what each line of code does. It is also instructive to consult the help pages for match.call (), eval.parent (), and so on. More details about writing statistical-modeling functions are available in the *R Project Developer Page* at <https://developer.r-project.org/model-fitting-functions.html>.<sup>40</sup>

**Figure 10.15** A formula method for zipmod (), using the standard arguments for

R modeling functions.

```

zipmod.formula <- function(formula, zformula, data, subset,
  na.action, model = TRUE, contrasts = NULL, ...){
  combineFormulas <- function(formula1, formula2){
    rhs <- as.character(formula2)[[2]]
    formula2 <- as.formula(paste("~ . +", rhs))
    update(formula1, formula2)
  }
  if (missing(zformula)) zformula <- formula[c(1, 3)]
  call <- match.call() # returns the function call
  mf <- match.call(expand.dots = FALSE)
    # the function call w/o ...
  args <- match(c("formula", "data", "subset", "na.action"),
    names(mf), 0) # which arguments are present?
  mf <- mf[c(1, args)]
  mf$drop.unused.levels <- TRUE
  mf[[1]] <- as.name("model.frame")
  mf$formula <- combineFormulas(formula, zformula)
  mf <- eval.parent(mf) # create a model frame
  terms <- attr(mf, "terms") # terms object for the model
  y <- model.response(mf) # response variable
  X <- model.matrix(formula, mf, contrasts)
  Z <- model.matrix(zformula, mf, contrasts)
  mod <- zipmod(X, y, Z, intercept.X=FALSE,
    intercept.Z=FALSE, ...)
  mod$na.action <- attr(mf, "na.action")
  mod$contrasts <- attr(X, "contrasts")
  if (model) {
    mod$model <- mf
    mod$X <- X
    mod$Z <- Z
    mod$y <- y
  }
  mod
}

```

[40](#) Because our `zipmod.formula()` function has *two* model formula arguments, as we'll explain presently, it's more complex than a typical single-formula statistical-modeling function. For an example of the latter, see the code for the `lm()` function, which you can examine by entering `lm` (without parentheses) at the R command prompt. By invoking `debugonce(lm)`, you can step through an example, examining the result produced by each line of the code. A complication here is that there are *two* model formulas, provided by the `formula` argument to construct the `X` model matrix for the Poisson part of the

model, and by the `zformula` argument to construct the `Z` model matrix for the logit part of the model. The `zformula` argument takes a one-sided formula, because the left-hand side of the model appears in the `formula` argument. The local function `combineFormulas()` consolidates the right-hand sides of the two formulas to produce a consistent data set (a *model frame*) with all the variables necessary to fit the model.<sup>41</sup> The `X` and `Z` matrices are then generated by calls to the `model.matrix()` function.

<sup>41</sup> By a consistent data set, we mean that the same rows are used to evaluate both model matrices and the response vector, for example, eliminating cases with missing data for any variables used in the model.

If the `zformula` argument is absent (as determined by a call to `missing()`), then it is set equal to the right-hand side of the `formula` argument. R model formulas can be manipulated as lists, and the first and third elements of `formula` are, respectively the `~` and the right-hand side of the model formula; the `~` is necessary to produce a one-sided formula for `zformula`. The second element of a two-sided formula is the left-hand side of the formula.

The `zipmod.formula()` function works by setting up a call to the default method of `zipmod()`, passing to the latter the Poisson model matrix `X`, the response vector `y`, and the logit model matrix `Z`. Because each model matrix already has a column of ones if the corresponding model formula implies an intercept, the `intercept.X` and `intercept.Z` arguments to `zipmod.default()` are set to `FALSE`, to avoid attaching redundant columns of ones. Finally, any other arguments to `zipmod.default()` are passed through via ....

We already have `print()`, `summary()`, and `vcov()` methods for "zipmod" objects, but to make the `zipmod()` function more useful, we should also write methods for other standard generic functions associated with statistical models, such as `anova()`, `fitted()`, and `residuals()`, among others, or, where appropriate, allow objects of class "zipmod" to inherit default methods. We illustrate with a `fitted()` method:

```

fitted.zipmod <- function(object, ...) {
  beta.gamma <- coef(object)
  npar <- object$npar
  beta <- beta.gamma[1:npar["beta"]]
  gamma <- beta.gamma[-(1:npar["beta"])]
  pi <- as.vector(1/(1 + exp(-(object$Z %*% gamma))))
  as.vector((1 - pi)*exp(object$X %*% beta))
}

```

Let us try out the new functions:

```

library("MASS") # for eqscplot()
ornstein.zip.3 <- zipmod(
  interlocks ~ log(assets) + nation + sector,
  ~ log(assets) + nation, data=Ornstein))

```

beta coefficients:

|           | (Intercept) | log(assets) | nationOTH | nationUK  | nationUS  |
|-----------|-------------|-------------|-----------|-----------|-----------|
| sectorBNK | -0.353097   | 0.393070    | -0.057149 | -0.410884 | -0.689410 |
| sectorMER | 0.056627    | -0.597393   | -0.053160 | 0.048310  | 0.222236  |
| sectorMIN | 0.015259    | 0.249458    | 0.168405  | 0.492339  |           |

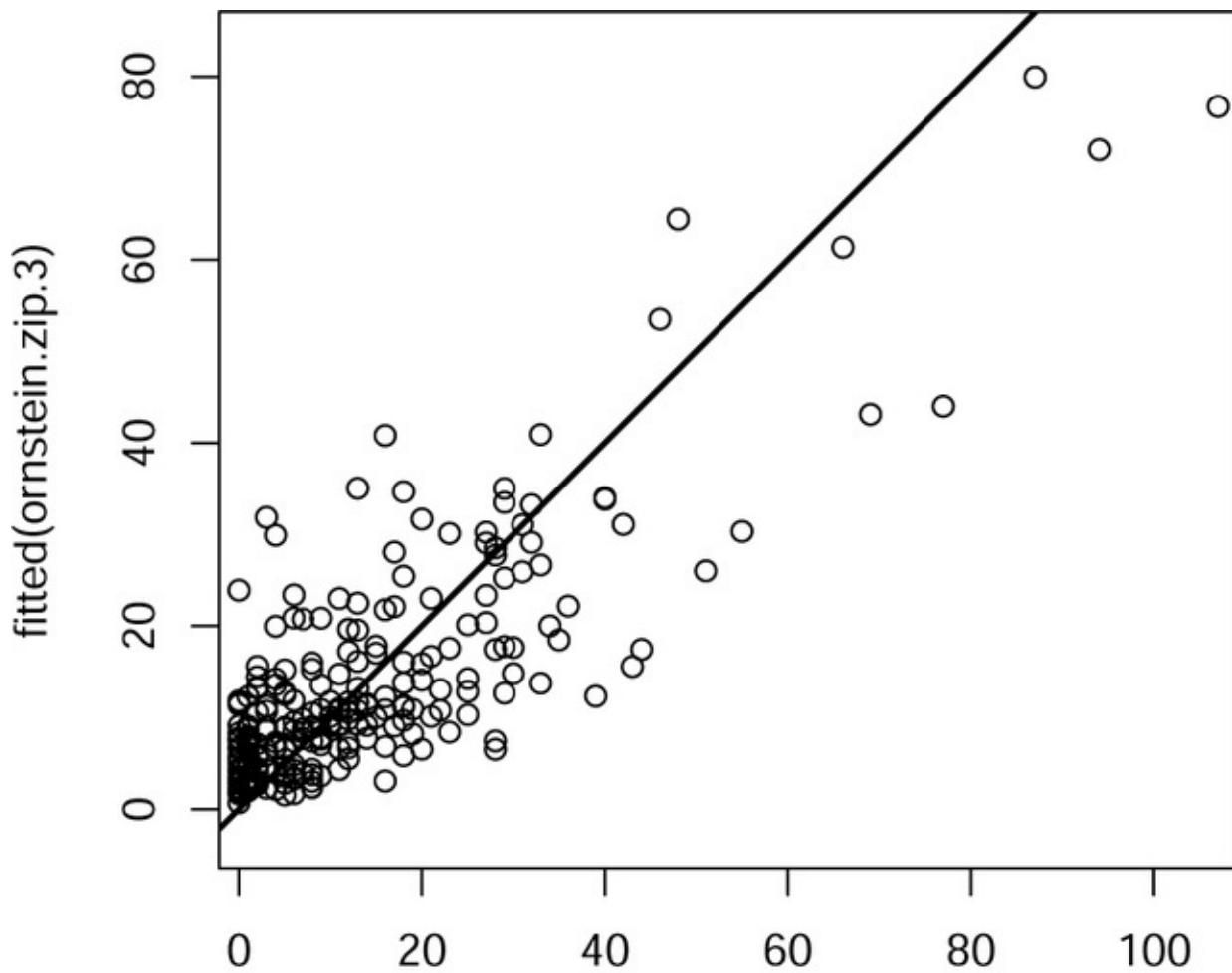
gamma coefficients:

|  | (Intercept) | log(assets) | nationOTH | nationUK | nationUS |
|--|-------------|-------------|-----------|----------|----------|
|  | 3.02719     | -0.85331    | 1.00316   | -0.24993 | 1.31224  |

```
eqscplot(Ornstein$interlocks, fitted(ornstein.zip.3))
```

```
abline(0, 1, lwd=2) # line y-hat = y
```

**Figure 10.16** Plot of fitted versus observed numbers of interlocks.



### Ornstein\$interlocks

We use the `eqscplot()` function from the **MASS** package for the plot of `fitted` versus observed numbers of interlocks (in [Figure 10.16](#)), which ensures that the two axes of the graph have the same scale (i.e., number of units per cm); the line on the graph is  $y = y$ .

## 10.11 Organizing Code for R Functions

You'll eventually accumulate a collection of functions written for various purposes. The best way to organize related R functions is to put them in a package, but package writing is beyond the scope of this book.<sup>42</sup> In this brief section, we make several suggestions for organizing your programming work short of writing a package:

- When you're working on a function, it is generally helpful to put it in its own file. The function *function-name* typically resides in the file *function-name.R*. For example, we saved the `zipmod()` function in a file named `zipmod.R`. “Sourcing” the file—either by invoking the `source()` function

(see below) or, more conveniently, by pressing the *Source* button at the upper right of the source file’s tab in the RStudio editor pane—reads the current version of the function into your R session. Thus, working on a function takes the form of an edit–source–test–edit loop.

- It’s natural to keep several related debugged and ready-to-use functions in the same source file, as in our file `zipmod-generic.R` (used in Sections 10.9 and 10.10), which includes the generic `zipmod()` function, together with related functions, such as `zipmod.default()`, `zipmod.formula()`, `print.zipmod()`, and `vcov.zipmod()`.
- Source files for functions associated with a particular project normally reside in the project’s directory or, for a complex project, in a subdirectory of the project directory. To use these functions in an R script or R Markdown document, you need only include an appropriate `source()` command in the script or document. The argument to the `source()` command is a character string specifying the path to the file to be sourced. Imagine that you’re working on the document `myDocument.Rmd` in the home directory of the current project and that you want to use functions in the file `my-project-functions.R` in the `R-source-files` subdirectory of the project. You can simply include an appropriate `source()` command in a code chunk near the top of the document; for example,

```
```{r echo=FALSE}
source ("R-source-files/my-project-functions.R")
```
```

In this case, specifying `echo=FALSE` hides the `source()` command when the document is compiled.

- If you write functions that you want to use routinely in your work, you can put an appropriate `source()` command in your `.Rprofile` file (see the Preface to the *R Companion*), so that the command is executed at the start of each R session.

[42](#) See the references at the end of the chapter.

## 10.12 Complementary Reading and References

- Several books by John Chambers and his colleagues describe various aspects of programming in R and its predecessor S: Becker et al. (1988), Chambers and Hastie (1992b), Chambers (1998), Chambers (2008), and Chambers (2016).
- Grolemund (2014) is a readable, easy-to-follow, basic introduction to R programming, which also introduces RStudio.
- Gentleman (2009) and Wickham (2015a) deal with more advanced aspects

of R programming. Wickham (2015b) is a companion volume that deals with writing R packages; Wickham’s approach to package development is somewhat idiosyncratic, but it is carefully considered.

- We, along with Brad Price, are planning a general text on R programming, extending the material on the topic in the *R Companion*, and including an explanation of how to write R packages.
- Books that treat traditional topics in statistical computing, such as optimization, simulation, probability calculations, and computational linear algebra using R, include Braun and Murdoch (2016), Jones, Maillardet, and Robinson (2014), and Rizzo (2008).
- Kennedy and Gentle (1980), Thisted (1988), Voss (2013), and most other general books on statistical computing discuss the nuts and bolts of generating pseudo-random numbers.

# References

- Adler, D., & Murdoch, D. (2017). rgl: 3D visualization using OpenGL [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=rgl> (R package version 0.98.1)
- Agresti, A. (2007). An introduction to categorical data analysis (2nd ed.). Hoboken, NJ: Wiley.
- Agresti, A. (2010). Analysis of ordinal categorical data (2nd ed.). Hoboken NJ: Wiley.
- Andrews, F. (2012). playwith: A GUI for interactive plots using GTK+ [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=playwith> (R package version 0.9-54)
- Atkinson, A. C. (1985). Plots, transformations and regression: An introduction to graphical methods of diagnostic regression analysis. Oxford: Clarendon Press.
- Bates, D., Mächler, M., Bolker, B., & Walker, S. (2015). Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67 (1), 1–48. Retrieved from <https://arxiv.org/abs/1406.5823>
- Bates, D., & Maechler, M. (2018). Matrix: Sparse and dense matrix classes and methods [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=Matrix> (R package version 1.2-14)
- Becker, R. A., Chambers, J. M., & Wilks, A. R. (1988). The new S language: A programming environment for data analysis and graphics. Pacific Grove, CA: Wadsworth.
- Becker, R. A., Wilks, A. R., Brownrigg, R., Minka, T. P., & Deckmyn, A. (2018). maps: Draw geographical maps [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=maps> (R package version 3.3.0)
- Benjamin, D. J., Berger, J. O., Johannesson, M., Nosek, B. A., Wagenmakers, E.-J., Berk, R., ... others (2017). Redefine statistical significance. *Nature Human Behaviour*, 6–10. Retrieved from <https://www.nature.com/articles/s41562-017-0189-z>
- Berndt, E. R. (1991). The practice of econometrics: Classic and contemporary. Reading, MA: Addison-Wesley.
- Birch, M. W. (1963). Maximum likelihood in three-way contingency tables. *Journal of the Royal Statistical Society. Series B (Methodological)*, 25(1), 220–233. Retrieved from <http://www.jstor.org/stable/2984562>
- Bowman, A. W., & Azzalini, A. (1997). Applied smoothing techniques for data analysis: The kernel approach with S-Plus illustrations. Oxford: Oxford University Press.

