The R Series

Reproducible Research with R and RStudio

Third Edition

Christopher Gandrud

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Reproducible Research with R and RStudio

Third Edition

Chapman & Hall/CRC
The R Series

Series Editors

John M. Chambers, Department of Statistics, Stanford University, California, USA
Torsten Hothorn, Division of Biostatistics, University of Zurich, Switzerland
Duncan Temple Lang, Department of Statistics, University of California, Davis, USA
Hadley Wickham, RStudio, Boston, Massachusetts, USA

Recently Published Titles

The Essentials of Data Science: Knowledge Discovery Using R
Graham J. Williams

blogdown: Creating Websites with R Markdown
Yihui Xie, Alison Presmanes Hill, Amber Thomas

Handbook of Educational Measurement and Psychometrics Using R
Christopher D. Desjardins, Okan Bulut

Displaying Time Series, Spatial, and Space-Time Data with R, Second Edition
Oscar Perpinan Lamigueiro

Reproducible Finance with R
Jonathan K. Regenstein, Jr

R Markdown
The Definitive Guide
Yihui Xie, J.J. Allaire, Garrett Grolemund

Practical R for Mass Communication and Journalism
Sharon Machlis

Analyzing Baseball Data with R, Second Edition
Max Marchi, Jim Albert, Benjamin S. Baumer

Spatio-Temporal Statistics with R
Christopher K. Wikle, Andrew Zammit-Mangion, and Noel Cressie

Statistical Computing with R, Second Edition
Maria L. Rizzo

Geocomputation with R
Robin Lovelace, Jakub Nowosad, Jannes Muenchow

Advanced R, Second Edition
Hadley Wickham

Dose Response Analysis Using R
Christian Ritz, Signe Marie Jensen, Daniel Gerhard, Jens Carl Streibig

Distributions for Modelling Location, Scale, and Shape
Using GAMLSS in R
Robert A. Rigby , Mikis D. Stasinopoulos, Gillian Z. Heller and Fernanda De Bastiani

Hands-On Machine Learning with R
Bradley Boehmke and Brandon Greenwell

Statistical Inference via Data Science
A ModernDive into R and the Tidyverse
Chester Ismay and Albert Y. Kim

Reproducible Research with R and RStudio, Third Edition
Christopher Gandrud

For more information about this series, please visit: https://www.crcpress.com/go/the-r-series

Reproducible Research with R and RStudio

Third Edition

Christopher Gandrud

CRC Press
Taylor & Francis Group
Boca Raton London New York

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

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2020 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper

International Standard Book Number-13: 978-0-367-14398-5 (Paperback)
International Standard Book Number-13: 978-0-367-14402-9 (Hardback)

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

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

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

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

Library of Congress Cataloging-in-Publication Data

Names: Gandrud, Christopher, author.
Title: Reproducible research with R and RStudio / by Christopher Gandrud.
Description: Third edition. | Boca Raton, FL : CRC Press, [2020] | Series: The R series | Includes bibliographical references and index. | Summary: "Brings together the skills and tools needed for doing and presenting computational research. Using straightforward examples, the book takes you through an entire reproducible research workflow"-- Provided by publisher.
Identifiers: LCCN 2019046298 (print) | LCCN 2019046299 (ebook) | ISBN 9780367143985 (paperback) | ISBN 9780367144029 (hardback) | ISBN 9780429031854 (ebook)
Subjects: LCSH: Research--Statistical methods. | R (Computer program language)
Classification: LCC Q180.55.S7 G36 2020 (print) | LCC Q180.55.S7 (ebook) | DDC 001.4/2202855133--dc23
LC record available at https://lccn.loc.gov/2019046298
LC ebook record available at https://lccn.loc.gov/2019046299

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface xi

About the Author xv

Stylistic Conventions xvii

Additional Resources xix

I Getting Started 1

1 Introducing Reproducible Research 3
- 1.1 What Is Reproducible Research? . . . 4
- 1.2 Why Should Research Be Reproducible? . . . 5
  - 1.2.1 For science . . . 5
  - 1.2.2 For you . . . 6
- 1.3 Who Should Read This Book? . . . 8
  - 1.3.1 Academic researchers . . . 9
  - 1.3.2 Students . . . 9
  - 1.3.3 Instructors . . . 9
  - 1.3.4 Editors . . . 10
  - 1.3.5 Private sector researchers . . . 10
- 1.4 The Tools of Reproducible Research . . . 11
  - 1.4.1 Why Use R, knitr/R Markdown, and RStudio for Reproducible Research? . . . 12
- 1.5 Installing the main software . . . 15
  - 1.5.1 Installing markup languages . . . 15
  - 1.5.2 GNU Make . . . 16
  - 1.5.3 Other tools . . . 16
- 1.6 Book Overview . . . 17
  - 1.6.1 How to read this book . . . 18
  - 1.6.2 Reproduce this book . . . 19
  - 1.6.3 Contents overview . . . 19

2 Getting Started with Reproducible Research 23
- 2.1 The Big Picture: A Workflow for Reproducible Research . . . 23
  - 2.1.1 Reproducible theory . . . 24
- 2.2 Practical Tips for Reproducible Research . . . 25

2.2.1 Document everything! . . . 26
2.2.2 Everything is a (text) file . . . 27
2.2.3 All files should be human readable . . . 28
2.2.4 Explicitly tie your files together . . . 30
2.2.5 Have a plan to organize, store, and make your files available . . . 32

3 Getting Started with R, RStudio, and knitr/R Markdown 33
3.1 Using R: The Basics . . . 33
3.1.1 Objects . . . 34
3.1.2 Functions . . . 42
3.1.3 The workspace and history . . . 45
3.1.4 R history . . . 46
3.1.5 Global R options . . . 46
3.1.6 Installing new packages and loading functions . . . 47
3.2 Using RStudio . . . 47
3.3 Using knitr and R Markdown: The Basics . . . 50
3.3.1 What *knitr* does . . . 50
3.3.2 What *rmarkdown* does . . . 50
3.3.3 File extensions . . . 53
3.3.4 Code chunks . . . 53
3.3.5 Global chunk options . . . 55
3.3.6 *knitr* package options . . . 57
3.3.7 Hooks . . . 57
3.3.8 knitr, R Markdown, and RStudio . . . 57
3.3.9 knitr and R . . . 61
3.3.10 R Markdown and R . . . 63

Appendix: Jupyter Interactive Notebooks 65

Appendix: knitr and Lyx 67

4 Getting Started with File Management 69
4.1 File Paths and Naming Conventions . . . 70
4.1.1 Root directories . . . 70
4.1.2 Sub-directories and parent directories . . . 70
4.1.3 Working directories . . . 71
4.1.4 Absolute vs. relative paths . . . 71
4.1.5 Spaces in directory and file names . . . 73
4.2 Organizing Your Research Project . . . 73
4.3 Organizing Research with RStudio Projects . . . 74
4.4 R File Manipulation Functions . . . 75
4.5 Unix-like Shell Commands for File Management . . . 79
4.6 File Navigation in RStudio . . . 83

II Data Gathering and Storage 85

5 Storing, Collaborating, Accessing Files, and Versioning 87
5.1 Saving Data in Reproducible Formats . . . 88
5.2 Storing Your Files in the Cloud: Dropbox . . . 89
5.2.1 Storage . . . 90
5.2.2 Accessing data . . . 91
5.2.3 Collaboration . . . 92
5.2.4 Version control . . . 92
5.3 Storing Your Files in the Cloud: GitHub . . . 93
5.3.1 Setting up GitHub: Basic . . . 95
5.3.2 Version control with Git . . . 96
5.3.3 Remote storage on GitHub . . . 104
5.3.4 Accessing on GitHub . . . 106
5.3.5 Summing up the GitHub workflow . . . 109
5.4 RStudio and GitHub . . . 110
5.4.1 Setting up Git/GitHub with Projects . . . 110
5.4.2 Using Git in RStudio Projects . . . 111

6 Gathering Data with R 113
6.1 Organize Your Data Gathering: Makefiles . . . 113
6.1.1 R Make-like files . . . 114
6.1.2 GNU Make . . . 115
6.2 Importing Locally Stored Data Sets . . . 121
6.3 Importing Data Sets from the Internet . . . 122
6.3.1 Data from non-secure (*http*) URLs . . . 122
6.3.2 Data from secure (*https*) URLs . . . 123
6.3.3 Compressed data stored online . . . 123
6.3.4 Data APIs and feeds . . . 124
6.4 Advanced Automatic Data Gathering: Web Scraping . . . 126

7 Preparing Data for Analysis 129
7.1 Cleaning Data for Merging . . . 129
7.1.1 Get a handle on your data . . . 129
7.1.2 Reshaping data . . . 132
7.1.3 Renaming variables . . . 135
7.1.4 Ordering data . . . 136
7.1.5 Subsetting data . . . 137
7.1.6 Recoding string/numeric variables . . . 139
7.1.7 Creating new variables from old . . . 140
7.1.8 Changing variable types . . . 143
7.2 Merging Data Sets . . . 143
7.2.1 Binding . . . 143
7.2.2 Merging data frames . . . 144
7.2.3 Duplicate columns . . . 147

Appendix 149

III Analysis and Results 151

8 Statistical Modeling and knitr/R Markdown 153
8.1 Incorporating Analyses into the Markup . . . 154
8.1.1 Full code chunks . . . 154
8.1.2 Showing code and results inline . . . 157
8.1.3 Dynamically including non-R code in code chunks . . 159
8.2 Dynamically Including Modular Analysis Files . . . 159
8.2.1 Source from a local file . . . 160
8.2.2 Source from a URL . . . 162
8.3 Reproducibly Random: `set.seed()` . . . 163
8.4 Computationally Intensive Analyses . . . 164

9 Showing Results with Tables 167
9.1 Basic *knitr* Syntax for Tables . . . 168
9.2 Table Basics . . . 168
9.2.1 Tables in LaTeX . . . 169
9.2.2 Tables in Markdown/HTML . . . 173
9.3 Creating Tables from Supported Class R Objects . . . 177
9.3.1 `kable` for Markdown and LaTeX . . . 177
9.3.2 *xtable* for LaTeX and HTML . . . 178
9.3.3 *texreg* for LaTeX and HTML . . . 181
9.3.4 Fitting large tables in LaTeX . . . 184
9.3.5 *xtable* with non-supported class objects . . . 185
9.3.6 Creating variable description documents with *xtable* . . 187