- Box, G. E. P., & Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26 (2), 211–252. Retrieved from <http://www.jstor.org/stable/2984418>
- Box, G. E. P., & Tidwell, P. W. (1962). Transformation of the independent variables. *Technometrics*, 4 (4), 531–550. Retrieved from <http://www.jstor.org/stable/1266288>
- Brant, R. (1990). Assessing proportionality in the proportional odds model for ordinal logistic regression. *Biometrics*, 46, 1171–1178. Retrieved from [http://www.jstor.org/stable/2532457?seq=1#page\\_scan\\_tab\\_contents](http://www.jstor.org/stable/2532457?seq=1#page_scan_tab_contents)
- Braun, W. J., & Murdoch, D. J. (2016). A first course in statistical programming with R (2nd ed.). Cambridge UK: Cambridge University Press.
- Bretz, F., Hothorn, T., & Westfall, P. (2011). Multiple comparisons using R. Boca Raton, FL: CRC/Chapman & Hall.
- Breusch, T. S., & Pagan, A. R. (1979). A simple test for heteroscedasticity and random coefficient variation. *Econometrica*, 47 (5), 1287–1294. Retrieved from <http://www.jstor.org/stable/1911963>
- Brooks, M. E., Kristensen, K., van Benthem, K. J., Magnusson, A., Berg, C. W., Nielsen, A., Skaug, H. J., Maechler, M., & Bolker, B. M. (2017). glmmTMB balances speed and flexibility among packages for zero-inflated generalized linear mixed modeling. *The R Journal*, 9(2), 378–400. Retrieved from <https://journal.r-project.org/archive/2017/RJ-2017-066/index.html>
- Campbell, A., Converse, P. E., Miller, P. E., & Stokes, D. E. (1960). *The American voter*. New York: Wiley.
- Canty, A., & Ripley, B. D. (2017). boot: Bootstrap functions [Computer software manual]. (R package version 1.3-20)
- Chambers, J. M. (1992). Linear models. In J. M. Chambers & T. J. Hastie (Eds.), *Statistical models in S* (pp. 95–144). Pacific Grove, CA: Wadsworth.
- Chambers, J. M. (1998). *Programming with data: A guide to the S language*. New York: Springer.
- Chambers, J. M. (2008). *Software for data analysis: Programming with R*. New York: Springer.
- Chambers, J. M. (2016). *Extending R*. Boca Raton, FL: CRC/Chapman & Hall.
- Chambers, J. M., Cleveland, W. S., Kleiner, B., & Tukey, P. A. (1983). *Graphical methods for data analysis*. Belmont, CA: Wadsworth.
- Chambers, J. M., & Hastie, T. J. (1992a). Statistical models. In J. M. Chambers & T. J. Hastie (Eds.), *Statistical models in S* (pp. 13–44). Pacific Grove, CA: Wadsworth.
- Chambers, J. M., & Hastie, T. J. (Eds.). (1992b). *Statistical models in S*. Pacific Grove, CA: Wadsworth.

- Chan, C., Chan, G. C., Leeper, T. J., & Becker, J. (2017). rio: A swiss-army knife for data file I/O [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=rio> (R package version 0.5.10)
- Chang, W., Cheng, J., Allaire, J. J., Xie, Y., & McPherson, J. (2017). shiny: Web application framework for R [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=shiny> (R package version 1.0.5)
- Christensen, R., Pearson, L. M., & Johnson, W. (1992). Case-deletion diagnostics for mixed models. *Technometrics*, 34, 38–45. Retrieved from <https://www.tandfonline.com/doi/abs/10.1080/00401706.1992.10485231>
- Christensen, R. H. B. (2018). ordinal: Regression models for ordinal data [Computer software manual]. Retrieved from <http://www.cran.r-project.org/package=ordinal/> (R package version 2018.4-19)
- Cleveland, W. S. (1979). Robust locally weighted regression and smoothing scatter-plots. *Journal of the American Statistical Association*, 74 (368), 829–836. Retrieved from <http://www.jstor.org/stable/2286407>
- Cleveland, W. S. (1993). Visualizing data. Summit, NJ: Hobart Press.
- Cleveland, W. S. (1994). The elements of graphing data, revised edition. Summit, NJ: Hobart Press.
- Cleveland, W. S., Grosse, E., & Shyu, W. M. (1992). Local regression models. In J. M. Chambers & T. J. Hastie (Eds.), *Statistical models in S* (pp. 309–376). Pacific Grove, CA: Wadsworth.
- Clogg, C. C., & Shihadeh, E. S. (1994). Statistical models for ordinal variables. Thousand Oaks, CA: Sage.
- Conover, W. J., Johnson, M. E., & Johnson, M. M. (1981). A comparative study of tests for homogeneity of variances, with applications to the outer continental shelf bidding data. *Technometrics*, 23, 351–361. Retrieved from <https://amstat.tandfonline.com/doi/abs/10.1080/00401706.1981.10487680>
- Cook, D., & Swayne, D. F. (2009). Interative dynamic graphics for data analysis: With R and GGobi. New York: Springer.
- Cook, R. D. (1977). Detection of influential observations in linear regression. *Technometrics*, 19(1), 15–18. Retrieved from <http://www.jstor.org/stable/1268249>
- Cook, R. D. (1993). Exploring partial residual plots. *Technometrics*, 35(4), 351–362. Retrieved from <http://www.jstor.org/stable/1270269>
- Cook, R. D. (1998). *Regression graphics: Ideas for studying regressions through graphics*. New York: Wiley.
- Cook, R. D., & Weisberg, S. (1982). *Residuals and influence in regression*. New York: CRC/Chapman & Hall.
- Cook, R. D., & Weisberg, S. (1983). Diagnostics for heteroscedasticity in

- regression. *Biometrika*, 70(1), 1–10. Retrieved from <http://www.jstor.org/stable/2335938>
- Cook, R. D., & Weisberg, S. (1991). Added variable plots in linear regression. In W. Stahel & S. Weisberg (Eds.), *Directions in robust statistics and diagnostics, part I* (pp. 47–60). Springer.
- Cook, R. D., & Weisberg, S. (1994). Transforming a response variable for linearity. *Biometrika*, 81(4), 731–737. Retrieved from <http://www.jstor.org/stable/2337076>
- Cook, R. D., & Weisberg, S. (1997). Graphics for assessing the adequacy of regression models. *Journal of the American Statistical Association*, 92(438), 490–499. Retrieved from <http://www.jstor.org/stable/2965698>
- Cook, R. D., & Weisberg, S. (1999). *Applied regression including computing and graphics*. New York: Wiley.
- Cowles, M., & Davis, C. (1987). The subject matter of psychology: Volunteers. *British Journal of Social Psychology*, 26, 97–102. Retrieved from <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.2044-8309.1987.tb00769.x>
- Cox, D. R., & Snell, E. J. (1968). A general definition of residuals. *Journal of the Royal Statistical Society. Series B (Methodological)*, 30(2), 248–275. Retrieved from <http://www.jstor.org/stable/2984505>
- Cunningham, R., & Heathcote, C. (1989). Estimating a non-gaussian regression model with multicollinearity. *Australian Journal of Statistics*, 31, 12–17. Retrieved from <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-842X.1989.tb00494.x>
- Davis, C. (1990). Body image and weight preoccupation: A comparison between exercising and non-exercising women. *Appetite*, 15, 13–21. Retrieved from <https://www.sciencedirect.com/science/article/pii/019566639090096Q>
- Davis, C., Blackmore, E., Katzman, D. K., & Fox, J. (2005). Female adolescents with anorexia nervosa and their parents: A case-control study of exercise attitudes and behaviours. *Psychological Medicine*, 35, 377–386. Retrieved from <https://doi.org/10.1017/S0033291704003447>
- Davison, A. C., & Hinkley, D. V. (1997). *Bootstrap methods and their application*. Cambridge: Cambridge University Press.
- Demidenko, E., & Stukel, T. A. (2005). Influence analysis for linear mixed-effects models. *Statistics in Medicine*, 24, 893–909. Retrieved from <https://onlinelibrary.wiley.com/doi/full/10.1002/sim.1974>
- Dowle, M., & Srinivasan, A. (2017). *data.table: Extension of ‘data.frame’ [Computer software manual]*. Retrieved from <https://CRAN.R-project.org/package=data.table> (R package version 1.10.4-3)
- Duncan, O. D. (1961). A socioeconomic index for all occupations. In A. J. Reiss

- Jr. (Ed.), Occupations and social status (pp. 109–138). New York: Free Press.
- Durbin, J., & Watson, G. S. (1950). Testing for serial correlation in least squares regression I. *Biometrika*, 37, 409–428. Retrieved from  
[http://www.jstor.org/stable/2332391?seq=1#page\\_scan\\_tab\\_contents](http://www.jstor.org/stable/2332391?seq=1#page_scan_tab_contents)
- Durbin, J., & Watson, G. S. (1951). Testing for serial correlation in least squares regression II. *Biometrika*, 38, 159–178. Retrieved from  
<http://www.jstor.org/stable/2332325>
- Efron, B. (2003). The statistical century. In J. Panaretos (Ed.), *Stochastic musings: Perspectives from the pioneers of the late 20th century* (pp. 29–44). Mahwah, NJ: Lawrence Erlbaum Associates.
- Efron, B., & Tibshirani, R. J. (1993). An introduction to the bootstrap. New York: CRC/Chapman & Hall.
- Erickson, E. P., Kadane, J. B., & Tukey, J. W. (1989). Adjusting the 1990 Census of Population and Housing. *Journal of the American Statistical Association*, 84, 927–944. Retrieved from  
<https://www.tandfonline.com/doi/abs/10.1080/01621459.1989.10478857>
- Feinerer, I., & Hornik, K. (2017). tm: Text mining package [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=tm> (R package version 0.7-3)
- Feinerer, I., Hornik, K., & Meyer, D. (2008, March). Text mining infrastructure in R. *Journal of Statistical Software*, 25(5), 1–54. Retrieved from  
<http://www.jstatsoft.org/v25/i05/>
- Fienberg, S. E. (1980). The analysis of cross-classified categorical data (2nd ed.). Cambridge, MA: MIT Press.
- Fox, J. (1991). Regression diagnostics: An introduction. Newbury Park, CA: Sage.
- Fox, J. (2000). Nonparametric simple regression: Smoothing scatterplots. Thousand Oaks, CA: Sage.
- Fox, J. (2002). An R and S-PLUS companion to applied regression. Thousand Oaks, CA: Sage.
- Fox, J. (2009a). Aspects of the social organization and trajectory of the R Project. *The R Journal*, 1(2). Retrieved from [http://journal.r-project.org/archive/2009-2/RJournal\\_2009-2\\_Fox.pdf](http://journal.r-project.org/archive/2009-2/RJournal_2009-2_Fox.pdf)
- Fox, J. (2009b). A mathematical primer for social statistics. Thousand Oaks, CA: Sage.
- Fox, J. (2016). Applied regression analysis and generalized linear models (3rd ed.). Thousand Oaks, CA: Sage.
- Fox, J. (2017). Using the R Commander: A point-and-click interface for R. Boca Raton, FL: CRC/Chapman & Hall.

- Fox, J., & Andersen, R. (2006). Effect displays for multinomial and proportional-odds logit models. *Sociological Methodology*, 36, 225–255.
- Fox, J., & Guyer, M. (1978). “Public” choice and cooperation in n-person prisoner’s dilemma. *The Journal of Conflict Resolution*, 22(3), 469–481. Retrieved from <http://www.jstor.org/stable/173730>
- Fox, J., & Monette, G. (1992). Generalized collinearity diagnostics. *Journal of the American Statistical Association*, 87 (417), 178–183. Retrieved from <http://www.jstor.org/stable/2290467>
- Fox, J., & Weisberg, S. (2018, in press). Visualizing fit and lack of fit in complex regression models with predictor effect plots and partial residuals. *Journal of Statistical Software*.
- Freedman, D. (2006). On the so-called ‘Huber sandwich estimator’ and ‘robust standard errors’. *The American Statistician*, 60(4), 299–302. Retrieved from <http://dx.doi.org/10.1198/000313006X152207>
- Freedman, D., & Diaconis, P. (1981). On the histogram as a density estimator. *Zeitschrift fur Wahrscheinlichkeitstheorie und verwandte Gebiete*, 57, 453–476. Retrieved from <https://link.springer.com/article/10.1007%2FBF01025868>
- Freedman, J. L. (1975). *Crowding and behavior*. New York: Viking.
- Friendly, M., & Fox, J. (2018). matlib: Matrix functions for teaching and learning linear algebra and multivariate statistics [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=matlib> (R package version 0.9.1)
- Friendly, M., Monette, G., & Fox, J. (2013). Elliptical insights: Understanding statistical methods through elliptical geometry. *Statistical Science*, 28, 1–39. Retrieved from <https://projecteuclid.org/euclid.ss/1359468407>
- Gelman, A., & Hill, J. (2007). *Data analysis using regression and multilevel/hierarchical models*. Cambridge UK: Cambridge University Press.
- Gelman, A., & Hill, J. (2015). mi: Missing data imputation and model checking [Computer software manual]. Retrieved from <https://cran.r-project.org/web/packages/mi/index.html> (R package Verion 1.0)
- Gentleman, R. (2009). *R programming for bioinformatics*. Boca Raton, FL: CRC/Chapman & Hall.
- Grolemund, G. (2014). *Hands-on programming with R*. Sebastopol, CA: O’Reilly.
- Grolemund, G., & Wickham, H. (2011). Dates and times made easy with lubridate. *Journal of Statistical Software*, 40(3), 1–25. Retrieved from <http://www.jstatsoft.org/v40/i03/>
- Halekoh, U., & Højsgaard, S. (2014). A Kenward-Roger approximation and

- parametric bootstrap methods for tests in linear mixed models—the R package pbkrtest. *Journal of Statistical Software*, 59(9), 1–30. Retrieved from <http://www.jstatsoft.org/v59/i09/>
- Harper, E. K., Paul, W. J., Mech, L. D., & Weisberg, S. (2008). Effectiveness of lethal, directed wolf-depredation control in Minnesota. *Journal of Wildlife Management*, 72(3), 778–784. Retrieved from <http://www.bioone.org/doi/abs/10.2193/2007-273> doi: 10.2193/2007-273
- Hawkins, D. M., & Weisberg, S. (2017). Combining the Box-Cox power and generalised log transformations to accommodate negative responses in linear and mixed-effects linear models. *South African Statistics Journal*, 51, 317–328. Retrieved from <https://journals.co.za/content/journal/10520/EJC-bd05f9440>
- Honaker, J., King, G., & Blackwell, M. (2011). Amelia II: A program for missing data. *Journal of Statistical Software*, 45(7), 1–47. Retrieved from <http://www.jstatsoft.org/v45/i07/>
- Huber, P., & Ronchetti, E. M. (2009). Robust statistics (2nd ed.). Hoboken NJ: Wiley.
- Ihaka, R., & Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5, 299–314. Retrieved from <https://www.tandfonline.com/doi/abs/10.1080/10618600.1996.10474713>
- Ioannidis, J. P. (2005). Why most published research findings are false. *PLoS medicine*, 2(8), e124. Retrieved from <http://journals.plos.org/plosmedicine/article?id=10.1371/journal.pmed.0020124>
- Jones, O., Maillardet, R., & Robinson, A. (2014). Introduction to scientific programming and simulation using R (2nd ed.). Boca Raton, FL: CRC/Chapman & Hall.
- Kahle, D., & Wickham, H. (2013). ggmap: Spatial visualization with ggplot2. *The R Journal*, 5(1), 144–161. Retrieved from <http://journal.r-project.org/archive/2013-1/kahle-wickham.pdf>
- Kennedy, W. J., Jr., & Gentle, J. E. (1980). Statistical computing. New York: Marcel Dekker.
- Kenward, M. G., & Roger, J. (1997). Small sample inference for fixed effects from restricted maximum likelihood. *Biometrics*, 53, 983–997. Retrieved from [http://www.jstor.org/stable/2533558?seq=1#page\\_scan\\_tab\\_contents](http://www.jstor.org/stable/2533558?seq=1#page_scan_tab_contents)
- Koenker, R. (2018). quantreg: Quantile regression [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=quantreg> (R package version 5.35)
- Laird, N. M., & Ware, J. H. (1982). Random-effects models for longitudinal