10 Showing Results with Figures 191
10.1 Including Non-knitted Graphics . . . 192
10.1.1 Including graphics in LaTeX . . . 192
10.1.2 Including graphics in Markdown/HTML . . . 194
10.1.3 Non-knitted graphics with *knitr/rmarkdown* . . . 195
10.2 Basic *knitr/rmarkdown* Figure Options . . . 196
10.2.1 Chunk options . . . 196
10.2.2 Global options . . . 197
10.3 Knitting R's Default Graphics . . . 198
10.4 Including *ggplot2* Graphics . . . 202
10.4.1 Showing regression results with caterpillar plots . . . 205
10.5 JavaScript Graphs with *googleVis* . . . 208
10.5.1 Basic googleVis figures . . . 209
10.5.2 Including *googleVis* in knitted documents . . . 210
10.5.3 JavaScript Graphs with *htmlwidgets*-based packages . . 211

IV Presentation Documents **213**

11 Presenting with LaTeX **215**
11.1 The Basics . . . 216
11.1.1 Getting started with LaTeX editors . . . 216
11.1.2 Basic LaTeX command syntax . . . 217
11.1.3 The LaTeX preamble and body . . . 217
11.1.4 Headings . . . 222
11.1.5 Paragraphs and spacing . . . 222
11.1.6 Horizontal lines . . . 222
11.1.7 Text formatting . . . 223
11.1.8 Math . . . 224
11.1.9 Lists . . . 225
11.1.10 Footnotes . . . 226
11.1.11 Cross-references . . . 226
11.2 Bibliographies with BibTeX . . . 226
11.2.1 The *.bib* file . . . 227
11.2.2 Including citations in LaTeX documents . . . 228
11.2.3 Generating a BibTeX file of R package citations . . . 228
11.3 Presentations with LaTeX Beamer . . . 231
11.3.1 Beamer basics . . . 231
11.3.2 *knitr* with LaTeX slideshows . . . 234

12 Presenting in a Variety of Formats with R Markdown **237**
12.1 The Basics . . . 237
12.1.1 Getting started with Markdown editors . . . 238
12.1.2 Preamble and document structure . . . 239
12.1.3 Headings . . . 240
12.1.4 Horizontal lines . . . 240
12.1.5 Paragraphs and new lines . . . 240
12.1.6 Italics and bold . . . 241
12.1.7 Links . . . 241
12.1.8 Lists . . . 241
12.1.9 Math with MathJax . . . 242
12.2 Further Customizability with *rmarkdown* . . . 243
12.2.1 CSS style files and Markdown . . . 247
12.3 Slideshows with Markdown, R Markdown, and HTML . . . 248
12.3.1 HTML slideshows with *rmarkdown* . . . 249
12.3.2 LaTeX Beamer slideshows with *rmarkdown* . . . 250
12.3.3 Slideshows with Markdown and RStudio's R Presentations . . . 251
12.4 Publishing HTML Documents Created with R Markdown . . . 254
12.4.1 Further information on R Markdown . . . 256

13 Conclusion **257**
13.1 Citing Reproducible Research 258
13.2 Licensing Your Reproducible Research 259
13.3 Sharing Your Code in Packages 259
13.4 Project Development: Public or Private? 260
13.5 Is it Possible to Completely Future-Proof Your Research? . . . 261

Bibliography **263**

Index **271**

Preface

Motivation

This book has its genesis in my PhD research at the London School of Economics. I started the degree with questions about the 2008/09 financial crisis and planned to spend most of my time researching capital adequacy requirements. But I quickly realized that I would actually spend a large proportion of my time learning the day-to-day tasks of data gathering, analysis, and results presentation. After plodding through for a while with Word, Excel, and Stata, my breaking point came while reentering results into a regression table after I had tweaked one of my statistical models, yet again. Surely there was a better way to *do* research that would allow me to spend more time answering my research questions. Making research reproducible for others also means making it better organized and efficient for yourself. My search for a better way led me straight to the tools for reproducible computational research.

The reproducible research community is very active, knowledgeable, and helpful. Nonetheless, I often encountered holes in this collective knowledge, or at least had no resource organizing it all together as a whole. That is my intention for this book: to bring together the skills I have picked up for actually doing and presenting computational research. Hopefully, the book, along with making reproducible research more widely used, will save researchers hours of googling, so they can spend more time addressing their research questions.

Changes to the Third Edition

- Spring cleaning: updated package recommendations, examples, and URLs. Removed technologies no longer in regular use.
- More advanced R Markdown and less LaTeX in discussions of markup languages and examples.
- Stronger focus on reproducible working directory tools.

- Updated discussion of cloud storage services and persistently citing reproducible material.
- Added discussion of Jupyter notebooks and reproducible practices in industry.
- Examples of data manipulation with Tidyverse tibbles (in addition to standard data frames) and `pivot_longer()` and `pivot_wider()` functions for pivoting data.
- Naming conventions are in current R-Tidyverse best practice.

A detailed list of changes for the third edition is available at `https://github.com/christophergandrud/Rep-Res-Book/issues/57#issuecomment-421739971`.

Changes to the Second Edition

The tools of reproducible research have developed rapidly since the first edition of this book was published just two years ago. The second edition has been updated to incorporate the most important of these advancements, including discussions of:

- The *rmarkdown* package, which allows you to create reproducible research documents in PDF, HTML, and Microsoft Word formats using the simple and intuitive Markdown syntax.
- Improvements and changes to RStudio's interface and capabilities, such as its new tools for handling R Markdown documents.
- Expanded *knitr* R code chunk capabilities.
- The `kable()` function in the *knitr* package and the *texreg* package for dynamically creating tables to present your data and statistical results.
- An improved discussion of file organization allowing you to take full advantage of relative file paths so that your documents are more easily reproducible across computers and systems.
- The *dplyr*, *magrittr*, and *tidyr* packages for fast data manipulation.
- Numerous changes to R syntax in user-created packages.
- Changes to GitHub's and Dropbox's interfaces.

Acknowledgments

I would not have been able to write this book without many people's advice and support. Foremost is John Kimmel, acquisitions editor at Chapman & Hall. He approached me in Spring 2012 with the general idea and opportunity for this book. Other editors at Chapman & Hall and Taylor & Francis have greatly contributed to this project, including Marcus Fontaine. I would also like to thank all of the book's reviewers whose helpful comments have greatly improved it. The first edition's reviewers include:

- Jeromy Anglim, Deakin University
- Karl Broman, University of Wisconsin, Madison
- Jake Bowers, University of Illinois, Urbana-Champaign
- Corey Chivers, McGill University
- Mark M. Fredrickson, University of Illinois, Urbana-Champaign
- Benjamin Lauderdale, London School of Economics
- Ramnath Vaidyanathan, McGill University

Many other anonymous reviewers also gave great feedback over the years.

The developer and blogging community has also been incredibly important for making this book possible. Foremost among these people is Yihui Xie. He is the main developer behind the *knitr* package, co-developer of *rmarkdown*, and also an avid blog writer and commenter. Without him, the ability to do reproducible research would be much harder and the blogging community that spreads knowledge about how to do these things would be poorer. Other great contributors to the reproducible research community include Carl Boettiger, Karl Broman, Markus Gesmann (who developed *googleVis*), Rob Hyndman, and Hadley Wickham (who has developed numerous very useful R packages). Thank you also to Victoria Stodden and Michael Malecki for helpful suggestions. And, of course, thank you to everyone at RStudio (especially JJ Allaire) for creating an increasingly useful program for reproducible research.

The second edition has benefited immensely from first edition readers' comments and suggestions. For a list of their valuable contributions, please see the book's GitHub Issues page `https://github.com/christophergandrud/Rep-Res-Book/issues` and the first edition's Errata page `http://christophergandrud.github.io/RepResR-RStudio/errata.htm`.

My students at Yonsei University were an important part of making the first edition. One of the reasons that I got interested in using many of the tools covered in this book, like using *knitr* in slideshows, was to improve a course I taught there: Introduction to Social Science Data Analysis. I tested many of the explanations and examples in this book on my students. Their feedback has been very helpful for making the book clearer and more useful. Their

experience with using these tools on Microsoft Windows computers was also important for improving the book's Windows documentation. Similarly, my students at the Hertie School of Governance inspired and tested key sections of the second edition.

The vibrant community at Stack Overflow `http://stackoverflow.com/` and Stack Exchange `http://stackexchange.com/` are always very helpful for finding answers to problems that plague any computational researcher. Importantly, the sites make it easy for others to find the answers to questions that have already been asked.

The library at the University of California, San Francisco was a great home for writing the third edition.

Kristina Gandrud has been immensely supportive and patient with me throughout the writing of this book (and my entire career).

About the Author

Christopher Gandrud is Head of Economics and Experimentation at Zalando SE. He leads teams of social data scientists and software engineers building and evaluating large-scale automated decision-making systems. He was previously a research fellow at the Institute for Quantitative Social Science, Harvard University developing statistical software for the social and physical sciences. He has held posts at City, University of London, the Hertie School of Governance, Yonsei University, and the London School of Economics where in 2012 he completed a PhD in quantitative political science.

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

Stylistic Conventions

I use the following conventions throughout the book:

- **Abstract variables**: Abstract variables, i.e. variables that do not reference specific objects, are in `ALL CAPS TYPEWRITER TEXT`.
- **Clickable buttons**: Clickable buttons are in `typewriter text`.
- **Code**: All code is in `typewriter text`.
- **File names and directories**: File names and directories more generally are printed in *italics*. Words are separated by em dashes—*kebab-case*.[1]
- **File extensions**: Like file names, file extensions are *italicized*.
- **Individual variable values**: Individual variable values mentioned in the text are in *italics*.
- **Objects**: Objects are printed in *italics*. I use underscores (_) to separate words in object names.
- **Object columns**: Data frame object columns are printed in **bold**.
- **R Function names**: are followed by parentheses (e.g., `stats::lm()`)
- **Packages**: **R** packages are printed in *italics*. When a system, rather than the package that shares its name is referred to, it is not italicized, e.g. R Markdown (system) vs. *rmarkdown* (package).[2]
- **Windows and RStudio panes**: Open windows and RStudio panes are written in *italics*.
- **Variable names**: Variable names are printed in **bold**. Underscores (_) separate words in variable names.

[1] See `https://stackoverflow.com/a/17820138`. Posted 23 July 2013.

[2] See Yihui Xie's comment at: `https://andrewgelman.com/2016/01/14/rstanarm-and-more/#comment-259425`. Posted 14 January 2016.

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

Additional Resources

You can freely download additional resources supplementing examples in this book. These resources include longer examples discussed in individual chapters and a complete short reproducible research project.

Chapter Examples

Longer examples discussed in individual chapters, including files to dynamically download data, code for creating figures, and markup files for creating presentation documents, can be accessed at: `https://github.com/christophergandrud/rep-res-book-v3-examples`. Please see Chapter 5 for more information on downloading files from GitHub, where the examples are stored.