- data. *Biometrics*, 38, 963–974. Retrieved from  
[http://www.jstor.org/stable/2529876?seq=1#page\\_scan\\_tab\\_contents](http://www.jstor.org/stable/2529876?seq=1#page_scan_tab_contents)
- Landwehr, J. M., Pregibon, D., & Shoemaker, A. C. (1980). Some graphical procedures for studying a logistic regression fit. In *Proceedings of the business and economics statistics section* (pp. 15–20). Alexandria, VA: American Statistical Association.
- Lawrence, M., & Temple Lang, D. (2010). RGtk2: A graphical user interface toolkit for R. *Journal of Statistical Software*, 37 (8), 1–52. Retrieved from  
<http://www.jstatsoft.org/v37/i08/>
- Leisch, F. (2002). Sweave, part I: Mixing R and Latex. *R News*, 2(3), 28–31. Retrieved from [https://cran.r-project.org/doc/Rnews/Rnews\\_2002-3.pdf](https://cran.r-project.org/doc/Rnews/Rnews_2002-3.pdf)
- Lemon, J. (2017). Plotrix: Various plotting functions [Computer software manual]. Retrieved from <https://cran.r-project.org/web/packages/plotrix/index.html> (R package version 3.7)
- Lenth, R. V. (2016). Least-squares means: The R package lsmeans. *Journal of Statistical Software*, 69(1), 1–33. Retrieved from  
<https://jstatsoft.tr1k.de/article/view/v069i01/v69i01.pdf> doi:  
10.18637/jss.v069.i01
- Lenth, R. V. (2018). emmeans: Estimated marginal means, aka least-squares means [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=emmeans> (R package version 1.1.3)
- Ligges, U., & Fox, J. (2008). R help desk: How can I avoid this loop or make it faster? *R News*, 8(1), 46–50. Retrieved from [http://CRAN.R-project.org/doc/Rnews/Rnews\\_2008-1.pdf](http://CRAN.R-project.org/doc/Rnews/Rnews_2008-1.pdf)
- Little, R. J. A., & Rubin, D. B. (2002). *Statistical analysis with missing data* (2nd ed.). Hoboken, NJ: Wiley.
- Long, J. S. (1997). *Regression models for categorical and limited dependent variables*. Thousand Oaks, CA: Sage.
- Long, J. S., & Ervin, L. H. (2000). Using heteroscedasticity consistent standard errors in the linear regression model. *The American Statistician*, 54 (3), 217–224. Retrieved from <http://www.jstor.org/stable/2685594>
- Lumley, T. (2010). *Complex surveys: A guide to analysis using R*. Hoboken, NJ: Wiley.
- Lumley, T. (2013). biglm: bounded memory linear and generalized linear models [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=biglm> (R package version 0.9-1)
- Maechler, M. (2018). sfsmisc: Utilities from 'seminar fuer statistik' ETH Zurich [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=sfsmisc> (R package version 1.1-2)

- Mallows, C. L. (1986). Augmented partial residuals. *Technometrics*, 28(4), 313–319. Retrieved from <http://www.jstor.org/stable/1268980>
- McCullagh, P., & Nelder, J. A. (1989). Generalized linear models (2nd ed.). London: CRC/Chapman & Hall.
- Mersmann, O. (2018). microbenchmark: Accurate timing functions [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=microbenchmark> (R package version 1.4-4.1)
- Milliken, G. A., & Johnson, D. E. (2004). Analysis of messy data: designed experiments (2nd ed., Vol. 1). Boca Raton, FL: CRC/Chapman & Hall.
- Minneapolis Police Department. (2018). Minneapolis police stop data. <http://opendata.minneapolismn.gov/datasets/police-stop-data>. (Accessed: 2018-04-27)
- Minnesota Compass. (2018). Minneapolis–Saint Paul neighborhoods. <http://www.mncompass.org/profiles/neighborhoods/minneapolis-saint-paul#!community-areas>. (Accessed: 2018-04-27)
- Moore, D. S., & McCabe, G. P. (1993). Introduction to the practice of statistics (2nd ed.). New York: Freeman.
- Mosteller, F., & Tukey, J. W. (1977). Data analysis and regression: A second course in statistics. Reading, MA: Addison-Wesley.
- Mroz, T. A. (1987). The sensitivity of an empirical model of married women's hours of work to economic and statistical assumptions. *Econometrica*, 55(4), 765–799. Retrieved from <http://www.jstor.org/stable/1911029>
- Munzert, S., Rubba, C., Meißner, P., & Nyhuis, D. (2014). Automated data collection with R: A practical guide to web scraping and text mining. Hoboken, NJ: Wiley.
- Murrell, P. (2011). R graphics (2nd ed.). Boca Raton, FL: CRC/Chapman & Hall.
- Murrell, P., & Ihaka, R. (2000). An approach to providing mathematical annotation in plots. *Journal of Computational and Graphical Statistics*, 9(3), 582–599. Retrieved from <http://www.jstor.org/stable/1390947>
- Nelder, J. A. (1977). A reformulation of linear models. *Journal of the Royal Statistical Society. Series A (General)*, 140(1), 48–77. Retrieved from <http://www.jstor.org/stable/2344517>
- Nelder, J. A., & Wedderburn, R. W. M. (1972). Generalized linear models. *Journal of the Royal Statistical Society. Series A (General)*, 135(3), 370–384. Retrieved from <http://www.jstor.org/stable/2344614>
- Nieuwenhuis, R., te Grotenhuis, M., & Pelzer, B. (2012). influence.ME: Tools for detecting influential data in mixed effects models. *The R Journal*, 4 (2), 38–47. Retrieved from <https://journal.r-project.org/archive/2012/RJ-2012->

[011/index.html](#)

- NORC. (2018). General social survey. <http://gss.norc.org>. (Accessed: 2018-04-27)
- O'Brien, R. G., & Kaiser, M. K. (1985). MANOVA method for analyzing repeated measures designs: An extensive primer. *Psychological Bulletin*, 97, 316–333. Retrieved from <http://psycnet.apa.org/record/1985-19145-001>
- Ornstein, M. D. (1976). The boards and executives of the largest Canadian corporations: Size, composition, and interlocks. *Canadian Journal of Sociology*, 1, 411–437. Retrieved from [http://www.jstor.org/stable/3339754?seq=1#page\\_scan\\_tab\\_contents](http://www.jstor.org/stable/3339754?seq=1#page_scan_tab_contents)
- Pan, J., Fie, Y., & Foster, P. (2014). Case-deletion diagnostics for linear mixed models. *Technometrics*, 56, 269–281. Retrieved from <https://www.tandfonline.com/doi/abs/10.1080/00401706.2013.810173>
- Phipps, P., Maxwell, J. W., & Rose, C. (2009). 2009 annual survey of the mathematical sciences. *Notices of the American Mathematical Society*, 57, 250–259.
- Pinheiro, J., & Bates, D. (2000). Mixed-effects models in S and S-PLUS. New York: Springer.
- Pinheiro, J., Bates, D., DebRoy, S., Sarkar, D., & R Core Team. (2018). nlme: Linear and nonlinear mixed effects models [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=nlme> (R package version 3.1-137)
- Powers, D. A., & Xie, Y. (2008). Statistical methods for categorical data analysis (2nd ed.). Bingley UK: Emerald Group Publishing.
- Pregibon, D. (1981). Logistic regression diagnostics. *The Annals of Statistics*, 9(4), 705–724. Retrieved from <http://www.jstor.org/stable/2240841>
- R Core Team. (2018). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from <https://www.R-project.org/>
- Raudenbush, S. W., & Bryk, A. S. (2002). Hierarchical linear models: Applications and data analysis methods (2nd ed.). Thousand Oaks, CA: Sage.
- Ripley, B. D. (2001, January). Using databases with R. *R News*, 1(1), 18–20. Retrieved from [http://CRAN.R-project.org/doc/Rnews/Rnews\\_2001-1.pdf](http://CRAN.R-project.org/doc/Rnews/Rnews_2001-1.pdf)
- Rizzo, M. L. (2008). Statistical computing with R. Boca Raton, FL: CRC/Chapman & Hall.
- Sall, J. (1990). Leverage plots for general linear hypotheses. *The American Statistician*, 44 (4), 308–315. Retrieved from <http://www.jstor.org/stable/2684358>
- Sarkar, D. (2008). Lattice: Multivariate data visualization with R. New York:

- Springer.
- Sarkar, D., & Andrews, F. (2016). latticeExtra: Extra graphical utilities based on lattice [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=latticeExtra> (R package version 0.6-28)
- Satterthwaite, F. E. (1946). An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2, 110–114. Retrieved from [http://www.jstor.org/stable/3002019?seq=1#page\\_scan\\_tab\\_contents](http://www.jstor.org/stable/3002019?seq=1#page_scan_tab_contents)
- Schafer, J. L. (1997). Analysis of incomplete multivariate data. New York: CRC/Chapman & Hall.
- Scheffé, H. (1959). The analysis of variance. New York: Wiley.
- Shi, L., & Chen, G. (2008). Case deletion diagnostics in multilevel models. *Journal of Multivariate Analysis*, 99, 1860–1877. Retrieved from <https://www.sciencedirect.com/science/article/pii/S0047259X0800033X>
- Silverman, B. W. (1986). Density estimation for statistics and data analysis. London: CRC/Chapman & Hall.
- Simonoff, J. S. (2003). Analyzing categorical data. New York: Springer.
- Singer, J. D. (1998). Using SAS PROC MIXED to fit multilevel models, hierarchical models, and individual growth models. *Journal of Educational and Behavioral Statistics*, 24, 323–355. Retrieved from <http://journals.sagepub.com/doi/abs/10.3102/10769986023004323>
- Snijders, T. A. B., & Bosker, R. J. (2012). Multilevel analysis: An introduction to basic and advanced multilevel modeling (2nd ed.). Thousand Oaks, CA: Sage.
- Spector, P. (2008). Data manipulation with R. New York: Springer.
- Stine, R., & Fox, J. (Eds.). (1996). Statistical computing environments for social research. Thousand Oaks, CA: Sage.
- Stroup, W. W. (2013). Generalized linear mixed models: Modern concepts, methods and applications. Boca Raton, FL: CRC Press.
- Swayne, D. F., Cook, D., & Buja, A. (1998). XGobi: Interactive dynamic data visualization in the X Window system. *Journal of Computational and Graphical Statistics*, 7 (1), 113–130. Retrieved from <http://www.jstor.org/stable/1390772>
- Therneau, T., Atkinson, B., & Ripley, B. (2018). rpart: Recursive partitioning and regression trees [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=rpart> (R package version 4.1-13)
- Thisted, R. A. (1988). Elements of statistical computing: Numerical computation (Vol. 1). Boca Raton, FL: CRC/Chapman & Hall.
- Tufte, E. R. (1983). The visual display of quantitative information. Cheshire, CT: Graphics Press.

- Tukey, J. W. (1949). One degree of freedom for non-additivity. *Biometrics*, 5(3), 232–242. Retrieved from <http://www.jstor.org/stable/3001938>
- Tukey, J. W. (1977). Exploratory data analysis. Reading, MA: Addison-Wesley.
- Urbanek, S. (2017). rJava: Low-level R to Java interface [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=rJava> (R package version 0.9-9)
- Urbanek, S., & Wichtrey, T. (2013). iplots: iPlots—interactive graphics for R [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=iplots> (R package version 1.1-7)
- van Buuren, S., & Groothuis-Oudshoorn, K. (2011). mice: Multivariate imputation by chained equations in R. *Journal of Statistical Software*, 45(3), 1–67. Retrieved from <http://www.jstatsoft.org/v45/i03/>
- Varshney, L. R., & Sun, J. Z. (2013). Why do we perceive logarithmically? *Significance*, 10(1), 28–31. Retrieved from <http://onlinelibrary.wiley.com/doi/10.1111/j.1740-9713.2013.00636.x/pdf>
- Velilla, S. (1993). A note on the multivariate Box-Cox transformation to normality. *Statistics and Probability Letters*, 17, 259–263. Retrieved from [http://dx.doi.org/10.1016/0167-7152\(93\)90200-3](http://dx.doi.org/10.1016/0167-7152(93)90200-3)
- Venables, W. N., & Ripley, B. D. (2002). Modern applied statistics with S (4th ed.). New York: Springer.
- Voss, J. (2013). An introduction to statistical computing: A simulation-based approach. Hoboken, NJ: Wiley.
- Wang, P. C. (1985). Adding a variable in generalized linear models. *Technometrics*, 27, 273–276. Retrieved from <https://amstat.tandfonline.com/doi/abs/10.1080/00401706.1985.10488051#.Wi>
- Wang, P. C. (1987). Residual plots for detecting nonlinearity in generalized linear models. *Technometrics*, 29(4), 435–438. Retrieved from <http://www.jstor.org/stable/1269454>
- Wasserstein, R. L., & Lazar, N. A. (2016). The ASA's statement on p-values: Context, process, and purpose. *The American Statistician*, 70(2), 129–133. Retrieved from <https://doi.org/10.1080/00031305.2016.1154108>
- Weisberg, S. (2004). Lost opportunities: Why we need a variety of statistical languages. *Journal of Statistical Software*, 13(1), 1–12. Retrieved from <http://www.jstatsoft.org/v13/i01>
- Weisberg, S. (2014). Applied linear regression (4th ed.). Hoboken, NJ: Wiley.
- White, H. (1980). A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity. *Econometrica*, 48(4), 817–838. Retrieved from <http://www.jstor.org/stable/1912934>
- Wickham, H. (2007). Reshaping data with the reshape package. *Journal of*

- Statistical Software, 21(12), 1–20. Retrieved from  
<http://www.jstatsoft.org/v21/i12/>
- Wickham, H. (2015a). Advanced R. Boca Raton, FL: CRC/Chapman & Hall.
- Wickham, H. (2015b). R packages. Sebastopol, CA: O'Reilly.
- Wickham, H. (2016). ggplot2: Using the grammar of graphics with R (2nd ed.). New York: Springer.
- Wickham, H. (2017). tidyverse: Easily install and load 'tidyverse' packages [Computer software manual]. Retrieved from <https://CRAN.R-project.org/package=tidyverse> (R package version 1.2.1)
- Wickham, H., & Grolemund, G. (2016). R for data science. Sebastopol, CA: O'Reilly.
- Wilkinson, G. N., & Rogers, C. E. (1973). Symbolic description of factorial models for analysis of variance. Journal of the Royal Statistical Society. Series C (Applied Statistics), 22(3), 392–399. Retrieved from  
<http://www.jstor.org/stable/2346786>
- Wilkinson, L. (2005). The grammar of graphics (2nd ed.). New York: Springer.
- Williams, D. A. (1987). Generalized linear model diagnostics using the deviance and single case deletions. Applied Statistics, 36, 181–191. Retrieved from  
[http://www.jstor.org/stable/2347550?seq=1#page\\_scan\\_tab\\_contents](http://www.jstor.org/stable/2347550?seq=1#page_scan_tab_contents)
- Wood, S. N. (2006). Generalized additive models: An introduction with R. Boca Raton, FL: CRC/Chapman & Hall.
- Xie, Y. (2015). Dynamic documents with R and knitr (2nd ed.). Boca Raton, FL: CRC/Chapman & Hall.
- Xie, Y. (2018). knitr: A general-purpose package for dynamic report generation in R [Computer software manual]. Retrieved from <http://yihui.name/knitr/> (R package version 1.20)
- Yee, T. W. (2015). Vector generalized linear and additive models: With an implementation in R. New York: Springer.
- Yeo, I.-K., & Johnson, R. A. (2000). A new family of power transformations to improve normality or symmetry. Biometrika, 87 (4), 954–959. Retrieved from  
<http://www.jstor.org/stable/2673623>
- Zeileis, A. (2004). Econometric computing with HC and HAC covariance matrix estimators. Journal of Statistical Software, 11(10), 1–17. Retrieved from  
<http://www.jstatsoft.org/v11/i10/>
- Zeileis, A., Hornik, K., & Murrell, P. (2009). Escaping RGBland: Selecting colors for statistical graphics. Computational Statistics & Data Analysis, 53, 3259–3270. Retrieved from  
<https://www.sciencedirect.com/science/article/pii/S0167947308005549>
- Zeileis, A., Kleiber, C., & Jackman, S. (2008). Regression models for count data

in R. Journal of Statistical Software, 27 (8), 1–25. Retrieved from  
<http://www.jstatsoft.org/v27/i08/>