Short Example Project

To download a full (though very short) example of a reproducible research project created using the tools covered in this book, go to: `https://github.com/christophergandrud/rep-res-book-v3-examples`. Please follow the replication instructions in the main *README.md*. It is a good idea to hold off looking at this complete example in detail until after you have become acquainted with the individual tools it uses. Become acquainted with the tools by reading through this book and working with the chapter examples.

The following two figures give you a sense of how the example's files are organized. Figure 1 shows how the files are organized in the file system. Figure 2 illustrates how the main files are dynamically tied together. In the *data* directory, we have files to gather raw data from the World Bank (2018) on fertilizer consumption and from Pemstein et al. (2010) on countries' levels of democracy. They are tied to the data through the `WDI()` and `download.file()`

functions. A *Makefile* can run *gather-1* and *gather-2.R* to gather and clean the data. It runs *merge-data.R* to merge the data into one data file called *main-data.csv*. It also automatically generates a variable description file and a *README.md* recording the session info.

The *analysis* folder contains two files that create figures presenting this data. They are tied to *main-data.csv* with the `import()` function. These files are run by the presentation documents when they are knitted. The presentation documents tie to the analysis documents with *knitr* and the `source()` function.

Though a simple example, hopefully these files will give you a complete sense of how a reproducible research project can be organized. Please feel free to experiment with different ways of organizing the files and tying them together to make your research really reproducible.

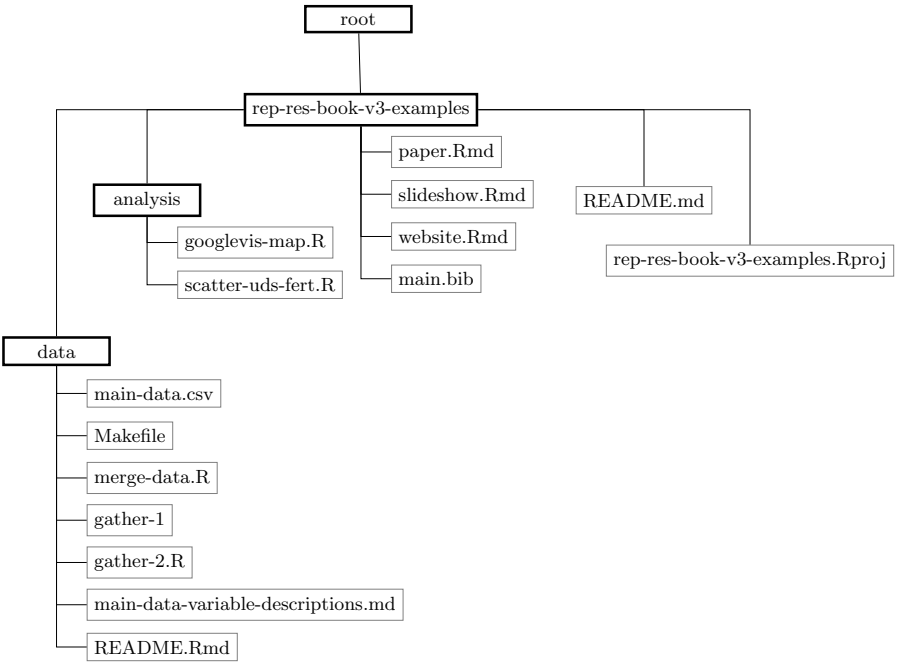


FIGURE 1: Short Example Project File Tree

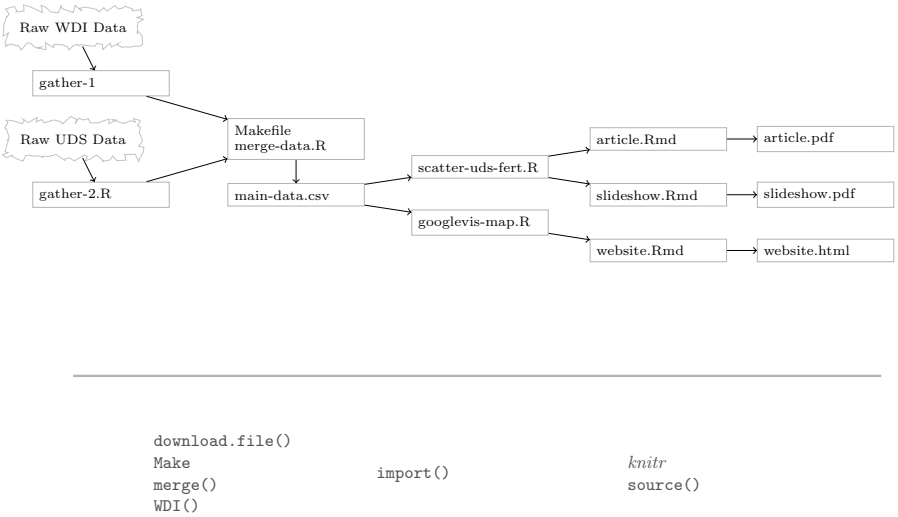


FIGURE 2: Short Example Main File Ties

Updates

Many of the reproducible research tools discussed in this book are improving rapidly. Because of this, I will regularly post updates to the content covered in the book at: `https://github.com/christophergandrud/Rep-Res-Book`.

Corrections

If you notice any corrections that should be made to fix typos, broken URLs, and so on, you can report them at: `https://github.com/christophergandrud/Rep-Res-Book/issues`. I'll post notifications of changes to an Errata page at: `http://christophergandrud.github.io/RepResR-RStudio/errata.htm`.

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

Part I

Getting Started

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

1

Introducing Reproducible Research

Research is typically presented in very selective containers: slideshows, journal articles, books, or websites. These presentation documents announce a project's findings and try to convince us that the results are correct (Mesirov, 2010). It's important to remember that these documents are not the research. Especially in the computational and statistical sciences, these documents are the "advertising". The research is the "full software environment, code, and data that produced the results" (Buckheit and Donoho, 1995; Donoho, 2010, 385). When we separate the research from its advertisement, we are making it difficult for others to verify the findings by reproducing them.

This book gives you the tools to dynamically combine your research with the presentation of your findings. The first tool is a workflow for reproducible research that weaves the principles of reproducibility throughout your entire research project, from data gathering to the statistical analysis, and the presentation of results. You will also learn how to use a number of computer tools that make this workflow easier and more robust. These tools include:

- the **R** statistical language that will allow you to gather data and analyze it;
- the **LaTeX** and **Markdown** markup languages that you can use to create documents–slideshows, articles, books, and webpages–for presenting your findings;
- the *knitr* and *rmarkdown* **packages** for R and other tools, including **command-line programs** like GNU Make and Git version control, for dynamically tying your data gathering, analysis, and presentation documents together so that they can be easily reproduced;
- **RStudio**, a program that brings all of these tools together.

1.1 What Is Reproducible Research?

Though there is some debate over the necessary and sufficient conditions for a full replication (Makel and Plucker, 2014, 2), research results are generally considered[1] *replicable* if there is sufficient information available for independent researchers to make the same findings using the same procedures with new data.[2] For research that relies on experiments, this can mean a researcher not involved in the original research being able to rerun the experiment, including sampling, and validate that the new results are comparable to the original results. In computational and quantitative empirical sciences, results are replicable if independent researchers can recreate findings by following the procedures originally used to gather the data and run the computer code. Of course, it is sometimes difficult to replicate the original data set because of issues such as limited resources to gather new data or because the original study already sampled the full universe of cases. So as a next-best standard, we can aim for "*really reproducible research*" (Peng, 2011, 1226).[3] In computational sciences[4] this means:

> the data and code used to make a finding are available and they are sufficient for an independent researcher to recreate the finding.

In practice, research needs to be *easy* for independent researchers to reproduce (Ball and Medeiros, 2011). If a study is difficult to reproduce, it's more likely that no one will reproduce it. If someone does attempt to reproduce this research, it will be difficult for them to tell if any errors they find were in the

[1] Rokem et al. (2018, 3-4) note that some disciplines, e.g. computing machinery and meteorology, give "replicable" and "reproducible" the exact opposite meanings from the way they are used in this book and many other disciplines such as biology, economics, and epidemiology.

[2] This is close to what Lykken (1968) calls "operational replication".

[3] The really reproducible computational research originates in the 1980s and early 1990s with Jon Claerbout and the Stanford Exploration Project (Fomel and Claerbout, 2009; Donoho et al., 2009). Further seminal advances were made by Jonathan B. Buckheit and David L. Donoho who created the Wavelab library of MATLAB routines for their research on wavelets in the mid-1990s (Buckheit and Donoho, 1995).

[4] Reproducibility is important for both quantitative and qualitative research (King et al., 1994). Nonetheless, we will focus mainly on on methods for reproducibility in quantitative computational research.

original research or problems they introduced during the reproduction. In this book, you will learn how to avoid these problems.

In particular, you will learn tools for dynamically "*knitting*"[5] the data and the source code together with your presentation documents. Combined with well-organized source files and clearly and completely commented code, independent researchers will be able to understand how you obtained your results. This will make your computational research easily reproducible.

1.2 Why Should Research Be Reproducible?

Reproducible research is one of the main components of science. If that's not enough reason for you to make your research reproducible, consider that the tools of reproducible research also have direct benefits for you as a researcher.

1.2.1 For science

Replicability has been a key part of scientific inquiry from perhaps the 1200s (Bacon, 1859; Nosek et al., 2012). It has even been called the "demarcation between science and non-science" (Braude, 1979, 2). Why is replication so important for scientific inquiry?

Standard to judge scientific claims

Replication opens claims to scrutiny, allowing us to keep what works and discard what doesn't. Science, according to the American Physical Society, "is the systematic enterprise of gathering knowledge . . . organizing and condensing that knowledge into testable laws and theories". The "ultimate standard" for evaluating scientific claims is whether or not the claims can be replicated (Peng, 2011; Kelly, 2006). Research findings cannot even really be considered "genuine contributions to human knowledge" until they have been verified through replication (Stodden, 2009b, 38). Replication "requires the complete and open exchange of data, procedures, and materials". Scientific conclusions

[5] Much of the reproducible computational research and literate programming literatures have traditionally used the term "weave" to describe the process of combining source code and presentation documents (see Knuth, 1992, 101). In the R community, the term "weave" is usually used to describe the combination of source code and LaTeX documents. The term "knit" reflects the vocabulary of the *knitr* R package (knit + R). It is used more generally to describe weaving with a variety of markup languages. The term is used by RStudio if you are using the *rmarkdown* package, which is similar to *knitr*. We also cover the *rmarkdown* package in this book. Because of this, I use the term knit rather than weave in this book.

that are not replicable should be abandoned or modified "when confronted with more complete or reliable . . . evidence".[6]

Reproducibility enhances replicability. If other researchers are able to clearly understand how a finding was originally made, then they will be better able to conduct comparable research in meaningful attempts to replicate the original findings. Sometimes strict replicability is not feasible, for example, when it is only possible to gather one data set on a population of interest. In these cases reproducibility is a "minimum standard" for judging scientific claims (Peng, 2011).

It is important to note that though reproducibility is a minimum standard for judging scientific claims, "a study can be reproducible and still be wrong" (Peng, 2014). For example, a statistically significant finding in one study may remain statistically significant when reproduced using the original data/code, but when researchers try to replicate it using new data and even methods, they are unable to find a similar result. The original finding could have been noise, even though it is fully reproducible.

Avoiding effort duplication and encouraging cumulative knowledge development

Not only is reproducibility important for evaluating scientific claims, it can also contribute to the cumulative growth of scientific knowledge (Kelly, 2006; King, 1995). Reproducible research cuts down on the amount of time scientists have to spend gathering data or developing procedures that have already been collected or figured out. Because researchers do not have to discover on their own things that have already been done, they can more quickly build on established findings and develop new knowledge.

1.2.2 For you

Working to make your research reproducible does require extra upfront effort. For example, you need to put effort into learning the tools of reproducible research by doing things such as reading this book. But beyond the clear benefits for science, why should you make this effort? Using reproducible research tools can make your research process more effective and (hopefully) ultimately easier.

[6] See the American Physical Society's website at `http://www.aps.org/policy/statements/99_6.cfm`. See also Fomel and Claerbout (2009).

Better work habits

Making a project reproducible from the start encourages you to use better work habits. It can spur you to more effectively plan and organize your research. It should push you to bring your data and source code up to a higher level of quality than you might if you "thought 'no one was looking'" (Donoho, 2010, 386). This forces you to root out errors–a ubiquitous part of computational research–earlier in the research process (Donoho, 2010, 385). Clear documentation also makes it easier to find errors.[7]

Reproducible research needs to be stored so that other researchers can actually access the data and source code. By taking steps to make your research accessible for others, you are also making it easier for yourself to find your data and methods when you revise your work or begin a new project. You are avoiding personal effort duplication, allowing you to cumulatively build on your own work more effectively.

Better teamwork

The steps you take to make sure an independent researcher can figure out what you have done also make it easier for your collaborators to understand your work and build on it. This applies not only to current collaborators, but also to future collaborators. Bringing new members of a research team up to speed on a cumulatively growing research project is faster if they can easily understand what has been done already (Donoho, 2010, 386).

Changes are easier

A third person may or may not actually reproduce your research even if you make it easy for them to do so. But, *you will almost certainly reproduce parts or even all of your own research.* No actual research process is completely linear. You almost never gather data, run analyses, and present your results without going backwards to add variables, make changes to your statistical models, create new graphs, alter results tables in light of new findings, and so on. You will probably try to make these changes long after you last worked on the project and long since you remembered the details of how you did it. Whether your changes are because of journal reviewers' and conference participants' comments or you discover that new and better data has been made available since beginning the project, designing your research to be reproducible from the start makes it much easier to change things later on.

Dynamic reproducible documents make changes much easier. Changes made to

[7] Of course, it's important to keep in mind that reproducibility is "neither necessary nor sufficient to prevent mistakes" (Stodden, 2009a).

one part of a research project have a way of cascading through the other parts. For example, adding a new variable to a largely completed analysis requires gathering new data and merging it with existing data sets. If you used data imputation or matching methods, you may need to rerun these models. You then have to update your main statistical analyses, and recreate the tables and graphs you used to present the results. Adding a new variable essentially forces you to reproduce large portions of your research. If when you started the project you used tools that make it easier for others to reproduce your research, you also made it easier to reproduce the work yourself. You will have taken steps to have a "better relationship with your future self" (Bowers, 2011, 2).

Higher research impact

Reproducible research is more likely to be useful for other researchers than non-reproducible research. Useful research is cited more frequently (Donoho, 2002; Piwowar et al., 2007; Vandewalle, 2012). Research that is fully reproducible contains more information, i.e. more reasons to use and cite it, than presentation documents merely showing findings. Independent researchers may use the reproducible data or code to look at other, often unanticipated, questions. When they use your work for a new purpose they will (should) cite your work. Because of this, Vandewalle et al. even argue that "the goal of reproducible research is to have more impact with our research" (2007, 1253).

A reason researchers often avoid making their research fully reproducible is that they are afraid other people will use their data and code to compete with them. I'll let Donoho et al. address this one:

> *True. But competition means that strangers will read your papers, try to learn from them, cite them, and try to do even better. If you prefer obscurity, why are you publishing?* (2009, 16)

1.3 Who Should Read This Book?

This book is intended primarily for researchers who want to use a systematic workflow that encourages reproducibility as well as practical state-of-the-art

computational tools to put this workflow into practice. These people include professional researchers, upper-level undergraduate, and graduate students working on computational data-driven projects. Hopefully, editors at academic publishers will also find the book useful for improving their ability to evaluate and edit reproducible research.

The more researchers that use the tools of reproducibility, the better. So I include enough information in the book for people who have very limited experience with these tools, including limited experience with R, LaTeX, and Markdown. They will be able to start incorporating reproducible research tools into their workflow right away. The book will also be helpful for people who already have general experience using technologies such as R and LaTeX, but would like to know how to tie them together for reproducible research.

1.3.1 Academic researchers

Hopefully so far in this chapter I've convinced you that reproducible research has benefits for you as a member of the scientific community and personally as a computational researcher. This book is intended to be a practical guide for how to actually make your research reproducible. Even if you already use tools such as R and LaTeX, you may not be getting their full potential. This book will teach you useful ways to get the most out of them as part of a reproducible research workflow.

1.3.2 Students

Upper-level undergraduate and graduate students conducting original computational research should make their research reproducible for the same reasons that professional researchers should. Forcing yourself to clearly document the steps you took will also encourage you to think more clearly about what you are doing and reinforce what you are learning. It will hopefully give you a greater appreciation of research accountability and integrity early in your career (Barr, 2012; Ball and Medeiros, 2011, 183).

Even if you don't have extensive experience with computer languages, this book will teach you specific habits and tools that you can use throughout your student research and hopefully your careers. Learning these things earlier will save you considerable time and effort later.

1.3.3 Instructors

When instructors incorporate the tools of reproducible research into their assignments, they not only build students' understanding of research best

practice, but are also better able to evaluate and provide meaningful feedback on students' work (Ball and Medeiros, 2011, 183). This book provides a resource that you can use with students to put reproducibility into practice.

If you are teaching computational courses, you may also benefit from making your lecture material dynamically reproducible. Your slides will be easier to update for the same reasons that it is easier to update research. Making the methods you used to create the material available to students will give them more information. Clearly documenting how you created lecture material can also pass information on to future instructors.

1.3.4 Editors

When the first edition of this book was published, there was a worrying lack of reproduciblity in published research. The infrastructure was weak (Peng, 2011) and many journals did not require it. However, the situation has largely changed for the better: many journals require all analyses to be in some sense reproducible. The journal *Biostatistics* is a good example of a publication that is encouraging (actually requiring) reproducible research. From 2009 the journal has had an editor for reproducibility that ensures replication files are available and that results can be replicated using these files (Peng, 2009). The more editors there are with the skills to work with reproducible research, the more likely it is that researchers will do it.

We need to maintain and continuously improve these standards. This book is useful for editors at academic publishers who want to be better at evaluating reproducible research, editing it, and developing systems to make it more widely available.

1.3.5 Private sector researchers

Researchers in the private sector may or may not want to make their work easily reproducible outside of their organization. Data compliance legislation, such as the European Union's General Data Protection Regulation (GDPR), may even make it legally problematic to share data even within a company in order to protect personal information. However, that does not mean that significant benefits cannot be gained from using the methods of reproducible research, even if only in part.

Even if a company has only one person doing research, it benefits from using reproducible research methods. Just as with academic research, this person actually does have a collaborator: their future self. As discussed above, reproducible research makes this collaboration easier.

Companies with more than one researcher do (or likely should) act as a research

community, even if public reproducibility is ruled out to guard proprietary information.[8] Making as much of your research reproducible (e.g. your source code, but not the raw data if it contains personal information) to members of your organization can spread valuable information about how analyses were done and data was collected. This will help build your organization's knowledge and avoid effort duplication. Just as a lack of reproducibility hinders the spread of information in the scientific community, it can hinder it inside of a private organization. Using the sort of dynamic automated processes run with clearly documented source code we will learn in this book can also help create robust data analysis methods that help your organization avoid errors that may come from cutting-and-pasting data across spreadsheets.[9]

1.4 The Tools of Reproducible Research

This book will teach you the tools you need to make your research highly reproducible. Reproducible research involves two broad sets of tools. The first is a **reproducible research environment** that includes the statistical tools you need to run your analyses as well as "the ability to automatically track the provenance of data, analyses, and results and to package them (or pointers to persistent versions of them) for redistribution". The second set of tools is a **reproducible research publisher**, which prepares dynamic documents for presenting results and is easily linked to the reproducible research environment (Mesirov, 2010, 415).

In this book, we will focus on learning how to use the widely available and highly flexible reproducible research environment–R/RStudio (R Core Team, 2019; RStudio, Inc., 2019).[10] R/RStudio can be linked to numerous reproducible research publishers such as LaTeX and Markdown with Yihui Xie's *knitr* package (2020b) or the related *rmarkdown* package (Allaire et al., 2019b). The main tools covered in this book include:

- **R**: a programming language primarily for statistics and graphics. It can also be useful for data gathering and creating presentation documents.
- ***knitr* and *rmarkdown***: related R packages for literate programming. They allow you to combine your statistical analysis and the presentation

[8] There are ways to enable some public reproducibility without revealing confidential information. See (Vandewalle et al., 2007) for a discussion of one approach.

[9] See this post by David Smith about how the J.P. Morgan "London Whale" problem may have been prevented with the type of processes covered in this book: `http://blog.revolutionanalytics.com/2013/02/did-an-excel-error-bring-down-the-london-whale.html` (posted 11 February 2013).

[10] The book was created with R version 3.6.2 and RStudio preview release version 1.2.5019.

of the results into one document. They work with R and a number of other languages such as Bash, Python, and Ruby.

- **Markup languages**: instructions for how to format a presentation document. In this book, we cover LaTeX, Markdown, and a little HTML.
- **RStudio**: an integrated developer environment (IDE) for R that tightly combines R, *knitr*, *rmarkdown*, and markup languages.
- **Cloud storage and versioning**: Git/GitHub that can store data, code, and presentation files, save previous versions of these files, and make this information widely available.
- **Unix-like shell programs**: These tools are useful for working with large research projects.[11] They also allow us to use command-line tools including GNU Make for compiling projects and Pandoc, a program useful for converting documents from one markup language to another.