# Subject Index

- adaptive-kernel density estimate, [37](#), *see also* density estimate
- added-variable plots, [43](#), [385](#), [392–395](#)
  - as influence diagnostic, [403–405](#)
  - for a constructed variable, [434](#)
  - for GLMs, [422](#)
- additive model, [205](#), [219](#)
- adjusted means, [204](#), [221](#)
- aesthetic, in **ggplot2** package, [471](#)
- AIC, [180](#), [281](#), [319](#), [329](#), [352](#)
- Akaike Information Criterion, *see* AIC
- aliased coefficients, [234](#)
- amount, [151](#)
- analysis of covariance, [205](#), [266](#)
- analysis of deviance, [264](#), [285–288](#), [322](#)
  - sequential, [286](#)
  - Type II, [293](#), [299](#), [311](#)
  - Type III, [293](#)
- analysis of variance, [201](#), [214](#), [259](#)
  - empty cells in, [233](#)
  - one-way, [201](#)
  - sequential, [260](#)
  - Type I, [260–262](#), [264](#), [265](#)
  - Type II, [217](#), [262–267](#)
  - Type III, [227](#), [265–267](#)
  - unbalanced, [264–267](#)
- analytical graphics, [437](#)
- anonymous function, [362](#), [370](#), [501](#)
- ANOVA, *see* analysis of variance
- argument, [7](#), [8](#)
  - default, [480](#)
  - ellipses, [484](#)
  - formal, [19](#), [20](#), [480](#)
  - missing, detecting, [535](#)
  - real, [484](#)
- arithmetic operators, [5](#), [478](#), [486–487](#)
  - and missing data, [75](#)
  - extension to vectors, [11](#)

in model formulas, [177](#), [235](#)  
arrays, [97](#)  
arrows, [449](#)  
assignment, [12](#)  
assignment operator, [12](#)  
atomic, [95](#), [98](#)  
attribute, [35](#), [96](#), [200](#)  
autocorrelated errors, in LMM, [372](#)  
awk, [110](#)  
axes, [440](#), [447](#), [452](#), [459](#)  
    logarithmic, [440](#)  
    position of, [447](#)  
B-splines, [195](#), *see also* regression splines  
balanced data, [215](#), [226](#)  
bandwidth, [128](#), [168](#), [457](#)  
bar charts, [143](#)  
baseline level, [200](#)  
    in multinomial logit model, [310](#)  
basic statistical functions, [48](#)  
Bayesian estimation, [xxvii](#)  
Bayesian Information Criterion, *see* BIC  
Bernoulli distribution, [274](#)  
best linear unbiased predictors, [359](#)  
between-subjects factors, [92](#)  
BIC, [180](#), [281](#), [352](#)  
big data, [117](#)  
binary logit model, [294](#)  
binary regression, [276](#)  
binary response, [274](#), [276](#)  
    and binomial response, [293–296](#)  
binding rows and columns, [86](#)  
binomial data, [273](#)  
binomial distribution, [132](#), [167](#)  
binomial regression, [289–296](#)  
    overdispersed, [326](#)  
binomial response, [276](#), [289](#)  
    and binary response, [293–296](#)  
bins, [81](#), [195](#), [198](#)  
Bioconductor Project, [xv](#), [117](#)

bits, [117](#)  
BLUPs, [359](#)  
Bonferroni adjustment, [42](#), [397–399](#), [419](#)  
bootstrap, [42](#), [248–252](#), [325](#), [397](#), [417](#)  
    case resampling, [249](#)  
    confidence interval, [255](#)  
    residual resampling, [249](#)  
boundary bias, [456](#)  
boundary knots, [195](#)  
Box-Cox transformations, *see* transformations, Box-Cox  
Box-Tidwell regression model, [435](#)  
boxplot, [133](#), [202](#), [366](#), [389](#), [421](#), [518](#)  
    and transformation for symmetry, [163](#)  
    in **lattice** package, [468](#), [469](#)  
    marginal, [137](#)  
    parallel, [141](#)  
braces, [497](#)  
break-point, [523](#), [526](#)  
browser mode, [523–526](#)  
bubble plot, [402](#), [483](#)  
bugs, *see* debugging  
bulging rule, [163](#), [164](#), [410](#), [423](#)  
byte, [117](#)  
byte code, [22](#), [50](#), [505](#)  
C, [xiv](#), [499](#), [505](#)  
C++, [xiv](#)  
call stack, [22](#), [523](#)  
camel case, [20](#)  
canonical link function, [274](#), [276](#)  
cascading conditionals, [85](#)  
case-control sampling, [307](#)  
case-resampling bootstrap, [249](#)  
cases, [53](#)  
categorical variables, *see* factors  
cells, [215](#)  
centering data, [340](#), [342](#), [395](#)  
CERES plots, [412](#), [422](#)  
chapter synopses, [xxiv–xxv](#)  
character data, [14](#), [110–117](#)

character expansion, [443](#)  
character string, [14](#), [448](#)  
character vector, [14](#)  
chart junk, [128](#)  
chi-square distribution, [132](#)  
chi-square test for independence, [302](#)  
Cholesky decomposition, [492](#)  
chunk options, [31](#)  
class, [49](#), [55](#), [144](#), [528](#)  
class variables, [197](#), *see also* factors  
classes of objects, [99](#)  
  boot, [250](#), [251](#), [256](#)  
  data.frame, [55](#), [67](#), [437](#)  
  Date, [107](#)  
  eff, [369](#)  
  gg, [471](#)  
  ggmap, [475](#)  
  ggplot, [471](#)  
  glm, [52](#), [331](#)  
  influence.lme, [429](#)  
  lm, [50](#), [52](#), [178](#), [192](#), [251](#), [255](#), [260](#), [375](#), [437](#)  
  lme, [351](#), [375](#)  
  lmList, [344](#), [345](#)  
  POSIXct, [109](#)  
  print.zipmod (), [530](#)  
  summary.zipmod, [532](#)  
  tbl\_df, [67](#)  
  trellis, [365](#), [468](#), [469](#)  
  zipmod, [528](#), [530–532](#), [535](#)  
clipboard, reading data from, [64](#)  
clustered data, [336](#), [429](#)  
code chunks, in R Markdown, [28](#), [30](#), [31](#), [33](#)  
coded scatterplot, [138](#)  
coefficient of variation, [324](#)  
coefficients-as-outcomes model, [349](#)  
coercion, [15](#), [78](#), [97](#), [119](#), [186](#), [199](#)  
collinearity, [429–434](#)  
color, [441](#), [443](#)  
  names of, [453](#)

palette, [453](#)  
    used by the **car** package, [454](#)

primary, [453](#)  
    specification of, [441](#), [452–454](#)

comma-separated values, [59](#), [62](#)

comments, [5](#), [482](#)

common logs, [149](#)

compiler, [xiv](#)

complementary log-log link, [273](#), [278](#)

complete-case analysis, [239](#)

component-plus-residual plots, [44](#), [410–412](#)  
    for GLMs, [422–425](#)  
    for mixed-effects models, [425](#)

compositional variable, [341](#), [342](#), *see also* contextual variable

compound command, [19](#), [36](#), [493](#), [497](#), [498](#)

compound expression, [129](#)

Comprehensive R Archive Network, *see* CRAN

concentration ellipse, [160](#)

conditional independence, [305](#)

conditional mean function, [136](#)

conditionals, [482](#), [492–495](#)  
    cascading, [494](#)

confidence ellipse, [256–258](#)

confidence ellipsoid, [257](#)

confidence envelope  
    for loess, in **ggplot2** package, [470](#)  
    for predictor effect plot, [189](#)  
    for quantile-comparison plot, [131](#)  
    for Studentized residuals in QQ-plot, [397](#)  
    Scheffé, [189](#)

confidence interval, [180](#), [243](#)  
    based on the bootstrap, [251](#)  
    for generalized linear models, [281](#)  
    for logistic-regression coefficients, [280](#)  
    for regression coefficients, [180](#)  
        based on robust standard errors, [248](#)  
    for transformation parameter, [156](#), [407](#), [408](#)  
    Wald, [254](#)

confidence interval generating ellipse, [257](#)

confidence region, [243](#), [256](#)  
for predictor effect plot, [210](#)  
of regression coefficients and collinearity, [432](#)

connections, [63](#)

constant regressor, [183](#)

constructed variable, [434](#)

contextual variable, [340](#), [342](#), *see also* compositional variable

contingency table, [290](#), [295](#), [301](#), [379](#), *see also* loglinear models  
three-way, [304](#), [308](#), [516](#)  
two-dimensional, [301](#)

continuous first-order autoregressive process, [372](#)

contrast matrix, [230](#)

contrasts, [199](#), [224](#), [240](#)  
as used in SAS, [225](#)  
deviation-coded, [226](#)  
dummy-coded, [199](#), [224](#)  
Helmert, [225](#)  
option for, [224](#)  
orthogonal-polynomial, [227](#)  
user-specified, [230](#)

convergence, [512](#)

Cook's distances, [43](#), [399–402](#), [421](#), [429](#), [482](#)

coordinate system, [439–441](#), [459](#)

coordinates, locating interactively, [448](#)

correlations, of coefficients, [245](#)

count data, [296](#)  
and offsets, [332](#)

covariance components, [337](#), [352](#), [353](#)

covariance matrix  
of GLM coefficients, [333](#)  
of predictors, [257](#)  
of regression coefficients, [248](#), [251](#), [257](#)  
of response in LMM, [338](#), [350](#)

covariates, [205](#)

Cox regression, [xxvii](#)

CRAN, [xiii](#), [xv–xvii](#), [xx](#), [10](#), [23](#), [24](#), [107](#), [311](#), [335](#), [467](#), [469](#), [509](#)

cross-tab, [201](#), *see also* contingency table

crossing operator, [213](#), [237](#)

cumulative distribution functions, [132](#)

curve, graphing, [451](#)

data

aggregating, [87](#)

atomic, [95](#)

comma-separated values, [62](#)

dates, [107](#)

exporting, [65–66](#)

from a package, [54](#)

input, [54–69](#)

long, [91](#)

merging, [88, 89](#)

mode of, [96](#)

modifying, [79](#)

recoding, [80](#)

saving, [120](#)

times, [107](#)

transformations, [79](#)

white-space-delimited, [59](#)

wide, [91](#)

data ellipse, [160, 257](#)

data ellipsoid, [146](#)

data file

missing values in, [61](#)

text, [59–63](#)

data frame, [34, 53, 95](#)

data management, [54, 478](#)

dates and times, [107–110](#)

debugging, [21, 522–526](#)

default argument, [480](#)

default method, [52, 528, 533](#)

degrees of freedom, [179](#)

for error, in regression, [179](#)

of Studentized residuals, [387](#)

Satterthwaite, in LMM, [357, 368](#)

delta method, [252–254, 288, 313, 325](#)

density estimate, [37, 41, 128–129, 148, 168, 362](#)

density functions, [132](#)

determinant, [491](#)

deviance, [275, 304](#)

deviation coding, [226](#)  
dfbeta, [402](#), [403](#), [429](#)  
dfbetas, [403](#), [429](#)  
diagonal matrix, [492](#)  
dispersion parameter, [274](#), [275](#), [281](#), [322](#), [323](#), [326](#), [333](#), [375](#), [377](#)  
distribution functions, [132](#)  
distributional family, [375](#)  
dot plot, in **lattice** package, [469](#)  
double-bracket notation, [103](#)  
double-precision number, [117](#)  
dummy argument, *see* formal argument  
dummy regressors, *see* contrasts, dummy-coded  
dummy-variable regression, [205](#)  
Durbin-Watson test, [435](#)  
*e* (mathematical constant), [149](#)  
EDA, *see* exploratory data analysis editing  
    in Console, [7](#)  
    R scripts, [25–28](#)  
effect plots, [218](#), [319](#), [369](#), [382](#), [467](#), [514](#), *see also* predictor effect plots  
eigenvalues and eigenvectors, [491](#)  
elasticity, [153](#)  
ellipse, [160](#), [257](#)  
ellipses argument, [484](#), [528](#)  
ellipsoid, [146](#), [432](#)  
empirical logits, [168](#)  
empty string, [113](#)  
environment, [13](#), [20](#), [70](#), [86](#), [512](#), *see also* global environment  
equality, testing, [508](#)  
error, [153](#), [175](#)  
    multiplicative, [153](#)  
error bars, [142](#), [218](#), [476](#)  
error message, [482](#), [497](#)  
error variance, [175](#), [414](#)  
errors, in code, *see* debugging  
escape character, [63](#)  
estimated marginal means, [203](#)  
example functions  
    abs1, [493](#)  
    combineFormulas, [533](#), [535](#)

convert2meters, [494](#), [495](#)  
fact, [498](#)  
fact1, [495](#), [496](#)  
fact2, [496](#), [497](#)  
fact3, [497](#)  
fact4, [498](#)  
fact5, [498](#)  
inflPlot, [483–485](#)  
Lag, [480](#), [481](#)  
makeScale, [501](#)  
mean, [19](#)  
myMean, [19–23](#)  
mySD, [21–23](#)  
negLogL, [512](#), [523](#), [524](#), [526](#)  
print.summary.zipmod, [532](#), [533](#)  
print.zipmod, [531](#), [533](#), [537](#)  
pvalCorrected, [371](#)  
sign1, [494](#)  
sign2, [494](#)  
summary.zipmod, [533](#)  
tc, [460](#)  
time1, [506](#)  
time2, [506](#)  
time3, [506](#), [507](#)  
time4, [507](#)  
time5, [507](#)  
tricube, [460](#)  
vcov.zipmod, [537](#)  
zipmod, [510–513](#), [524](#), [526](#), [528](#), [529](#), [533–535](#), [537](#)  
zipmod.default, [528](#), [529](#), [533](#), [535](#), [537](#)  
zipmod.formula, [533–535](#), [537](#)  
zipmodBugged, [523–527](#)  
examples, in help pages, [10](#)  
Excel, [66](#)  
explanatory variables, [174](#)  
exploratory data analysis, [123](#), [162–166](#)  
exponential families, [272](#), [276](#), [375](#)  
exponential function, [149](#)  
exponentiated coefficients

for logistic regression, [280](#), [281](#), [381](#)  
for multinomial logistic regression, [313](#)  
for Poisson regression, [299](#)

expression, [440](#)  
    in graphs, [150](#)

*F*-distribution, [132](#)

factors, [35](#), [58](#), [95](#), [174](#), [197–207](#), [224–232](#)  
    baseline level for, [225](#)  
    coding, [224–231](#)  
    contrasts for, [224](#)  
    dummy coding, [224](#)  
    ordered, [224](#), [227](#)  
    polynomial contrasts for, [227](#)  
    releveling, [225](#)  
    unordered, [224](#)  
    user-defined contrasts for, [230](#)  
    vs. character variables, [198](#)  
    vs. logical variables, [198](#)

family generator function, [276](#), [277](#)

fields, [59](#)

files  
    comma-separated values, [62](#)  
    Excel, [66](#)  
    other input types, [66](#), [67](#)  
    text, [59–63](#)  
    white-space-delimited, [59–62](#)  
        writing, [65](#)

fitted odds, [281](#)

fitted values, [184](#), [289](#), [387](#), [389](#), [390](#)

fixed effects, [337](#), [338](#), [349–352](#), [355](#), [356](#), [367](#), [376](#), [380](#), [429](#)  
    likelihood-ratio test for, [356](#), [357](#)

fixed-bandwidth local regression, [457](#)

floating-point number, [117](#), [482](#)

formal argument, [19](#), [20](#), [480](#)

formula, [39](#), [143](#), [177–178](#), [184](#), [202](#), [203](#), [207](#), [214](#), [235–238](#), [276](#), [350](#),  
[356](#), [505](#), [517](#), [533](#)  
    colon in, [207](#)  
    for boxplot, [141](#)  
    for graph, [144](#)

for lattice graphics, [467](#)  
for linear model, [177](#)  
for scatterplot, [136](#), [138](#)  
for scatterplot matrix, [147](#)  
one-sided, [37](#), [130](#), [134](#), [147](#), [200](#), [350](#), [362](#), [391](#), [535](#)  
shortcuts, [213](#)

formula method, [533](#)