1.4.1 Why Use R, knitr/R Markdown, and RStudio for Reproducible Research?

Why use R?

Why use a statistical programming language like R for reproducible research? R has a very active development community that is constantly expanding what it is capable of. As we will see in this book, R enables researchers across a wide range of disciplines to gather data and run statistical analyses. Using the *knitr* or *rmarkdown* package, you can connect your R-based analyses to presentation documents created with markup languages such as LaTeX and Markdown. This allows you to dynamically and reproducibly present results in articles, slideshows, and webpages.

The way you interact with R has benefits for reproducible research. In general you interact with R (or any other programming and markup language) by explicitly writing down your steps as source code. This promotes reproducibility more than your typical interactions with Graphical User Interface (GUI) programs like SPSS[12] and Microsoft Word. When you write R code and embed it in presentation documents created using markup languages, you are forced to explicitly state the steps you took to do your research. When you do research by clicking through drop-down menus in GUI programs, your steps are lost, or at least documenting them requires considerable extra effort. Also it is generally more difficult to dynamically embed your analysis in presentation

[11] In this book, I cover the Bash shell for Linux and Mac as well as Windows PowerShell.

[12] I know you can write scripts in statistical programs like SPSS, but doing so is not encouraged by the program's interface and you often have to learn multiple languages for writing scripts that run analyses, create graphics, and deal with matrices.

documents created by GUI word processing programs in a way that will be accessible to other researchers both now and in the future. I'll come back to these points in Chapter 2.

Why use knitr and R Markdown?

Literate programming is a crucial part of reproducible quantitative research.[13] Being able to directly link your analyses, your results, and the code you used to produce the results makes tracing your steps much easier. There are many different literate programming tools for a number of different programming languages.[14] Previously, one of the most common tools for researchers using R and the LaTeX markup language was *Sweave* (Leisch, 2002). The packages I am going to focus on in this book are newer and have more capabilities. They are called *knitr* and *rmarkdown*. Why are we going to use these tools in this book and not *Sweave* or some other tool?

The simple answer is that they are more capable than *Sweave*. Both *knitr* and *rmarkdown* can work with markup languages other than LaTeX including Markdown and HTML. *rmarkdown* can even output Microsoft Word documents. They can work with programming languages other than R. They highlight R code in presentation documents making it easier for your readers to follow.[15] They give you better control over the inclusion of graphics and can cache code chunks, i.e. save the output for later. *knitr* has the ability to understand *Sweave*-like syntax, so it will be easy to convert backwards to *Sweave* if you want to.[16] You also have the choice to use much simpler and more straightforward syntax with *knitr* and *rmarkdown*.

knitr and *rmarkdown* have broadly similar capabilities and syntax. They both are literate programming tools that can produce presentation documents from multiple markup languages. They have almost identical syntax when used in Markdown. Their main difference is that they take different approaches to creating presentation documents. *knitr* documents must be written using the markup language associated with the desired output. For example, with *knitr*, LaTeX must be used to create PDF output documents and Markdown or HTML must be used to create webpages. R Markdown builds directly on knitr,

[13] Donald Knuth coined the term literate programming in the 1970s to refer to a source file that could be both run by a computer and "woven" with a formatted presentation document (Knuth, 1992).

[14] A very interesting tool that is worth taking a look at for the Python programming language is HTML Notebooks created with Jupyter. For more details see `http://jupyter.org/`. We will also discuss these at the end of Chapter 3.

[15] Syntax highlighting uses different colors and fonts to distinguish different types of text.

[16] Note that the Sweave-style syntax is not identical to actual *Sweave* syntax. See Yihui Xie's discussion of the differences between the two at: `http://yihui.name/knitr/demo/sweave/`. *knitr* has a function (`Sweave2knitr`) for converting *Sweave* to *knitr* syntax.

the key difference being that it uses the straightforward Markdown markup language to generate PDF, HTML, and MS Word documents.[17]

Because you write with the simple Markdown syntax, R Markdown is generally easier to use. It has the advantage of being able to take the same markup document and output multiple types of presentation documents. Nonetheless, for complex documents like books and long articles or work that requires custom formatting, knitr LaTeX is often preferable and extremely flexible, though the syntax is more complicated.

Why use RStudio?

Why use the RStudio integrated development environment for reproducible research? R by itself has the capabilities necessary to gather data, analyze it, and, with a little help from knitr/R Markdown and markup languages, present results in a way that is highly reproducible. RStudio allows you to do all of these things, but simplifies many of them and allows you to navigate through them more easily. It also is a happy medium between R's text-based interface and a pure GUI.

Not only does RStudio do many of the things that R can do but more easily, it is also a very good standalone editor for writing documents with LaTeX and Markdown. For LaTeX documents it can, for example, insert frequently used commands like `\section{}` for numbered sections (see Chapter 11).[18] There are many LaTeX editors available, both open source and paid. But RStudio is currently the best program for creating reproducible LaTeX and Markdown documents. It has full syntax highlighting. Its syntax highlighting can even distinguish between R code and markup commands in the same document. It can spell check LaTeX and Markdown documents. It handles knitr/R Markdown code chunks beautifully (see Chapter 3).

Finally, RStudio not only has tight integration with various markup languages, it also has capabilities for using other tools such as C++, CSS, JavaScript, Python, and a few other programming languages. It is closely integrated with the version control programs Git and SVN. Both of these programs allow you to keep track of the changes you make to your documents (see Chapter 5). This is important for reproducible research since version control programs can document many of your research steps. It also has a built-in ability to make HTML slideshows from knitr/R Markdown documents. Basically, RStudio

[17] It does this by relying on a tool called Pandoc (MacFarlane, 2019).

[18] If you are more comfortable with a what-you-see-is-what-you-get (WYSIWYG) word processor like Microsoft Word, you might be interested in exploring Lyx. It is a WYSIWYG-like LaTeX editor that works with *knitr*. It doesn't work with the other markup languages covered in this book. For more information, see: `https://www.lyx.org/`. I give some brief information on using Lyx with *knitr* in Chapter 3's Appendix.

makes it easy to create and navigate through complex reproducible research documents.

1.5 Installing the main software

Before you read this book you should install the main software. All of the software programs covered in this book are open source and can be easily downloaded for free. They are available for Windows, Mac, and Linux operating systems. They should run well on most modern computers.

You should install R before installing RStudio. You can download the programs from the following websites:

- **R**: `https://www.r-project.org/`,
- **RStudio Desktop (Open Source License)**: `https://www.rstudio.com/products/rstudio/download/`.

The webpages for downloading these programs have comprehensive information on how to install them. Please refer to those pages for more information.

After installing R and RStudio, you will probably also want to install a number of user-written packages that are covered in this book. To install all of these user-written packages, please see this chapter's Appendix.

1.5.1 Installing markup languages

You will need to install the R package *rmarkdown* (Allaire et al., 2019b) to turn your markdown documents into polished output that can be presented (e.g. as a website or PDF). To do this in R, use:

```
install.packages("rmarkdown")
```

If you plan to render your R Markdown documents from the console without RStudio, you will need to install Pandoc. For instructions, see Pandoc's download page: `https://pandoc.org/installing.html`. If you use RStudio, this step is unnecessary as Pandoc will be installed automatically.

If you want to create LaTeX (PDF) documents, you can install a TeX distri-

bution.[19] The simplest way to get all of the LaTeX capabilities you will need for this book is to use the *tinytex* (Xie, 2020c) R package:

```
install.packages('tinytex')
tinytex::install_tinytex()
```

If you want a full LaTeX distribution, see `https://www.latex-project.org/get/` for installation information.

1.5.2 GNU Make

If you are using a Linux computer, you already have GNU Make installed.[20] Mac users will need to install the command-line developer tools. There are two ways to do this. One is go to the App Store and download Xcode (it's free). Once Xcode is installed, install command-line tools, which you will find by opening Xcode then clicking on `Preference Downloads`. However, Xcode is a very large download and you only need the command-line tools for Make. To install just the command-line tools, open the Terminal and try to run Make by typing `make` and hitting return. A box should appear asking you if you want to install the command-line developer tools. Click `Install`. Windows users will have Make installed if they have already installed *Rtools* (see this chapter's Appendix). Mac and Windows users will need to install this software not only so that GNU Make runs properly, but also so that other command-line tools work well.

1.5.3 Other tools

We will discuss other tools such as Git that can be a useful part of a reproducible research workflow. Installation instructions for these tools will be discussed below.

[19] LaTeX is really a set of macros for the TeX typesetting system. It is included in all major TeX distributions.

[20] To verify this, open the Terminal and type: `make -version` (I used version 3.81 for this book). This should output details about the current version of Make installed on your computer.

1.6 Book Overview

The purpose of this book is to give you the tools that you will need to do reproducible research with R and RStudio. This book describes a workflow for reproducible research primarily using R and RStudio. It is designed to give you the necessary tools to use this workflow for your own research. It is not designed to be a complete reference for R, RStudio, *knitr*/*rmarkdown*, Git, or any other program that is a part of this workflow. Instead, it shows you how these tools can fit together to make your research more reproducible. To get the most out of these individual programs, I will along the way point you to other resources that cover these programs in more detail.

To that end, I can recommend a number of resources that cover more of the nitty-gritty:

- Michael J. Crawley's (2013) encyclopedic R book, appropriately titled ***The R Book***, published by Wiley.
- Hadley Whickham (2014a) has a great new book out from Chapman & Hall on ***Advanced R***.
- Yihui Xie's (2015) book ***R Markdown: The Definitive Guide***, published by Chapman & Hall, is needless to say the definitive guide on R Markdown syntax. It's a good complement to this book's generally more research project–level focus.
- Cathy O'Neil and Rachel Schutt (2013) give an introduction to the field of data science generally in ***Doing Data Science***, published by O'Reilly Media Inc.
- For many real-world examples of reproducible research in action see Kitzes et al.'s (2018) collection of case studies ***The Practice of Reproducible Research***.
- For an excellent introduction to the command-line in Linux and Mac, see William E. Shotts Jr.'s (2012) book ***The Linux Command-line: A Complete Introduction*** published by No Starch Press. It is also helpful for Windows users running PowerShell (see Chapter 4). Sean Kross' (2018) ***The Unix Workbench*** is also a great freely available online introduction to the topic.
- The RStudio website (`https://support.rstudio.com/hc/en-us/categories/200035113-Documentation`) has a number of useful tutorials on how to use *knitr* with LaTeX and Markdown. They also have very good documentation for *rmarkdown* at `https://rmarkdown.rstudio.com/`.

That being said, my goal is for this book to be *self-sufficient*. A reader without

a detailed understanding of these programs will be able to understand and use the commands and procedures I cover in this book. While learning how to use R and the other programs, I personally often encountered illustrative examples that included commands, variables, and other things that were not well explained in the texts that I was reading. This caused me to waste many hours trying to figure out, for example, what the `$` is used for (preview: it's the component selector). I hope to save you from this wasted time by either providing a brief explanation of possibly frustrating and mysterious things and/or pointing you in the direction of good explanations.

1.6.1 How to read this book

This book gives you a workflow. It has a beginning, middle, and end. So, unlike a reference book, it can and should be read linearly as it takes you through the organizational steps of an empirical research process from an empty folder to a completed set of documents that reproducibly showcase your findings.

That being said, readers with more experience using tools like R or LaTeX may want to skip over the nitty-gritty parts of the book that describe how to manipulate data frames or compile LaTeX documents into PDFs. Please feel free to skip these sections.

More experienced R users

If you are an experienced R user you may want to skip over the first section of Chapter 3: Getting Started with R, RStudio, and *knitr*/*rmarkdown*. But don't skip over the whole chapter. The latter parts contain important information on the *knitr*/*rmarkdown* packages. If you are experienced with R data manipulation, you may also want to skip all of Chapter 7.

More experienced LaTeX users

If you are familiar with LaTeX, you might want to skip the first part of Chapter 11. The second part may be useful as it includes information on how to dynamically create BibTeX bibliographies with *knitr* and how to include *knitr*/*rmarkdown* output in a Beamer slideshow.

Less experienced LaTeX/Markdown users

If you do not have experience with LaTeX or Markdown, you may benefit from reading, or at least skimming, the introductory chapters on these top topics (Chapters 11 and 12) before reading Part III.

1.6.2 Reproduce this book

This book practices what it preaches. It can be reproduced. I wrote the book using the programs and methods that I describe. Full documentation and source files can be found at the book's GitHub repository. Feel free to read and even use (within reason and with attribution, of course) the book's source code. You can find it at: `https://github.com/christophergandrud/Rep-Res-Book`. This is especially useful if you want to know how to do something in the book that I don't directly cover in the text.

If you notice any errors or places where the book can be improved please report them on the book's GitHub Issues page: `https://github.com/christophergandrud/Rep-Res-Book/issues`. Corrections will be posted at: `http://christophergandrud.github.io/RepResR-RStudio/errata.htm`.

1.6.3 Contents overview

The book is broken into four parts. Chapters 2, 3, and 4 give an overview of the reproducible research workflow as well as the general computer skills that you'll need to use this workflow. Each of the next three parts of the book guides you through the specific skills you will need for each part of the reproducible research process. Chapters 5, 6, and 7 cover the data gathering and file storage process. Chapters 8, 9, and 10 teach you how to dynamically incorporate your statistical analysis, results figures, and tables into your presentation documents. Finally, Chapters 11 and 12 cover how to create reproducible presentation documents including LaTeX articles, slideshows, and webpages.

Appendix: Additional R Setup

Some setup is required to reproduce this book. Here are key R packages you should consider installing and specific instructions for Windows and Linux users.

R Packages

In this book, I discuss how to use a number of user-written R packages for reproducible research. Many of these packages are not included in the default R installation. They need to be installed separately.

Note: in general you should aim to minimize the number of packages that your research depends on. Doing so will lessen the possibility that your code will "break" when a package is updated. This book depends on relatively many packages because of its special and unusual purpose of illustrating a variety of tools that you can use for reproducible research.

To install key user-written packages discussed in this book, copy the following code and paste it into your R console:

```
# Packages to install
pkg_to_install <- c("brew", "brms", "bookdown", "devtools",
                    "googleVis", "knitr", "rio", "rmarkdown",
                    "tidyverse", "WDI", "xfun", "texreg",
                    "xtable")


# Check if the packages are installed, if not install them
lapply(
    pkg_to_install,
    function(pkg) {
        if (system.file(package = pkg) == "") {
            install.packages(pkg,
                repos = "http://cran.us.r-project.org"
            )
        }
    }
)
```

Note that I specified a US based R Project CRAN "mirror" to download the packages from.[21] There are many others to choose from. See: `https://cran.r-project.org/mirrors.html`.

The *xfun* package (Xie, 2020d) contains a function called `pkg_attach2()`. When supplied with a vector of package names like those in `pkg_to_install` above, will install all non-installed packages. `p_load()` from the *pacman* package (Rinker and Kurkiewicz, 2019) works in a similar way. These functions are much less verbose than the example above, but they do require the user to install the package separately before `pkg_attach2()` or `p_load()` can be used. The example above relies only on functions available in the basic **R** installation.

[21] CRAN stands for the Comprehensive R Archive Network.

Special issues

You may need to install ImageMagick `https://www.imagemagick.org/script/index.php` compile the book from source.

If you are using Windows, you will also need to install *Rtools*. You can install *Rtools* from: `http://cran.r-project.org/bin/windows/Rtools/`. Please use the recommended installation to ensure that your system PATH is set up correctly. Otherwise, your computer will not know where the tools are. Alternatively, use the `install.Rtools()` function from the *installr* (Galili et al., 2018) package to install it.

On Linux, you will need to install the *RCurl* (Temple Lang and the CRAN team, 2020) package separately. Use your Terminal to install these packages with the following (or similar depending on your system) code:

```
apt-get update

apt-get install libcurl4-gnutls-dev
apt-get install r-cran-rcurl-dev
```

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

2

Getting Started with Reproducible Research

Researchers often start thinking about making their work reproducible near the end of the research process when they write up their results or maybe later when a journal requires their data and code be made available for publication. Or maybe later when another researcher asks if they can use the data from a published article to reproduce the findings. By then there may be numerous versions of the data set and records of the analyses stored across multiple folders on the researcher's computers. It can be difficult and time consuming to sift through these files to create an accurate account of how the results were reached. Waiting until near the end of the research process to start thinking about reproducibility can lead to incomplete documentation that does not give an accurate account of how findings were made. Focusing on reproducibility from the beginning of the process and continuing to follow a few simple guidelines throughout your research can help you avoid these problems. Remember "reproducibility is not an afterthought–it is something that must be built-into the project from the beginning" (Donoho, 2010, 386).

This chapter first gives you a brief overview of the reproducible research process: a workflow for reproducible research. Then it covers some of the key guidelines that can help make your research more reproducible.

2.1 The Big Picture: A Workflow for Reproducible Research

The three basic stages of a typical computational empirical research project are:

- data gathering,
- data analysis, and
- results presentation.

Each stage is part of the reproducible research workflow covered in this book. Tools for reproducibly gathering data are covered in Part II. Part III teaches

tools for tying the data we gathered to our statistical analyses and presenting the results with tables and figures. Part IV discusses how to tie these findings into a variety of documents you can use to advertise your findings.

Instead of starting to use the individual tools of reproducible research as soon as you learn them, I recommend briefly stepping back and considering how the stages of reproducible research *tie* together. This will make your workflow more coherent from the beginning and save you a lot of backtracking later on. Figure 2.1 illustrates the workflow. Notice that most of the arrows connecting the workflow's parts point in both directions, indicating that you should always be thinking about how to make it easier to go backward through your research, i.e. reproduce it, as well as forward.

Around the edges of the figure are some of the functions you will learn to make it easier to go forward and backward through the process. These functions tie your research together. For example, you can use API-based R packages to gather data from the internet. You can use R's `merge()` function to combine data gathered from different sources into one data set. The `getURL()` function from R's *RCurl* package (Temple Lang and the CRAN team, 2020) and the `read.table()` function in base R or the much more versatile `import()` function from the *rio* package (hong Chan and Leeper, 2018) can be used to bring this data set into your statistical analyses. The *knitr* or *rmarkdown* package then ties your analyses into your presentation documents. This includes the code you used, the figures you created, and, with the help of tools such as the `kable()` function in the *knitr* package, tables of results. You can even tie multiple presentation documents together. For example, you can access the same figure for use in a LaTeX article and a Markdown-created website with the LaTeX `includegraphics` function or *knitr*'s `include_graphics()` function. This helps you maintain a consistent presentation of results across multiple document types. We'll cover these functions in detail throughout the book. See Table 2.1 for an additional overview of some of the *tie functions*.

2.1.1 Reproducible theory

An important part of the research process that I do not discuss in this book is the theoretical stage. If you are using a deductive research design, the bulk of this work will precede and guide the data gathering and analysis stages. Just because I don't cover this stage of the research process doesn't mean that theory building can't and shouldn't be reproducible. It can in fact be "the easiest part to make reproducible" (Vandewalle et al., 2007, 1254). Quotes and paraphrases from previous works in the literature obviously need to be fully cited so that others can verify that they accurately reflect the source material. For mathematically based theory, you should give clear and complete descriptions of the proofs.

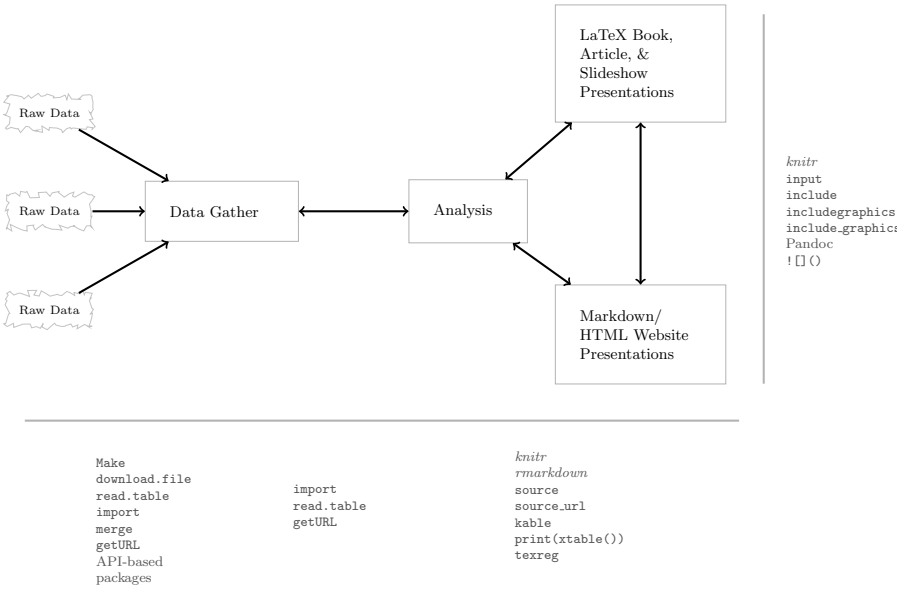


FIGURE 2.1: Example Workflow and a Selection of Functions to Tie It Together

Though I don't actively cover theory replication in depth in this book, I do touch on some of the ways to incorporate proofs and citations into your presentation documents. These tools are covered in Part IV.