Fortran, [xiv](#), [499](#), [505](#)

full quadratic model, [222](#)

function, [7](#)

- anonymous, [362](#), [370](#), [501](#)
- creating, from a script, [483](#)
- defining, [479](#)
- graphing, [451](#)
- local, [512](#), [533](#)
- primitive, [20](#)
- recursive, [498](#)
- user-defined, [18–21](#)
- value returned by, [482](#)
  - invisible, [483](#)

gamma distribution, [323](#), [328](#)

gamma regression, [323](#), [417](#)

garbage collection, [120](#)

Gauss, [xiv](#)

general linear hypothesis, *see* hypothesis test, general linear

general mean, [230](#)

generalized distance, *see* Mahalanobis distance

generalized inverse, [492](#)

generalized linear mixed models, [335](#), [375](#)

- matrix form of, [376](#)

generalized linear models, [271–333](#), *see also* gamma regression; logistic regression; Poisson regression

- diagnostics for, [417](#)
- distributional family, [272](#), [274](#)
- fitted values, [289](#)
- inverse-link function, [273](#)
- likelihood-ratio tests, [286](#)
- linear predictor, [273](#)
- link function, [273](#)

quasi-Poisson, [328](#)  
random component, [272](#)  
residuals, [418–419](#)  
structure of, [272–274](#)  
generalized variance inflation factor, [432, 434](#)  
generic function, [9, 14, 19, 35, 41, 49, 127, 135, 144, 260, 388, 401, 439, 520, 528, 530, 533](#)  
geometric mean, [153](#)  
geometry, in **ggplot2** package, [470](#)  
gets, [12](#)  
GGobi, [146, 476](#)  
gigabyte, [117](#)  
GLMMs, *see* generalized linear mixed models  
GLMs, *see* generalized linear models  
global environment, [13, 20, 70, 86, 117](#), *see also* environment  
global variables, [20](#)  
Google Maps, [473](#)  
gradient, [512](#)  
grammar of graphics, [469](#)  
graphical user interfaces, [xiv](#)  
graphics  
    interactive, [476](#)  
    object-oriented, [467](#)  
graphics parameters, [145, 150, 166, 346, 441–442](#)  
graphs, statistical, [123](#)  
gray levels, [452](#)  
grep, [110](#)  
grid, [447](#)  
grouped-data object, [339](#)  
GTK+, [476](#)  
GUIs, [xiv](#)  
GVIF, [432](#)  
hat-matrix, [247, 399](#)  
hat-value, *see* leverage  
HCL color space, [452](#)  
header, [30](#)  
Helmert contrasts, [225](#)  
help, [8, 23](#)  
Hessian, [318, 512](#)

heteroscedasticity, *see* nonconstant variance  
hexadecimal numbers, for color specification, [453](#)  
hierarchical data, [335](#), [337](#), [339](#)  
hinges, [165](#)  
histogram, [36](#), [124–128](#), [216](#), [469](#)  
history, of commands, [7](#), [25](#)  
HSD method, of multiple comparisons, [204](#)  
HSV color space, [452](#)  
hypothesis test, [243](#)  
    by simulation, [520](#)  
    chi-square, for independence, [302](#), [304](#)  
    deviance-based, [275](#)  
    for coefficients in an LMM, [351](#), [357](#)  
    for contrasts, [231](#)  
    for fixed effects, [368](#)  
    for lack of fit, [304](#), [305](#), [389](#), [420](#)  
    for logistic-regression coefficients, [281](#), [285–288](#)  
    for nonadditivity, [390](#)  
    for nonconstant variance, [416–417](#)  
    for nonlinearity, [412](#)  
    for outliers in regression, [398](#), [419](#)  
    for proportional odds, [319–322](#)  
    for random effects, [370](#)  
    for regression coefficients, [179](#), [180](#), [248](#)  
    for transformation parameter, [156–158](#), [407](#), [408](#)  
    for variance and covariance components, [353](#)  
general linear, [267–270](#)  
in unbalanced analysis of variance, [264](#)  
incremental, [259](#)  
likelihood-ratio, *see* likelihood-ratio test  
Type I, [286](#)  
Type II, [287](#), [305](#), [322](#)  
Type III, [288](#), [323](#)  
Wald, [179](#), [254](#), [258](#), [260](#), [267](#), [281](#), [287](#), [288](#), [322](#), [357](#), [368](#), [381](#)  
    adjusted for overdispersion, [327](#)  
    degrees of freedom for, in LMM, [356](#), [357](#)  
    for logistic regression, [286](#)  
ID variable, [91](#)  
IDE, [xviii](#)

identity function, [236](#)  
identity matrix, [492](#)  
in-line code, in R Markdown, [31](#)  
indentation of code, [494](#), [498](#)  
index plot, [144](#), [399](#), [403](#), [421](#), [443](#)  
indexing, [99](#)  
    and assignment, [100](#), [102](#), [104](#)  
    arrays, [100](#)  
    data frames, [70](#), [105](#)  
    lists, [102](#)  
    matrices, [100](#)  
    vectors, [16](#), [99](#)  
individual-level variable, [340](#)  
infinity, [150](#)  
influence, [43](#), [45](#), [395](#), [399–405](#), [434](#)  
    in GLMs, [421–422](#)  
    in mixed-effects models, [428–429](#)  
    joint, [43](#), [403](#)  
influence plot, [482–484](#)  
inheritance, [52](#), *see also* generic function; method function; object-oriented programming  
inhibit function, [236](#)  
initialization, [506](#)  
ink-on-paper metaphor, [438](#), [467](#)  
inner fences, [133](#)  
inner variable, [340](#)  
inner-product operator, [487](#)  
integer, [482](#)  
integration, numeric, [503](#), [504](#)  
interaction, [174](#), [207](#), [262](#), [421](#)  
    in loglinear models, [304](#), [305](#), [310](#)  
    linear by linear, [222](#), [425](#)  
    of numeric predictors, [222–224](#)  
    partial residuals for, [412](#)  
interactive development environment, [xviii](#)  
interactive graphics, [476](#)  
intercept, [174](#), [178](#), [238](#)  
    suppressing, [231](#), [238](#)  
internet address, [xxvii](#), *see also* websites

interpreter, [xiv](#)  
inverse response plot, [406](#), [435](#)  
inverse transformation, [406](#)  
inverse-Gaussian distribution, [323](#)  
inverse-link function, [273](#), [274](#)  
invisible result, [533](#)  
iterated weighted least squares, *see* IWLS  
iteration, [495–498](#), [505–509](#), *see also* loops  
IWLS, [332](#), [418](#)  
Java, [476](#)  
jittering, [139](#)  
joint confidence region, [257](#)  
joint influence, [43](#), [422](#)  
*Journal of Statistical Software*, [xv](#)  
Julia, [xiv](#)  
justification of text, [443](#), [448](#)  
kernel density estimate, [128](#), *see also* density estimate  
kernel function, [128](#), [457](#)  
kernel mean function, [273](#)  
kernel regression, [456](#)  
key, *see* legend  
keyboard data input, [56–58](#)  
keyboard equivalents, in RStudio, [28](#)  
knots, [195](#)  
lack of fit, [304](#), [305](#), [389](#), [420](#)  
ladder of powers and roots, [162](#), [411](#), [423](#)  
lag, [479](#)  
latent response variable, [317](#)  
Latex, installing, [xxiii](#)  
lattice graphics, [189](#), [343](#), [365](#), [467–469](#), [517](#), *see also* lattice package  
    modifying, [468](#)  
lazy data, [55](#)  
lazy evaluation, [497](#)  
lead, [480](#)  
leakage, [412](#)  
legend, [450](#), [452](#)  
    in lattice graphs, [468](#)  
length, [97](#)  
letters, [99](#)

level, [164](#)  
levels, [35](#), [174](#), [197](#), [199](#), *see also* factors  
Levene's test, [435](#)  
leverage, [43](#), [136](#), [182](#), [387](#), [394](#), [395](#), [398–399](#), [401](#), [402](#), [405](#), [434](#), [482](#)  
    for GLMs, [419](#), [421](#)  
leverage plots, [395](#)  
library, [xxii](#)  
likelihood-ratio test, [156](#), [157](#), [179](#), [296](#), [304](#), [322](#), [323](#), [353](#), [356](#), [370](#), [407](#),  
[408](#)  
    for GLMs, [286](#)  
    for logistic regression, [286](#)  
    for variance and covariance components, [354](#), [371](#)  
line segments, [449](#)  
line type, [441](#), [443–445](#)  
line width, [441](#), [443](#), [445](#)  
linear algebra, *see* matrix algebra  
linear hypothesis, [288](#)  
linear mixed-effects model, [337](#)  
    hierarchical form of, [348](#)  
    matrix form of, [338](#)  
linear model, [173](#), [174](#), [336](#)  
    assumptions of, [174–175](#)  
    matrix form of, [336](#)  
    normal, [176](#)  
    weighted, [176](#), [239](#)  
linear predictor, [175](#), [274](#), [375](#), [376](#), [420](#)  
linear regression  
    computing, [489–491](#)  
    multiple, [183–184](#)  
    simple, [176–183](#)  
linear simultaneous equations, [489](#)  
linear-by-linear interaction, [282](#), [283](#), [425](#)  
linear-model object, [39](#), [178](#)  
lines, [443](#), [446](#)  
link function, [273](#), [274](#), [276](#), [277](#), [375](#), [376](#), [409](#)  
    canonical, [274](#)  
Linux, [xvii](#), [xviii](#)  
Lisp-Stat, [xiv](#)  
lists, [98–99](#)

LMM, *see* linear mixed-effects model  
loadings, [491](#)  
local function, [512](#), [533](#)  
local linear regression, [77](#), [455–461](#)  
local polynomial regression, [77](#)  
local regression, fixed-bandwidth, [457](#)  
local variables, [20](#), [512](#)  
loess, [44](#), [139](#), [171](#), [390](#), [391](#), [410](#), [413](#), [414](#), [425](#), [455](#), [468](#), [470](#)  
log rule, [151](#), [154](#), [423](#)  
log transformation, [183](#), [191](#)  
log-odds, [167](#), [278](#), [281](#)  
logarithms, [7](#), [148](#), [406](#), [423](#)  
    as zeroth power, [155](#)  
    axes in graphs, [440](#)  
    common, [149](#)  
    natural, [149](#)  
    review of, [7](#)  
logical data, [15](#)  
logical operators, [17](#), [478](#), [497](#)  
logistic distribution, [318](#)  
logistic regression, [167](#), [271](#), [276](#), [278–296](#), [423](#), [425](#)  
    and loglinear models, [307–309](#)  
    and residual plots, [420](#)  
    binary, [294](#), [295](#)  
    binomial, [295](#), [333](#)  
    mixed-effects model, [380](#)  
    multinomial, [309](#)  
    nested dichotomies, [314](#)  
    proportional-odds, [317](#)  
logit, [166](#), [167](#), [278](#)  
logit link, [273](#), [276](#), [380](#)  
logit model, *see* logistic regression  
loglinear models, [167](#), [296](#), [301–309](#)  
    and logistic regression, [307–309](#)  
    for three-way tables, [304](#)  
    for two-way tables, [301](#)  
    sampling plans for, [306–307](#)  
    with response variables, [307](#)  
logs, *see* logarithms

long data, [91](#)  
longitudinal data, [91](#), [335](#), [337](#), [360](#)  
loop variable, [497](#)  
loops, [495–498](#), [505–509](#)  
    avoiding, [499](#), [505](#)  
    breaking out of, [498](#)  
    rules for, [506–509](#)  
lower-order relatives, [214](#)  
lowess, [77](#)  
macOS, [xvii–xxi](#), [xxiii](#), [5](#), [28](#), [64](#), [448](#)  
MacTeX, [xxiii](#)  
Mahalanobis distance, [44](#), [137](#), [146](#), [152](#), [171](#), [404](#)  
main diagonal, [492](#)  
main effect, [263](#)  
maps, drawing, [472–475](#)  
marginal-conditional plots, [395–396](#)  
marginal-model plots, [391–392](#)  
marginality, principle of, [214](#), [305](#), [307](#)  
margins, [166](#), [443](#), [463](#)  
    text in, [449](#)  
Markdown, [25](#), [28–32](#), *see also* R Markdown  
Markov Chain Monte Carlo, [515](#)  
masking objects, [19](#), [70](#), [72](#), [86](#)  
match-merging, *see* merging data sets  
matching operator, [85](#)  
mathematical notation, in graphs, [150](#), [440](#), [459](#)  
matrices, [96–97](#)  
matrix algebra, [486–492](#)  
maximum likelihood, [274](#), [406](#), [510](#)  
MCMC, [515](#)  
mean function, [175](#), [273](#)  
mean-shift outlier model, [397](#)  
means and standard deviations, table of, [201](#), [215](#)  
megabyte, [118](#)  
memory usage, [118](#)  
menus and menu items  
    Cheatsheets, [29](#)  
    Code, [27](#), [33](#), [483](#)  
    Debug, [523](#)

Debugging Help, [523](#)  
Edit, [26](#)  
Exit, [xxvi](#)  
Extract Function, [483](#)  
File, [xxvi](#), [2](#), [26](#), [29](#), [30](#)  
Find, [26](#)  
General, [5](#)  
Global Options, [xx](#), [xxi](#), [5](#), [28](#)  
Global Profile, [26](#)  
Help, [29](#)  
Import Dataset, [68](#)  
Install, [xxi](#)  
Markdown Quick Reference, [29](#)  
New File, [26](#), [29](#), [30](#)  
New Project, [2](#)  
Open File, [26](#)  
R Markdown, [29](#), [30](#)  
R Script, [26](#)  
Recent Files, [26](#)  
Run Current Chunk, [33](#)  
Run Selected Line (s), [27](#), [33](#)  
Save, [26](#)  
Save as, [26](#)  
Session, [63](#)  
Set Working Directory, [63](#)  
Stop, [448](#)  
Tools, [xx](#), [xxi](#), [5](#), [26](#), [28](#)  
merging data sets, [89](#), [341](#), [347](#)  
meta-characters, [113](#)  
method function, [35](#), [50](#), [127](#), [528](#), [530](#), [533](#), [535](#)  
methods, *see* method function; generic function  
MiKTeX, [xxiii](#)  
missing data, [60](#), [72–78](#), [174](#), [191](#), [239–240](#), [260](#), [457](#), [500](#), [535](#)  
    complete cases, [76](#), [77](#)  
    filtering, [77](#)  
    in statistical-modeling functions, [76](#)  
    multiple imputation of, [xxvii](#), [72](#), [174](#), [378](#)  
    pattern of, [73](#)  
    testing for, [78](#)