2.2 Practical Tips for Reproducible Research

Before we start learning the details of the reproducible research workflow with R and RStudio, it's useful to cover a few broad tips that will help you organize your research process and put these skills in perspective. The tips are:

1. Document everything!
2. Everything is a (text) file.
3. All files should be human readable.
4. Explicitly tie your files together.
5. Have a plan to organize, store, and make your files available.

Using these tips will help make your computational research really reproducible.

2.2.1 Document everything!

In order to reproduce your research, others must be able to know what you did. You have to tell them what you did by documenting as much of your research process as possible. Ideally, you should tell your readers how you gathered your data, analyzed it, and presented the results. Documenting everything is the key to reproducible research and lies behind all of the other tips in this chapter and tools you will learn throughout the book.

Document your R session info

Before discussing the other tips, it's important to learn a key part of documenting with R. You should *record your session info*. Many things in R have stayed the same since it was introduced in the early 1990s. This makes it easy for future researchers to recreate what was done in the past. However, things can change from one version of R to another and especially from one version of an R package to another. Also, the way R functions and how R packages are handled can vary across different operating systems, so it's important to note what system you used. Finally, you may have R set to load packages by default (see Section 3.1.6 for information about packages). These packages might be necessary to run your code, but other people might not know what packages and what versions of the packages were loaded from just looking at your source code. The `sessionInfo()` function in R prints a record of all of these things. The information from the session I used to create this book is:

```
# Print R session info
sessionInfo()
```

```
## R version 3.6.2 (2019-12-12)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS Catalina 10.15.2
##
## Matrix products: default
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] methods   base
```

```
##
## loaded via a namespace (and not attached):
##  [1] compiler_3.6.2  magrittr_1.5    bookdown_0.17
##  [4] tools_3.6.2     htmltools_0.4.0 rstudioapi_0.10
##  [7] yaml_2.2.0      Rcpp_1.0.3      stringi_1.4.5
## [10] rmarkdown_2.0   knitr_1.27      stringr_1.4.0
## [13] xfun_0.12       digest_0.6.23   rlang_0.4.2
## [16] evaluate_0.14
```

Chapter 4 gives specific details about how to create files with dynamically included session information. If you used non-R tools you should also record what versions of these tools you used.

2.2.2 Everything is a (text) file

Your documentation is stored in files that include data, analysis code, the write-up of results, and explanations of these files (e.g. data set codebooks, session info files, and so on). Ideally, you should use the simplest file format possible to store this information. Usually the simplest file format is the humble, but versatile, text file.[1]

Text files are extremely nimble. They can hold your data in, for example, comma-separated values (CSV) format. They can contain your analysis code in files. And they can be the basis for your presentations written in markup languages such as Markdown and LaTeX. All of these files can be opened by any program that can read text files.

One reason reproducible research is best stored in text files is that this helps *future-proof* your research. Other file formats, like those used by Microsoft Word (`.docx`) or Excel (`.xlsx`), change regularly and may not be compatible with future versions of these programs. Text files, on the other hand, can be opened by a very wide range of currently existing programs and, more likely than not, future ones as well. Even if future researchers do not have R or a LaTeX distribution, they will still be able to open your text files and, aided by frequent comments (see below), be able to understand how you conducted your research (Bowers, 2011, 3).

Text files are also very easy to search and manipulate with a wide range of programs–such as R and RStudio–that can find and replace characters as well as merge and separate files. Finally, text files are easy to version control. Changes can be tracked using programs such as Git (see Chapter 5).

[1] Plain text files are usually given the file extension `.txt`. Depending on the size of your data set, it may not be feasible to store it as a text file. Nonetheless, text files can still be used for analysis code and presentation files.

Learn from the text file: keep it simple

Text files are simple. Their simplicitly increases the probability of baseline usefulness in the future to researchers who will reproduce the work. We can extend the logic of the simple text file to all of the tools we use: keep it simple. Avoid adding dependencies you don't need to actually gather your data, analyze it, and present the results. For example, I have been tempted to make my presentation slides look nicer with custom fonts. I was later burned when I wanted to make minor changes to slides a year after I first presented them (and a day before teaching an upcoming class) only to find that the custom fonts were no longer available. This broke my slides and forced me to spend considerable time reworking writing my source documents. If I, the creator of the slides, found this time consuming and annoying, an independent researcher would likely find it even more difficult.

2.2.3 All files should be human readable

Treat all of your research files as if someone who has not worked on the project will, in the future, try to understand them. Computer code is a way of communicating with the computer. It is 'machine readable' in that the computer is able to use it to understand what you want to do.[2] However, there is a very good chance that other people (or you six months in the future) will not understand what you were telling the computer. So, you need to make all of your files 'human readable'. To make them human readable, you should comment on your code with the goal of communicating its design and purpose (Wilson et al., 2012). With this in mind, it is a good idea to *comment frequently* (Bowers, 2011, 3) and *format your code using a style guide* (Nagler, 1995). For especially important pieces of code, you should use *literate programming*–where the source code and the presentation text describing its design and purpose appear in the same document. Doing this will make it very clear to others how you accomplished a piece of research.

Commenting

In R, everything on a line after a hash character (also known as 'number', 'pound', or 'sharp') is ignored by R, but is readable to people who open the file. The hash character is a comment declaration character. You can use a hash to place comments telling other people what you are doing. Here are some examples:

[2] Of course, if the computer does not understand, it will usually give an error message.

```
# A complete comment line
2 + 2 # A comment after R code
```

```
## [1] 4
```

On the first line, the hash is placed at the very beginning, so the entire line is treated as a comment. On the second line the hash is placed after the simple equation `2 + 2`. R runs the function and finds the answer `4`, but it ignores all of the words after the hash.

Different languages have different comment declaration characters. In LaTeX everything after the percent sign is treated as a comment, and in Markdown/HTML comments are placed inside of `<!- ->`. The hash character is used for comment declaration in command-line shell scripts as well as many other programming languages such as Python and Julia.

Nagler (1995, 491) gives some advice on when and how to use comments:

- write a comment before a block of code describing what the code does,
- comment on any line of code that is ambiguous.

In this book, I follow these guidelines when displaying code. Nagler also suggests that all of your source code files should begin with a comment header. At the least, the header should include:

- a description of what the file does,
- the date it was last updated,
- the name of the file's creator and any contributors.

You may also want to include other information in the header such as what files it depends on, what output files it produces, what version of the programming language you are using, sources that may have influenced the code, and how the code is licensed. Here is an example of a minimal file header for an R source code file that creates the third figure in an article titled 'My Article':

```
############################
# R Source code file used to create Figure 3 in 'My Article'
# Created by Christopher Gandrud
# MIT License
############################
```

Feel free to use things like the long series of hash marks above and below the header, white space, and indentations to make your comments more readable.

Style guides

In natural language writing you don't necessarily have to follow a style guide. People could probably figure out what you are trying to say, but it is a lot easier for your readers if you use consistent rules. The same is true when writing computer code. It's good to follow consistent rules for formatting your code so that it's easier for you and others to understand.

There are a number of R style guides. Most of them are similar to the Google R Style Guide.[3] Hadley Wickham also has a nicely presented R style guide.[4] You may want to use the *styler* (Müller and Walthert, 2019) package to automatically reformat your code so that it is easier to read.

Literate programming

For particularly important pieces of research code, it may be useful to not only comment on the source file, but also display code in presentation text. For example, you may want to include key parts of the code you used for your main statistical models and an explanation of this code in an appendix following your article. This is commonly referred to as literate programming (Knuth, 1992).

2.2.4 Explicitly tie your files together

If everything is just a text file, then research projects can be thought of as individual text files that have a relationship with one another. They are tied together. A data file is used as input for an analysis file. The results of an analysis are shown and discussed in a markup file that is used to create a PDF document. Researchers often do not explicitly document the relationships between files that they used in their research. For example, the results of an analysis–a table or figure–may be copied and pasted into a presentation document. It can be very difficult for future researchers to trace the table or figure back to a particular statistical model and a particular data set without clear documentation. Therefore, it is important to make the links between your files explicit.

Tie functions are the most dynamic way to explicitly link your files together. These functions instruct the computer program you are using to use information from another file. In Table 2.1, I have compiled a selection of key tie functions you will learn how to use in this book. We'll discuss many more, but these are some of the most important.

[3] See: `http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html`.
[4] You can find it at `http://adv-r.had.co.nz/Style.html`.

TABLE 2.1: A Selection of Functions/Packages/Programs for Tying Together Your Research Files

Function/Package/ Program	Language	Description	Chapters Discussed
knitr	R	R package with commands for tying analysis code into presentation documents including those written in LaTeX and Markdown.	Throughout
rmarkdown	R	R package that builds on *knitr*. It allows you to use Markdown to output to HTML, PDFs compiled with LaTeX or Microsoft Word.	Throughout
`download.file`	R	Downloads a file from the internet.	6
`read.table`	R	Reads a table into R. You can use this to import a plain-text file formatted data into R.	6
`read.csv`	R	Same as `read.table` with default arguments set to import `.csv` formatted data files.	6
`import`	R	Reads a table stored locally or on the internet into R. You can use it to import a wide variety of plain-text data formats into R from secure (https) URLs.	6
API-based packages	R	Various packages use APIs to gather data from the internet.	6
`merge`	R	Merges together data frames.	7
`source`	R	Runs an R source code file.	8
`source_url`	R	From the *devtools* package. Runs an R source code file from a secure (`https`) url like those used by GitHub.	8
`kable`	R	Creates tables from data frames that can be rendered using Markdown or LaTeX.	9
`toLaTeX`	R	Converts R objects to LaTeX.	2
`includegraphics`	LaTeX	Inserts a figure into a LaTeX document.	10
`include_graphics`	R/R Markdown	Inserts a figure into an R Markdown document.	10
`![]()`	Markdown	Inserts a figure into a Markdown document.	12
Pandoc	shell	A shell program for converting files from one markup language to another. Allows you to tie presentation documents together.	12
Make	shell	A shell program for automatically building many files at the same time.	6

2.2.5 Have a plan to organize, store, and make your files available

Finally, in order for independent researchers to reproduce your work, they need to be able access the files that instruct them how to do this. Files also need to be organized so that independent researchers can figure out how they fit together. So, from the beginning of your research process, you should have a plan for organizing your files and a way to make them accessible.

One rule of thumb for organizing your research in files is to limit the amount of content any one file has. Files that contain many different operations can be very difficult to navigate, even if they have detailed comments. For example, it would be very difficult to find any particular operation in a file that contained the code used to gather the data, run all of the statistical models, and create the results, figures and tables. If you have a hard time finding things in a file you created, think of the difficulties independent researchers will have!