mixed-effects models, [335](#), *see also* linear mixed-effects model; generalized-linear mixed models  
diagnostics for, [425](#)  
mode, [95](#), [96](#)  
model formula, *see* formula  
model frame, [535](#)  
model matrix, [200](#)  
Monte-Carlo simulation, *see* simulation  
multinomial distribution, [307](#)  
multinomial logit model, [309–314](#), [319](#)  
multinomial response, [309](#), [310](#)  
multinomial sampling, [307](#)  
multinomial-regression model, [310](#)  
multiple comparisons, [204](#), [219](#), *see also* simultaneous inference  
multiple correlation, [179](#)  
multiple regression, *see* linear regression, multiple  
multiplicative error, [153](#)  
multivariate data, plotting, [145–148](#)  
multivariate linear models, [xxvii](#)  
multiway ANOVA, [216](#)  
name clash, [70](#), [72](#)  
names, rules for, [12](#), [19](#)  
namespace, [51](#), [70](#)  
natural logs, [7](#), [149](#)  
natural splines, [195](#), *see also* regression splines  
negation, [15](#)  
negative-binomial regression, [328–330](#)  
nested dichotomies, [314–316](#), [319](#)  
nesting operator, [236](#)  
nesting, of models, [260](#)  
new-line character, [459](#), [533](#)  
nonadditivity test, [390](#)  
nonconstant variance, [45](#), [246](#), [247](#), [388](#), [392](#), [405](#), [414–417](#), [435](#)  
nonlinear mixed model, [335](#)  
nonlinear regression, [xxvii](#)  
nonlinearity, [388](#), [389](#), [391](#), [392](#), [395](#), [405](#), [407](#), [410](#), [412](#), [413](#), [423](#), [463](#)  
nonparametric density estimation, *see* density estimate  
nonparametric regression, [xxvii](#), [38](#), [77](#), [136](#), [138](#), [146](#), [148](#), [171](#), [194](#), [195](#), [410](#), [412](#), [455](#), *see also* loess; lowess; scatterplot smoother

normal distribution, [132](#)  
not a number, [150](#)  
not operator, [15](#)  
null plot, [388](#)  
object, [49](#)  
object-oriented programming, [20](#), [49](#), [99](#), [127](#), [144](#), [527–533](#)  
objective function, [509](#)  
observations, [53](#)  
odds, [278](#), [281](#)  
odds ratio, [281](#)  
offset, [240](#), [331](#)  
OLS, *see* ordinary least squares  
one-dimensional scatterplot, [189](#), *see also* rug-plot  
one-sided formula, *see* formula, one-sided  
OpenGL, [145](#), [172](#), [476](#)  
optimization, [509](#)  
ordered probit model, [318](#)  
ordinal response, [317](#)  
ordinal variable, [197](#)  
ordinary least squares, [176](#), [386](#)  
ordinary residuals, [418](#)  
orthogonal-polynomial regressors, [193](#), [228](#)  
outer product, [488](#)  
outer variable, [340](#)  
outliers, [133](#), [136](#), [181](#), [182](#), [397–399](#), [401](#), [405](#), [461](#)  
overdispersion, [326–330](#)  
overplotting, [140](#)  
packages, [xxii](#)

- contributed, [xv](#)
- installing, [xx](#)
- news for, [xiii](#)
- updating, [xxii](#)
- versus library, [xxii](#)

  
pairwise comparisons, [203](#), *see also* simultaneous inference  
palette, of colors, [453](#)  
panel function, [362](#), [469](#)  
panel, in lattice graphics, [467](#)  
parallel boxplots, [141](#)  
parentheses, use of, [6](#)

partial fit, [410](#)  
partial regression function, [190](#)  
partial residuals, *see* residuals, partial  
partial slopes, [184](#)  
partial-regression plots, *see* added-variable plots  
partial-residual plots, *see* component-plus-residual plots  
paths, file, [63](#)  
Pearson statistic, [333](#)  
Pearson's chi-square, [302](#), [323](#), [326](#)  
percentage table, [303](#), [379](#), [517](#)  
periods, in object names, [20](#)  
PERL, [110](#)  
permutation, random, [96](#)  
pipe operator, [88](#)  
plot layout, [442](#), [443](#), [457](#), [461–466](#)  
plot margins, [166](#)  
plotting character, [443](#)  
plotting symbol, [443](#)  
point identification, [44](#), [132](#), [141](#), [146](#), [148](#), [152](#), [169–171](#), [390](#), [404](#)  
points, [443](#)  
Poisson distribution, [132](#), [328](#), [512](#)  
Poisson regression, [296–301](#), [408](#)  
    overdispersed, [326](#)  
Poisson sampling, [307](#)  
polygon, [449](#)  
polynomial regression, [154](#), [190–194](#), [212](#), [410](#)  
polynomials, [228](#), [229](#)  
polytomous response, [271](#)  
population marginal mean, [265](#)  
precedence of operators, [6](#)  
predicted values, [184](#), [289](#)  
predictor effect plots, [187–190](#), [205](#), [209](#), [223](#), [299](#), [306](#), [313](#), [352](#), [425](#), *see also* effect plots  
    for logistic regression, [283–285](#)  
predictors, [174](#)  
presentation graphics, [123](#), [437](#)  
principal-components analysis, [434](#), [491](#)  
principle of marginality, [214](#), [263](#), [310](#)  
prior weights, [331](#)

probability functions, [132](#)  
probability-mass function, [132](#)  
probit link, [273](#), [278](#)  
product-multinomial sampling, [307](#)  
program control, [492–499](#)  
projects, *see* RStudio, projects  
prompt, [5](#)  
proportional odds, testing for, [319–322](#)  
proportional-odds model, [317–319](#)  
protected expressions, [236](#)  
pseudo-random numbers, *see* random numbers  
Python, [xiv](#)  
QQ-plot, *see* quantile-comparison plot  
QR decomposition, [491](#), [492](#)  
quadratic model, [222](#)  
quadratic polynomial, [191](#)  
quadratic regression, [411](#)  
quadratic relationship, [410](#)  
qualitative variables, *see* factors  
quantile functions, [132](#)  
quantile regression, [171](#)  
quantile-comparison plot, [41](#), [130](#), [397](#)  
quasi-likelihood estimation, [325](#), [375](#)  
quasi-Poisson model, [328](#)  
quotes, use of, [12](#)  
R  
  advantages of, [xiv](#)  
  basics, [5–21](#)  
  command prompt, [7](#)  
  commands, continuation of, [26](#)  
  console, [5](#)  
  customizing, [xxii](#)  
  email lists, [xxiii](#)  
  help, [23](#)  
  installing, [xvi–xviii](#)  
  interpreter, [5](#)  
  manuals, [xxiii](#)  
  origins of, [xiv](#)  
  profile, [xxii](#)

script, 25  
R Commander, xiv  
*R Journal*, xv  
R Markdown, xx, xxiii, xxvii, 1, 3, 7, 25, 28–34, 36, 37, 56, 64, 80, 112, 170, 290, 475, 478, 537  
R.app, xviii  
ragged array, 504  
random effects, 328, 336–338, 349, 350, 356, 367, 375–377, 380, 429  
    BLUPs for, 359  
    crossed, 356  
    nested, 356  
random intercept, 339  
random numbers, 13, 132, 515  
random permutuation, 96  
random-number generators, 515  
rate, 332  
recoding, 80  
recursion, 498–499  
recursive data structure, 98  
recycling of elements, 12, 96  
reference classes, 49  
reference level, 201  
regression constant, 174  
regression diagnostics, 40, 385  
regression function, 136  
regression splines, 190, 194–197, 212, 410  
regression tree, 437  
regressors, 174, 273  
    interaction, 209  
regular expressions, 23, 110, 112  
relational operators, 16, 478  
REML, 350, 370  
    versus ML, 356, 370  
removing cases, 45  
repeated-measures analysis of variance, xxvii  
reproducible research, xxvii, 1, 13, 25, 80, 521  
reserved symbols, 15  
reshaping data, 91–95  
residual deviance, 275, 304

residual sum of squares, [386](#), [388](#)  
residuals, [386–388](#)

- deviance, [419](#), [420](#)
- for generalized linear models, [418–419](#)
- ordinary, [386](#), [418](#)
- partial, [410](#), *see also* component-plus-residual plots
- for effect plots, [412–414](#), [425](#), [427](#)
- for mixed-effect models, [425–427](#)
- Pearson, [387](#), [388](#), [414](#), [418](#), [420](#)
- plots of, [388–391](#)
- response, in GLMs, [418](#)
- standardized, [387](#), [401](#)
- standardized deviance, [419](#)
- standardized Pearson, [419](#)
- Studentized, [41](#), [387](#), [397–399](#), [401](#), [402](#), [419](#), [482](#)
- approximate, in GLMs, [419](#)
- working, [418](#)

response variable, [174](#)  
restricted maximum likelihood, *see* REML  
retrospective sampling, [307](#)  
RGB color space, [452](#), [453](#)  
right-assign operator, [89](#)  
risk factors, [281](#)  
robust regression, [xxvii](#), [398](#)  
robust standard errors, *see* sandwich estimator  
Rogers-Wilkinson notation, [177](#)  
rotating scatterplot, [145](#)  
row names, [34](#), [55](#), [59](#)  
RStudio, [xiii](#), [xvi–xxiv](#), [xxvi](#), [xxvii](#), [xxix](#), [1–5](#), [7](#), [8](#), [13](#), [18](#), [20](#), [23–30](#),  
[32–34](#), [36](#), [37](#), [58–60](#), [63](#), [67](#), [68](#), [70](#), [79](#), [111](#), [112](#), [122](#), [170](#), [176](#), [442](#), [448](#),  
[475](#), [483](#), [494](#), [523](#), [526](#), [527](#), [537](#)

- customization of, [26](#)
- installing, [xviii–xx](#)
- interface, [xix](#)
- projects, [1–5](#), [63](#)
- rug-plot, [37](#), [130](#), [189](#), [300](#)

S, [xiv](#), [xv](#), [xxviii](#), [49–51](#), [76](#), [241](#), [437](#), [467](#), [528](#), [529](#), [537](#)  
S-PLUS, [xiv](#)  
S3, [xxviii](#), [49](#), [52](#), [99](#), [527–533](#)

S4, [xxviii](#), [49](#)  
sampling variance, [243](#)  
sandwich estimator, [246–248](#), [264](#), [269](#), [417](#)  
    for clustered data, [373–375](#)  
SAS, [xiv](#), [xxvi](#), [61](#), [66](#), [67](#), [79](#), [163](#), [177](#), [197](#), [225](#), [261](#), [339](#), [383](#)  
saturated model, [275](#), [291](#), [296](#), [304](#)  
saving objects, [99](#)  
scalar, [486](#)  
scale function, for lattice graph, [469](#)  
scale parameter, *see* dispersion parameter  
scaled deviance, [275](#)  
scaled power transformations, [155](#)  
scatterplot, [134–141](#), [395](#)  
    coded, [138](#)  
    enhanced, [136](#)  
    in **ggplot2** package, [469](#)  
    in **lattice** package, [469](#)  
    jittered, [139](#)  
    marked by group, [138](#)  
    three-dimensional, [145](#), [476](#)  
scatterplot matrix, [146](#), [183](#), [410](#)  
scatterplot smoother, [38](#), [77](#), [136](#), [139](#), [171](#), [390](#), [391](#), [413](#), [414](#), [422](#), [425](#),  
[455](#)  
scientific notation, [10](#), [40](#)  
score test, [304](#), [416](#)  
search path, [69](#)  
sed, [110](#)  
seed, for random-number generator, [521](#)  
selector variable, [520](#)  
semicolon, as command separator, [36](#)  
sequence function, [11](#)  
sequence operator, [11](#)  
shadowing objects, [19](#), [70](#), [72](#), [86](#)  
shiny interactive graphics, [475](#)  
side effect, [36](#), [49](#), [483](#)  
sigma constraints, [226](#)  
simple linear regression, *see* linear regression, simple  
simulation, [13](#), [398](#), [478](#), [515–522](#)  
    to test a regression model, [516–522](#)

simultaneous inference, [143](#), [189](#), [204](#), [206](#), [219](#), [397](#)  
singular-value decomposition, [434](#), [492](#)  
singularity, [232–235](#), [240](#)  
small multiples, [467](#)  
snake case, [20](#)  
source code, [xiv](#)  
sourcing code, [537](#)  
spaces, use of, in expressions, [6](#)  
span, [77](#), [139](#), [171](#), [413](#), [425](#), [456](#)  
special form, [480](#)  
splines, *see* regression splines  
spread, [164](#)  
spread-level plot, [164](#), [435](#)  
    application to regression, [435](#)  
SPSS, [xiv](#), [66](#), [79](#)  
stack, [22](#)  
stacked area graph, [313](#)  
StackOverflow, [xxiii](#), [25](#)  
standard error, [243](#)  
    and collinearity, [430](#)  
    Kenward and Roger, for LMM, [357](#), [368](#)  
    of adjusted means, [204](#), [206](#)  
    of regression coefficients, [179](#), [244–254](#)  
        based on the bootstrap, [249](#)  
        robust, [246](#)  
        sandwich estimator, [246](#)  
standardization, [186](#)  
standardized regression coefficients, [185–187](#)  
standardized residuals, *see* residuals, standardized  
start values, [331](#), [512](#)  
start, for power transformation, [161](#), [407](#)  
Stata, [xiv](#), [66](#)  
statistical computing environment, [xiv](#)  
statistical graphs, [437](#)  
statistical model, [174](#)  
statistical package, [xiv](#)  
statistical-modeling functions, writing, [533–536](#)  
stem-and-leaf display, [128](#)  
strip function, for lattice graph, [469](#)

structural dimension, [136](#)  
Studentized residuals, *see* residuals, Studentized sum-to-zero constraints, [226](#)  
summary graph, [135](#)  
surveys, complex, [xxvii](#)  
survival analysis, [xxvii](#)  
Sweave, [xxix](#)  
*t*-distribution, [132](#)  
target model, [363](#)  
task views, [24](#)  
term, [192](#)  
text mining, [122](#)  
text, plotting, [448](#)  
three-dimensional plots, [145](#), [469](#)  
tibble, [67](#)  
tick labels, [447](#)  
tick marks, [447](#)  
tidyverse, [53](#), [67](#), [68](#), [72](#), [88](#), [107](#)  
time data, [107](#)  
time series, [479](#)  
time-series regression, [xxvii](#)  
timing computations, [118](#), [506](#), [507](#)  
tracing, [499](#)  
transformations, [148–169](#), [405](#)

- arcsine square root, [167](#)
- axes for, [166](#)
- Box-Cox, [154](#), [155](#), [157](#), [406–407](#)
  - Box-Cox, with zero or negative values, [161](#), [407–409](#)
  - by group, [158](#)
  - for constant variability, [154](#), [164](#)
  - for linearity, [154](#), [157](#), [163](#)
  - for symmetry, [154](#), [162](#)
  - in exploratory data analysis, [162](#)
  - inverse, [155](#)
  - log, [148](#), [155](#)
  - log-odds, [167](#)
  - logit, [166](#), [167](#)
  - multivariate, [157](#)
  - of percentages, [167](#)

of predictors, [410–414](#), [435](#)  
of proportions, [167](#)  
of restricted-range variables, [167](#)  
power, [154](#)  
reciprocal, [155](#)  
response, [406–409](#), [435](#)  
role of, in regression, [154](#)  
scaled power, [155](#)  
simple power, [154](#)  
toward normality, [154](#), [157](#)  
understanding, [409](#)  
variance-stabilizing, [167](#)  
with zero or negative values, [161](#), [362](#), [435](#)  
Yeo-Johnson, [435](#)

transparency, [453](#)

transpose, [488](#)

trellis graphics, *see* lattice graphics

tricube kernel function, [457](#), [460](#)

Tukey's test for nonadditivity, [390](#)

tuning parameter, [456](#)

two-way ANOVA, [216](#)

typographical conventions, [xxv–xxvi](#)

underdispersion, [326](#)

uniform distribution, [132](#)

Unix, [xvii](#), [xviii](#), [110](#), [115](#)

unmodeled heterogeneity, [326](#)

unusual data, [388](#), [396](#)

URL, [63](#), *see also* websites

user coordinates, [459](#)

variable, [12](#)  
    local, [512](#)

variable names, in data file, [59](#)

variables, in data set, [53](#)

variance components, [336](#), [337](#), [352](#), [353](#)

variance function, [136](#), [137](#), [274](#), [275](#)

variance inflation factor, [431](#), [434](#)

variance-stabilizing transformation, [167](#)

vectorization, [16](#), [493](#), [499](#), [503–505](#), [507](#)

vectors, [11](#), [95](#), [487](#)

VIF, *see* variance inflation factor  
vignettes, [24](#)  
Wald test, *see* hypothesis test, Wald  
warnings, [23](#)  
websites, [xxvii](#)  
    cran.r-project.org, [xvii](#), [24](#), [467](#)  
    en.wikipedia.org, [122](#), [172](#)  
    miktex.org, [xxiii](#)  
    socialsciences.mcmaster.ca/jfox/Books/Companion/index.html, [xxvii](#)  
    stackoverflow.com, [xxiii](#)  
    tinyurl.com/rcompanion, [xxvii](#)  
    www.bioconductor.org, [117](#)  
    www.opengl.org, [172](#), [476](#)  
    www.r-project.org, [xxiii](#), [25](#), [534](#)  
    www.rstudio.com, [xviii](#)  
    www.sagepub.com, [xxvii](#)  
    www.tug.org, [xxiii](#)  
    www.xquartz.org, [xviii](#)  
    yihui.name/knitr/options/, [31](#)  
        searching, [24](#)  
weighted least squares, [176](#), [239](#), [331](#), [387](#), [417](#), [455](#)  
white space, [59](#)  
white-space-delimited values, [59](#)  
wide data, [91](#)  
Windows, [xvii–xix](#), [xxii](#), [xxiii](#), [xxvi](#), [5](#), [27](#), [63](#), [64](#), [118](#), [448](#)  
within-subjects factors, [94](#)  
WLS, *see* weighted least squares  
workflow, [25](#)  
working directory, [3](#), [63](#)  
working residuals, [332](#)  
working response, [332](#)  
working weights, [331](#), [333](#)  
workspace, [13](#), [117](#)  
zero-inflated data, [362](#)  
zero-inflated Poisson model, [509–514](#), [523–536](#)  
ZIP model, *see* zero-inflated Poisson model

# Data Set Index

- Adler, [240](#)  
AMSSurvey, [301](#), [304](#), [306](#), [307](#)  
Baumann, [201](#), [230](#), [240](#)  
Blackmore, [360](#), [366](#), [367](#), [369](#), [371](#), [372](#), [426](#), [428–431](#)  
Campbell, [290](#), [294](#)  
Challeng, [55](#)  
Cowles, [282](#), [287](#), [289](#)  
Davis, [54](#), [55](#), [176](#), [238](#), [252](#), [259](#), [269](#), [270](#)  
DavisThin, [499](#), [500](#), [502](#)  
Depredations, [472](#)  
Duncan, [xxvi](#), [34](#), [37](#), [47](#), [49](#), [53](#), [65](#), [67](#), [69–72](#), [125](#), [145](#), [227](#), [236](#), [239](#),  
[257](#), [397](#), [399](#), [401–403](#), [405](#), [483](#), [489–491](#)  
Erickson, [432](#), [433](#)  
Freedman, [73](#), [74](#), [78](#)  
GSSvocab, [214](#), [220](#), [266](#)  
Guyer, [58](#), [59](#), [79](#), [80](#), [86](#), [105](#), [106](#), [235](#), [504](#)  
MathAchieve, [339–341](#)  
MathAchSchool, [339–341](#)  
MplsDemo, [88](#), [90](#), [377](#)  
MplsStops, [87–89](#), [91](#), [109](#), [377](#)  
Mroz, [52](#), [53](#), [279](#), [285](#), [287–289](#), [423](#)  
nutrition, [233](#)  
OBrienKaiser, [91](#), [93](#), [95](#)  
Ornstein, [142](#), [150](#), [161](#), [164](#), [165](#), [296](#), [326](#), [328](#), [408](#), [422](#), [509](#), [528](#)  
Prestige, [72](#), [124](#), [125](#), [129](#), [131](#), [134](#), [136](#), [147](#), [157–160](#), [168](#), [183](#), [198](#),  
[205](#), [207](#), [208](#), [259](#), [260](#), [388](#), [390–394](#), [396](#), [469](#)  
Salaries, [468](#), [516](#), [519](#)  
SLID, [190](#), [192](#), [194](#), [196](#), [212](#), [223](#)  
Transact, [185](#), [186](#), [244](#), [247](#), [248](#), [250–256](#), [264](#), [268](#), [270](#), [324](#), [325](#), [415](#),  
[416](#)  
UN, [151](#), [152](#), [154–157](#), [163](#), [455](#), [457](#)  
Vocab, [139](#), [141](#)  
Womenlf, [82](#), [83](#), [86](#), [309](#), [310](#), [318](#), [419](#)  
Wool, [406](#), [407](#)

# Data Set Index

**alr4**, [55](#)  
**Amelia**, [72](#)  
**aplypack**, [128](#)  
**base**, [50](#), [70](#)  
**biglm**, [117](#)  
**boot**, [248–251](#), [256](#)  
**bootstrap**, [248](#)  
**car**, [xiii](#), [xviii](#), [xx](#), [xxii](#), [xxiv–xxvi](#), [xxviii](#), [2](#), [3](#), [24](#), [33](#), [34](#), [37](#), [38](#), [40–46](#),  
[51](#), [54](#), [55](#), [66–68](#), [72](#), [75–77](#), [80](#), [81](#), [83](#), [91](#), [123](#), [124](#), [128](#), [130–133](#), [136](#),  
[138](#), [141](#), [143–145](#), [147](#), [150](#), [153](#), [155](#), [157](#), [161–163](#), [165](#), [166](#), [168–172](#),  
[175](#), [178](#), [180–182](#), [192](#), [201](#), [207](#), [217](#), [226](#), [232](#), [236](#), [239](#), [243–245](#),  
[247–251](#), [253](#), [255–257](#), [259](#), [261](#), [262](#), [268](#), [272](#), [279](#), [287](#), [299](#), [301](#), [311](#),  
[322](#), [327](#), [330](#), [347](#), [357](#), [362](#), [370](#), [385](#), [386](#), [388–391](#), [393](#), [395](#), [397–399](#),  
[402](#), [404–406](#), [408](#), [410](#), [412](#), [416](#), [417](#), [425](#), [426](#), [428](#), [429](#), [432](#), [434](#), [437](#),  
[454](#), [455](#), [476](#), [479](#), [482](#), [499](#), [505](#)  
**carData**, [xiii](#), [xx](#), [xxii](#), [xxiv](#), [xxviii](#), [2](#), [3](#), [34](#), [52–55](#), [70](#), [72](#), [73](#), [82](#), [86](#), [87](#),  
[91](#), [124](#), [139](#), [151](#), [157](#), [176](#), [185](#), [190](#), [201](#), [214](#), [244](#), [279](#), [282](#), [309](#), [360](#),  
[377](#), [406](#), [415](#), [455](#), [457](#), [469](#), [473](#), [499](#), [504](#), [509](#), [516](#)  
**colorspace**, [454](#)  
**data.table**, [69](#)  
**dplyr**, [xx](#), [67](#), [72](#), [81](#), [88](#), [89](#), [341](#)  
**effects**, [xiii](#), [xx](#), [xxiv](#), [xxvi](#), [2](#), [123](#), [188](#), [189](#), [206](#), [217](#), [234](#), [283](#), [299](#), [331](#),  
[352](#), [369](#), [385](#), [412](#), [425](#), [426](#)  
**emmeans**, [203](#), [204](#), [218](#), [233](#), [292](#)  
**foreign**, [66](#)  
**ggmap**, [467](#), [469](#), [473](#)  
**ggplot2**, [438](#), [467](#), [469](#), [471](#), [472](#), [475](#), [476](#), [564](#)  
**glmmTMB**, [362](#)  
**grid**, [467](#), [476](#)  
**influence.ME**, [428](#)  
**iplots**, [476](#)  
**knitr**, [xx](#), [xxix](#), [25](#), [29](#)  
**lattice**, [146](#), [188](#), [189](#), [212](#), [216](#), [343](#), [361](#), [362](#), [365](#), [438](#), [467–469](#), [471](#),  
[476](#), [517](#), [518](#)  
**latticeExtra**, [468](#), [469](#)  
**lme4**, [91](#), [335](#), [339](#), [340](#), [344](#), [350](#), [352](#), [355](#), [358](#), [359](#), [361](#), [376](#), [377](#), [426](#)  
**lsmeans**, [203](#)

**lubridate**, [107–110](#)  
**maps**, [467](#), [469](#), [472](#)  
**MASS**, [318](#), [324](#), [328](#), [329](#), [406](#), [489](#), [492](#), [536](#)  
**matlib**, [492](#)  
**Matrix**, [492](#)  
**mgcv**, [171](#)  
**mi**, [72](#)  
**mice**, [72](#), [73](#)  
**microbenchmark**, [507](#)  
**nlme**, [91](#), [335](#), [339](#), [340](#), [343](#), [350](#), [352](#), [361](#), [426](#)  
**nnet**, [311](#)  
**norm**, [72](#)  
**ordinal**, [318](#), [322](#)  
**pbkrtest**, [357](#)  
**playwith**, [476](#)  
**plotrix**, [143](#), [476](#)  
**pscl**, [509](#)  
**quantreg**, [171](#)  
**Rcmdr**, [xiv](#)  
**readr**, [67](#), [68](#)  
**reshape2**, [95](#)  
**rggobi**, [146](#), [476](#)  
**rgl**, [145](#), [476](#)  
**RGtk2**, [476](#)  
**rio**, [66](#), [67](#)  
**rJava**, [476](#)  
**rmarkdown**, [xxii](#), [29](#)  
**rpart**, [437](#)  
**sandwich**, [247](#), [374](#)  
**sfsmisc**, [449](#)  
**shiny**, [475](#)  
**splines**, [195](#)  
**stats**, [70](#)  
**survey**, [117](#), [308](#)  
**tidyverse**, [67](#)  
**utils**, [70](#)  
**VGAM**, [311](#), [318](#), [322](#)

# Data Set Index

Functions marked with \* are in the **car** or **effects** packages; those marked with † are in CRAN packages that are not part of the standard R distribution.

\*[, 5, 7, 75, 212–214, 237, 486–488](#)  
\*\*[, 5](#)  
+[, 5, 7, 213, 236, 237, 471, 486](#)  
-[, 5, 7, 213, 237, 486](#)  
->[, 12, 89](#)  
/[, 5, 7, 237](#)  
:[, 11, 207, 213, 214, 236, 237, 496](#)  
::[, 72](#)  
<[, 16, 18](#)  
<-[, 12](#)  
<=[, 16](#)  
=[, 8, 12, 17](#)  
==[, 16, 17, 508](#)  
!=[, 16](#)  
>[, 16](#)  
>=[, 16](#)  
?[, 8, 23, 47](#)  
??[, 24](#)  
[][, 16, 70](#)  
[[ ]][, 103](#)  
%\*%[, 119, 487](#)  
%>%[, 88, 89, 341](#)  
%in%[, 85, 236, 237](#)  
%o%[, 488](#)  
&[, 17](#)  
&&[, 17, 497](#)  
![, 15, 17, 18](#)  
||[, 17, 497](#)  
|[, 17, 497](#)  
\$\a href="#">, 70  
^[, 5, 7, 237, 238](#)  
{ }[, 19, 36, 482, 497](#)  
abline[, 76, 145, 181, 446, 459, 483](#)  
abs[, 18, 457, 458, 493](#)  
adaptiveKernel\*[, 128–130, 362](#)

aes†, [471](#)  
all, [84](#)  
all.equal, [85](#), [508](#)  
Anova\*, [91](#), [192](#), [193](#), [217](#), [232](#), [261](#), [262](#), [264](#), [267](#), [272](#), [286](#), [287](#), [293](#),  
[296](#), [299](#), [305](#), [311](#), [322](#), [323](#), [327](#), [357](#), [368](#), [381](#), [417](#)  
anova, [48](#), [259–263](#), [285–287](#), [293](#), [356](#), [370](#), [535](#)  
aov, [48](#)  
apply, [499](#), [501–503](#), [509](#)  
apropos, [23](#)  
args, [10](#), [14](#), [127](#), [235](#)  
array, [97](#), [506](#)  
arrows, [449](#)  
as.factor, [199](#)  
as.list, [502](#)  
as.matrix, [97](#)  
as.numeric, [199](#)  
as.vector, [119](#)  
attach, [71](#), [72](#)  
avPlots\*, [43](#), [386](#), [393](#), [403](#), [404](#), [422](#), [437](#)  
axis, [143](#), [144](#), [440](#), [447](#), [452](#)  
basicPowerAxis\*, [150](#), [166](#)  
bcnPower\*, [161](#), [363](#), [368](#), [370](#), [408](#), [409](#), [427](#), [435](#)  
bcnPowerAxis\*, [166](#)  
bcnPowerInverse\*, [370](#), [409](#)  
bcPower\*, [163](#), [406](#)  
bcPowerAxis\*, [166](#)  
binom.test, [48](#)  
binomial, [280](#)  
Boot\*, [248](#), [249](#), [255](#)  
boot, [249](#)  
boot.ci, [256](#)  
bootCase\*, [249](#)  
box, [126](#), [440](#), [444](#)  
boxCox\*, [406](#)  
boxcox, [406](#)  
boxCoxVariable\*, [434](#)  
Boxplot\*, [133](#), [144](#), [166](#)  
boxplot, [48](#), [133](#), [144](#)  
boxTidwell\*, [435](#)

break, [498](#)  
brief\*, [55](#), [58](#), [68](#), [153](#), [180](#), [192](#), [283](#), [344](#), [417](#)  
browser, [523–526](#)  
bs, [195](#)  
bwplot, [468–470](#), [518](#)  
c, [11](#), [56](#), [82](#), [98](#), [149](#)  
carPalette\*, [454](#)  
carWeb\*, [xxviii](#), [3](#), [24](#)  
cat, [531](#), [533](#)  
cbind, [87](#), [182](#), [347](#)  
ceresPlots\*, [412](#)  
character, [506](#)  
chisq.test, [48](#), [302](#)  
choose, [31](#)  
class, [50](#), [198](#), [528](#)  
clm†, [318](#), [322](#)  
clm2†, [318](#)  
clmm†, [318](#)  
cloud, [146](#)  
coef, [182](#), [245](#), [345](#), [530](#)  
colMeans, [500](#)  
colors, [453](#)  
colSums, [500](#)  
compareCoefs\*, [46](#), [182](#), [375](#)  
complete.cases, [77](#)  
complex, [506](#)  
confidenceEllipse\*, [257](#)  
Confint\*, [180](#), [182](#), [251](#), [255](#), [256](#), [264](#), [281](#), [417](#)  
confint, [180](#), [251](#), [255](#)  
contr.helmert, [226](#), [265](#)  
contr.poly, [224](#), [227](#), [228](#), [265](#)  
contr.SAS, [225](#)  
contr.sum, [225](#), [226](#), [265](#), [293](#)  
contr.Treatment\*, [226](#)  
contr.treatment, [224–226](#), [265–267](#)  
contrasts, [200](#), [224](#)  
cooks.distance, [401](#), [421](#), [483](#)  
cor, [48](#), [228](#), [229](#), [491](#)  
corCAR1, [372](#)

count.fields, [61](#)  
cov, [48](#)  
cov2cor, [245](#)  
crPlots\*, [44](#), [171](#), [410](#), [412](#), [424](#), [426](#)  
curve, [451](#), [460](#)  
cut, [80](#), [81](#)  
D, [253](#)  
data, [55](#)  
data.frame, [58](#), [217](#)  
data\_frame†, [67](#)  
dataEllipse\*, [257](#)  
dbinom, [48](#), [133](#)  
dchisq, [48](#), [132](#), [133](#)  
debug, [523](#)  
debugonce, [523](#)  
deltaMethod\*, [253](#), [254](#), [272](#), [288](#), [325](#), [417](#)  
density, [129](#), [130](#)  
densityPlot\*, [41](#), [130](#)  
densityplot, [361](#)  
det, [491](#)  
detach, [71](#)  
deviance, [293](#)  
df, [48](#), [133](#)  
df.residual, [293](#)  
dfbeta, [403](#)  
dfbetas, [403](#)  
diag, [492](#)  
dim, [34](#)  
dmy†, [107](#)  
dnorm, [48](#), [132](#), [133](#), [503](#)  
dpois, [512](#)  
droplevels, [379](#)  
dt, [48](#), [133](#)  
dunif, [133](#)  
durbinWatsonTest\*, [435](#)  
Effect\*, [217](#), [234](#), [313](#), [369](#), [370](#), [426](#)  
eigen, [434](#), [491](#)  
else, [493](#), [494](#)  
emmeans†, [203](#), [204](#), [206](#), [218](#), [292](#), [293](#)

eqscplot, [536](#)  
eval.parent, [534](#)  
example, [10](#)  
exp, [9](#), [149](#)  
Export\*, [67](#)  
export†, [66](#), [67](#)  
expression, [459](#)  
factor, [198](#), [199](#), [227](#)  
factorial, [496](#)  
file.choose, [31](#), [64](#)  
fitted, [289](#), [535](#)  
floor, [482](#)  
for, [294](#), [446](#), [461](#), [495–498](#)  
format, [108](#)  
fractions, [489](#)  
friedman.test, [48](#)  
ftable, [48](#), [295](#), [379](#), [516](#)  
function, [19](#), [480](#)  
gam†, [171](#)  
gamLine\*, [171](#)  
gamma, [276](#)  
gamma.shape, [324](#)  
gaussian, [276](#)  
gc, [120](#)  
get\_googlemap†, [474](#)  
getOption, [224](#)  
getwd, [63](#)  
ggmap†, [474](#)  
ggplot†, [471](#)  
ginv, [492](#)  
glm, [52](#), [117](#), [118](#), [144](#), [193](#), [272](#), [274](#), [276](#), [277](#), [280](#), [281](#), [290](#), [311](#), [328](#),  
[330–332](#), [478](#)  
glm.nb, [328](#), [329](#)  
glmer†, [376](#), [377](#), [380](#), [426](#), [428](#)  
gray, [193](#), [452](#), [459](#)  
grep, [115](#)  
grid, [447](#)  
group\_by†, [89](#), [341](#)  
gsub, [114](#)

hatvalues, [388](#), [399](#), [483](#)  
hccm\*, [247](#), [248](#), [254](#), [255](#), [374](#)  
hcl, [452](#)  
head, [34](#), [176](#)  
help, [8](#), [14](#), [23](#), [25](#), [47](#), [55](#)  
help.search, [24](#)  
hist, [36](#), [47](#), [48](#), [125–127](#), [130](#), [250](#), [521](#)  
hist.default, [127](#)  
histogram, [216](#)  
hm†, [108](#)  
hms†, [108](#)  
hsv, [452](#)  
I, [193](#), [236](#), [366](#)  
identify, [31](#), [169–171](#), [437](#)  
if, [482](#), [492](#), [494](#)  
ifelse, [80](#), [84](#), [85](#), [294](#), [460](#), [493](#), [494](#)  
Import\*, [66](#), [67](#)  
import†, [66](#), [67](#)  
influence, [428](#), [429](#)  
influenceIndexPlot\*, [43](#), [399](#), [401](#), [421](#), [429](#)  
influencePlot\*, [402](#), [479](#), [482](#)  
install.packages, [20](#), [25](#)  
integer, [506](#)  
integrate, [503](#), [504](#)  
inverseResponsePlot\*, [406](#), [435](#)  
is.na, [78](#)  
kruskal.test, [48](#)  
labs†, [475](#)  
lag, [480](#)  
lapply, [499](#), [502](#), [503](#)  
layout, [463](#)  
legend, [77](#), [181](#), [438](#), [450](#), [452](#)  
length, [18](#), [19](#), [111](#)  
levels, [198](#)  
leveneTest\*, [435](#)  
leveragePlots\*, [395](#)  
library, [xxii](#), [3](#), [54](#), [55](#), [70–72](#), [176](#)  
linearHypothesis\*, [259](#), [264](#), [268](#), [270](#), [272](#), [288](#), [357](#), [417](#)  
lines, [130](#), [143](#), [438](#), [443–445](#), [452](#), [460](#)

list, 98  
llines, 362  
lm, xxvi, 39, 41, 46, 48, 50, 76, 79, 117–119, 144, 173, 176–179, 183, 186, 191, 193–195, 202, 207, 217, 227, 232, 234, 235, 240, 249, 272, 276, 280, 331, 350, 351, 460, 478, 483, 490, 491, 534  
lme, 335, 339, 349, 350, 356, 357, 367, 368, 372, 426, 428  
lmer†, 339, 349, 350, 355–357, 363, 367, 372, 376, 380, 426, 428  
lmList, 343  
load, 65, 66, 99, 121  
locator, 448, 450  
loess, 77, 171, 455, 461  
loessLine\*, 145, 171  
log, 7–10, 75, 149  
log10, 8, 9, 149  
log2, 9, 149  
logb, 149  
logical, 506  
logit\*, 168, 236  
lowess, 77  
lpoints, 362  
map†, 473  
mapply, 499, 503, 504  
marginalModelPlots\*, 391, 392  
match.arg, 495  
match.call, 534  
matrix, 96, 97, 486, 506  
mcPlot\*, 395  
mcPlots\*, 395  
md.pattern†, 73  
mdy†, 107  
mean, xxvi, 18, 19, 47, 48, 71, 74, 503, 505  
median, 48, 74, 76  
memory.limit, 118  
memory.size, 118, 120  
merge, 90, 341, 347  
methods, 51  
microbenchmark†, 507  
missing, 535  
model.matrix, 200, 228, 535

ms†, [108](#)  
mtext, [449](#), [461](#), [463](#), [465](#)  
multinom, [311](#)  
n†, [89](#)  
na.exclude, [76](#), [239](#)  
na.fail, [76](#), [239](#)  
na.omit, [76](#), [77](#), [215](#), [239](#), [260](#), [379](#), [457](#)  
nchar, [111](#)  
ncvTest\*, [45](#), [416](#)  
negative.binomial, [328](#)  
news, [xiii](#)  
ns, [195](#)  
numeric, [461](#), [506](#)  
offset, [241](#), [331](#)  
optim, [509](#), [510](#), [512](#), [513](#), [523](#), [528](#)  
options, [40](#), [76](#), [224](#), [441](#)  
order, [457](#)  
ordered, [227](#)  
outer, [488](#)  
outlierTest\*, [42](#), [398](#), [419](#)  
p.arrows†, [449](#)  
pairs, [146](#), [147](#), [171](#)  
palette, [453](#), [454](#)  
panel.lmline, [343](#)  
panel.loess, [343](#)  
panel.points, [343](#)  
panel.rug, [362](#)  
par, [144](#), [150](#), [166](#), [441–443](#), [457](#), [459](#), [463](#), [465](#)  
paste, [14](#), [111](#), [199](#)  
pbinom, [48](#), [133](#)  
pchisq, [48](#), [133](#)  
persp, [146](#)  
pf, [48](#), [133](#)  
pie, [453](#), [454](#)  
plot, [47](#), [48](#), [74](#), [75](#), [79](#), [134](#), [135](#), [144](#), [145](#), [147](#), [152](#), [171](#), [181](#), [188](#), [189](#),  
[193](#), [210](#), [212](#), [213](#), [284](#), [297](#), [300](#), [313](#), [352](#), [370](#), [425](#), [437–444](#), [447](#), [459](#),  
[465](#), [476](#), [483](#), [484](#)  
plot.default, [439](#)  
plot.formula, [439](#)

plot.lm, [439](#)  
plotCI†, [143](#), [476](#)  
pmax, [85](#)  
pnorm, [48](#), [132](#), [133](#), [504](#)  
points, [297](#), [438](#), [443](#), [452](#), [459](#), [460](#), [473](#)  
poisson, [296](#)  
polr, [318](#)  
poly, [154](#), [192](#), [193](#), [223](#), [228](#), [229](#), [411](#)  
polygon, [449](#), [450](#)  
poTest\*, [322](#)  
powerTransform\*, [155](#)–[158](#), [161](#), [162](#), [362](#), [363](#), [406](#), [409](#)  
prcomp, [434](#), [491](#)  
Predict\*, [417](#)  
predict, [289](#), [291](#), [460](#), [520](#)  
predictorEffects\*, [188](#), [192](#), [193](#), [211](#), [212](#), [221](#), [284](#), [301](#), [352](#), [426](#)  
princomp, [434](#), [491](#)  
print, [47](#), [68](#), [192](#), [357](#), [365](#), [468](#), [471](#), [528](#), [530](#), [531](#), [533](#), [535](#)  
printCoefmat, [531](#), [533](#)  
probabilityAxis\*, [166](#)  
prop.table, [48](#), [110](#), [379](#), [517](#)  
prop.test, [48](#)  
pt, [48](#), [133](#), [259](#)  
punif, [133](#)  
qbinom, [48](#), [133](#)  
qchisq, [48](#), [132](#), [133](#)  
qf, [48](#), [133](#)  
qnorm, [48](#), [132](#), [133](#)  
qplot†, [469](#)–[471](#)  
qqnorm, [131](#)  
qqPlot\*, [41](#), [42](#), [131](#), [132](#), [397](#)  
qt, [48](#), [133](#), [255](#)  
quantile, [48](#), [74](#), [81](#)  
quantregLine\*, [171](#)  
quasi, [276](#), [326](#)  
quasibinomial, [276](#), [326](#)  
quasipoisson, [276](#), [326](#)  
qunif, [133](#)  
rainbow, [452](#), [454](#)  
ranef†, [359](#)

range, [48](#)  
rbind, [86](#)  
rbinom, [48](#), [119](#), [133](#)  
rchisq, [48](#), [132](#), [133](#)  
read.csv, [62–65](#), [198](#)  
read.csv2, [62](#)  
read.fwf, [63](#)  
read.table, [59–62](#), [64](#), [120](#), [121](#), [198](#)  
read\_csv†, [67](#)  
read\_excel†, [67](#)  
read\_table†, [67](#), [68](#)  
readLines, [63](#), [111](#)  
readRDS, [66](#)  
Recall, [498](#)  
Recode\*, [80–82](#), [84](#)  
recode\*, [72](#), [81](#)  
recode†, [81](#)  
relevel, [201](#), [225](#)  
remove, [86](#)  
rep, [57](#)  
repeat, [495](#), [498](#)  
reshape, [93](#), [94](#), [294](#)  
residualPlot\*, [24](#), [171](#)  
residualPlots\*, [389–393](#), [411](#), [420](#), [437](#)  
residuals, [388](#), [535](#)  
return, [19](#), [496](#)  
rev, [463](#)  
rf, [48](#), [133](#)  
rnorm, [13](#), [48](#), [132](#), [133](#)  
round, [228](#)  
rowMeans, [500](#)  
rowSums, [500](#), [509](#)  
rqss†, [171](#)  
RSiteSearch, [24](#)  
rstandard, [388](#)  
rstudent, [41](#), [388](#), [419](#), [483](#)  
rt, [48](#), [133](#)  
rug, [130](#)  
runif, [133](#)

S\*, [178](#), [179](#), [186](#), [192](#), [245](#), [247](#), [248](#), [251](#), [253](#), [258](#), [264](#), [280](#), [281](#), [287](#),  
[299](#), [313](#), [318](#), [323](#), [330](#), [356](#), [358](#), [417](#)  
sample, [48](#), [83](#), [521](#)  
sapply, [499](#), [502](#), [503](#)  
save, [65](#), [66](#), [80](#), [99](#), [121](#)  
saveRDS, [66](#)  
scale, [186](#)  
scan, [57](#)  
scatter3d\*, [145](#), [146](#), [476](#)  
scatterplot\*, [136](#)–[140](#), [145](#), [147](#), [148](#), [171](#), [172](#), [347](#), [437](#), [455](#), [470](#)  
scatterplotMatrix\*, [37](#), [39](#), [147](#), [148](#), [152](#), [158](#), [159](#), [171](#), [437](#)  
sd, [21](#), [48](#), [74](#)  
search, [69](#)  
segments, [449](#)  
seq, [11](#)  
set.seed, [83](#)  
setwd, [63](#)  
shapiro.test, [132](#)  
showLabels\*, [74](#), [75](#), [170](#), [171](#), [181](#)  
sign, [494](#)  
simpleKey, [517](#)  
sin, [507](#)  
solve, [488](#), [489](#)  
some\*, [207](#)  
sort, [83](#), [115](#), [458](#)  
source, [537](#)  
splom, [146](#)  
spreadLevelPlot\*, [165](#), [435](#)  
sqrt, [9](#), [13](#)  
stem, [48](#), [128](#)  
stem.leaf†, [128](#)  
stop, [482](#), [497](#)  
str, [58](#), [69](#), [70](#), [502](#)  
strsplit, [112](#)  
strwrap, [111](#)  
sub, [114](#)  
subset, [106](#), [215](#)  
substring, [112](#)  
sum, [18](#), [19](#), [22](#), [78](#), [89](#), [111](#), [500](#), [501](#)

summarize†, [89](#), [341](#)  
summary, [2](#), [14](#), [35](#), [39](#), [47](#), [49–51](#), [81](#), [178](#), [179](#), [245](#), [251](#), [330](#), [363](#), [380](#),  
[528](#), [530](#), [532](#), [533](#), [535](#)  
summary.boot\*, [51](#)  
summary.default, [51](#)  
summary.lm, [50–52](#)  
summary.zipmod, [531](#)  
svd, [434](#)  
switch, [494](#), [495](#)  
symbolbox\*, [163](#)  
system.time, [118](#), [506](#), [507](#)  
t, [488](#)  
t.test, [48](#)  
table, [48](#), [115](#), [119](#)  
Tapply\*, [143](#), [201](#), [202](#), [215](#), [499](#), [505](#)  
tapply, [143](#), [202](#), [499](#), [504](#), [505](#)  
testTransform\*, [158](#)  
text, [448](#), [449](#), [459](#), [461](#), [483](#)  
title, [440](#), [465](#), [473](#)  
tolower, [114](#)  
trace, [499](#)  
traceback, [22](#), [523](#)  
transform, [80](#), [156](#)  
ts.plot, [48](#)  
unique, [115](#)  
untrace, [499](#)  
update, [46](#), [181](#), [182](#), [193](#), [205](#), [208](#)  
url, [64](#)  
UseMethod, [50](#)  
useOuterStrips†, [469](#)  
var, [48](#), [74](#), [186](#)  
vcov, [245](#), [248](#), [251](#), [253](#), [259](#), [375](#), [530](#), [531](#), [535](#)  
vcovCL†, [374](#), [375](#)  
vcovHC†, [247](#)  
vector, [506](#)  
Vectorize, [499](#), [504](#)  
vglm†, [318](#), [322](#)  
View, [34](#), [176](#)  
vif\*, [432](#), [434](#)

vignette, [24](#)  
weights, [331](#)  
whichNames\*, [46](#), [405](#)  
while, [495](#), [497](#), [498](#)  
wilcox.test, [48](#)  
wireframe, [146](#)  
with, [36](#), [71](#), [74](#), [80](#), [106](#), [125](#), [129](#), [491](#)  
within, [80](#)  
write.csv, [65](#)  
write.table, [65](#), [121](#)  
xtabs, [48](#), [201](#), [295](#), [297](#), [379](#), [380](#), [516](#)  
xyplot, [343](#), [365](#), [467](#), [469](#), [517](#)  
yjPower\*, [435](#)  
ymd†, [108](#)  
zapsmall, [229](#)  
zeroinfl†, [509](#)