Because we have so many ways to link files together, there is really no need to lump many different operations into one file. So, we can make our files modular. One source code file should be used to complete one or just a few tasks. Breaking your operations into discrete parts will also make it easier for you and others to find errors (Nagler, 1995, 490).

Chapter 4 discusses file organization in much more detail. Chapter 5 teaches you a number of ways to make your files accessible through the cloud computing services like GitHub.

3

Getting Started with R, RStudio, and knitr/R Markdown

If you have rarely or never used R before, the first section of this chapter gives you enough information to be able to get started and understand the R code I use throughout the book. For more detailed introductions on how to use R, please refer to the resources mentioned in Chapter 1 (Section 1.6). Experienced R users might want to skip the first section.

In the second section, I'll give a brief overview of RStudio. I highlight the key features of the main RStudio panel (what appears when you open RStudio) and some of its main tools for reproducible research. Finally, I discuss the basics of the *knitr* and *rmarkdown* packages, how to use them in R, and how they are integrated into RStudio.

3.1 Using R: The Basics

To get you started with reproducible research, we'll cover some very basic R syntax—the rules for talking to R. I cover key parts of the R language including:

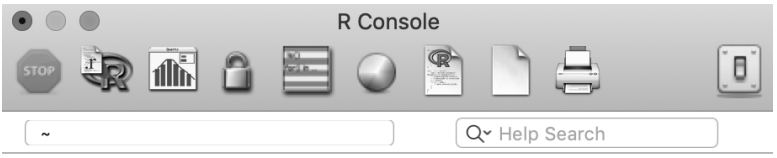
- objects and assignment,
- component selection,
- functions,
- arguments,
- the workspace and history,
- packages.

Before discussing each of these in detail, let's open R and look around.[1] When you open the R GUI program by clicking on the R icon, you should get a

[1] Please see Chapter 1 for instructions on how to install R.

window that looks something like Figure 3.1.[2] This window is the **R console**. Below the start-up information—information about what version of R you are using, license details, and so on—you should see a `>` (greater-than sign). This prompt is where you enter R code.[3] To run R code that you have typed after the prompt, press the `Return` or `Enter` key.

Now that we have a new R session open, we can get started.




```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

[R.app GUI 1.70 (7543) x86_64-apple-darwin15.6.0]

[History restored from /Users/cgandrud/.Rapp.history]

>
```

FIGURE 3.1: R Console at Startup

3.1.1 Objects

If you've read a description of the R language before, you will probably have seen it referred to as an 'object-oriented language'. What are objects? Objects

[2] This figure and almost all screenshots in this book were taken on a computer using the macOS 10.14 operating system.

[3] If you are using a Unix-like system such as Linux Ubuntu or macOS, you can also access R via an application called the Terminal. If you have installed R on your computer, you can type `R` into the Terminal. This will begin a new R session. You will know you are in a new R session because the same type of start-up information as in Figure 3.1 will be printed in your Terminal.

are like the R language's nouns. They are things, like a vector of numbers, a data set, a word, a table of results from some analysis, and so on. Saying that R is object-oriented means that R is focused on doing actions to objects. We will talk about the actions, functions, later in this section.[4] Now let's create a few objects.

Numeric and string objects

Objects can have a number of different types. Let's make two simple objects. The first is a numeric-type object. The other is a character object.

We can choose almost any name we want for our objects as long as it begins with an alphabetic character and does not contain spaces.[5] Just because there are relatively few hard restrictions on object names, doesn't mean that you should name your object anything. Your code will be much easier to read if object names are short and meaningful. Give each object a unique name to avoid confusion and conflicts. For example, if you reuse an object name in an R session, you could easily accidentally overwrite it.

Let's begin working with numeric objects by creating a new object called *number* with the number 10 in it. Use the assignment operator[6] (`<-`) to put something into the object:

```
number <- 10
```

To see the contents of our object, type its name into the R console.

```
number
```

```
## [1] 10
```

Let's briefly breakdown this output. `10` is clearly the contents of *number*. The double hash (`##`) is included here to tell you that this is output rather than R code.[7] If you run functions in your R console, you will not get the double hash

[4] Functions are also objects. In this chapter, I treat them as distinct from other object types to avoid confusion.

[5] Wickham (2014a) argues that underscores (`_`) should be used to separate words in object names to make the names easier to read. For example: `health_data` rather than `healthdata`. The underscore object naming convention appears to now be the dominant style in the R community. There are other conventions. These include using periods (`.`) or capital letters (referred to as CamelBack) to separate words. For more information on R naming conventions, see Bååth (2012).

[6] The assignment operator is sometimes also referred to as the 'gets arrow'.

[7] The double hash is generated automatically by *knitr*. Prepending the output with hashes makes it easier to copy and paste code into R from a document created by *knitr*/*rmarkdown* because R will ignore everything after a hash.

in your output. Finally, `[1]` gives the position in the object that the number 10 is on. Our object only has one position.

Creating an object with words and other characters, a character object, is very similar. The only difference is that you enclose the character string (letters in a word for example) inside of single or double quotation marks (`''`, or `""`).[8] Let's create an object called *words* containing the character string `Hello World`:

```
words <- "Hello World"
```

An object's type is important to keep in mind. It determines what we can do to the object. For example, you cannot take the mean of a character object like the *words* object:

```
mean(words)
```

```
## Warning in mean.default(words): argument is not numeric
## or logical: returning NA
```

```
## [1] NA
```

Trying to find the mean of our *words* object gives us a warning message and returns the value NA: not applicable. You can also think of `NA` as meaning "missing". To find out an object's type, use the `class()` function.[9] For example:

```
class(words)
```

```
## [1] "character"
```

Vector and data frame objects

So far, we have only looked at objects with a single number or character string.[10] Clearly we often want to use objects that have many strings and numbers. In R these are usually data frame-type objects and are roughly equivalent to the data structures you would be familiar with from using a program such as Microsoft Excel. We will be using data frames extensively

[8] Single and double quotation marks are interchangeable in R for this purpose. In this book I always use double quotes, except for *knitr* code chunk options.

[9] R object types are not fixed. They can be implicitly converted by assigning values of a different type to them. Other languages, such as Scala, prohibit implicit type conversions. These languages are sometimes referred to as 'type safe'. They make it impossible to implicitly change an object's type, which can sometimes produce errors.

[10] These might be called scalar objects, though in R, scalars are just vectors with a length of 1.

throughout the book. Before looking at data frames it is useful to first look at the simpler objects that make up data frames. These are called vectors. Vectors are R's "workhorse" (Matloff, 2011). Knowing how to use vectors will be especially helpful when you clean up raw data in Chapter 7 and make tables in Chapter 9.[11]

Vectors

Vectors are the "fundamental data type" in R (Matloff, 2011). They are an ordered group of numbers, character strings, and so on.[12] It may be useful to think of most data in R as composed of vectors. For example, data frames are basically collections of vectors of the same length, i.e. they have the same number of rows, attached together to form columns.

Let's create a simple numeric vector containing the numbers 2.8, 2, and 14.8. To do this, we will use the `c()` (combine) function and separate the numbers with commas (`,`):

```
numeric_vector <- c(2.8, 2, 14.8)

# Show numeric_vector's contents
numeric_vector
```

```
## [1]  2.8  2.0 14.8
```

Vectors of character strings are created in a similar way. The only difference is that each character string is enclosed in quotation marks like this:

```
character_vector <- c("Albania", "Botswana", "Cambodia")

# Show character_vector's contents
character_vector
```

```
## [1] "Albania"  "Botswana" "Cambodia"
```

Matrices

To give you a preview of what we are going to do when we start working with real data sets, let's combine the two vectors *numeric_vector* and *character_vector*

[11] If you want information about other types of R objects such as lists and matrices, Chapter 1 of Norman Matloff's (2011) book is a really good place to look.

[12] In a vector, every member of the group must be of the same type. If you want an ordered group of values with different types, you can use lists.

into a new object with the `cbind()` function. This function binds the two vectors together side-by-side as columns.[13]

```
string_num_matrix <- cbind(character_vector, numeric_vector)

string_num_matrix
```

```
##      character_vector numeric_vector
## [1,] "Albania"        "2.8"
## [2,] "Botswana"       "2"
## [3,] "Cambodia"       "14.8"
```

By binding these two objects together, we've created a new matrix object.[14] You can see that the numbers in the **numeric_vector** column are between quotation marks. Matrices, like vectors, can only have one data type, so R has converted the numbers to strings.

Data frames

If we want to have an object with rows and columns and allow the columns to contain data with different types, we need to use data frames. Let's use the `data.frame` function to combine the *numeric_vector* and *character_vector* objects.

```
string_num_df <- data.frame(character_vector, numeric_vector)

string_num_df
```

```
##   character_vector numeric_vector
## 1          Albania            2.8
## 2         Botswana            2.0
## 3         Cambodia           14.8
```

In this output, you can see the data frame's *names* attribute.[15] It is the column names. You can use the `names()` function to see any data frame's names:[16]

```
names(string_num_df)
```

[13] If you want to combine objects as if they were rows of the same column(s), use the `rbind()` function.

[14] Matrices are basically collections of vectors, each represented as a column.

[15] Matrices can also have a *names* attribute.

[16] You can also use `names()` to assign names for the entire data frame. For example, `names(string_num_df) <- c(variable_1, variable_2)`

```
## [1] "character_vector" "numeric_vector"
```

You will also notice that the first column of the data set has no name and is a series of numbers. This is the *row.names* attribute. Data frame rows can be given any name as long as each row name is unique. We can use the `row.names()` function to set the row names from a vector. For example,

```
# Reassign row.names
row.names(string_num_df) <- c("First", "Second", "Third")

# Display new row.names
row.names(string_num_df)
```

```
## [1] "First"  "Second" "Third"
```

You can see in this example how `row.names()` can also be used to print the row names.[17] The *row.names* attribute does not behave like a regular data frame column. You cannot, for example, include it as a variable in a regression. You can use the `row.names()` function to assign the *row.names* values to a regular column.

You will notice in the output for *string_num_df* that the strings in the **character_vector** column are not in quotation marks. This does not mean that they are now numeric data. To prove this, try to find the mean of **character_vector** by running it through the `mean()` function:

```
mean(string_num_df$character_vector)
```

```
## Warning in
## mean.default(string_num_df$character_vector): argument
## is not numeric or logical: returning NA

## [1] NA
```

Component selection

The last bit of code we just saw will probably be confusing. Why do we have a dollar sign (`$`) between the name of our data frame object name and the `character_vector` variable? The dollar sign is called the component selector. It's also sometimes called the element name operator. Either way, it extracts a part, component, of an object. In the previous example, it extracted the **character_vector** column from the *string_num_df* so that it could be fed to the `mean()` function.

[17] Note that this is really only useful for data frames with few rows.

We can use the component selector to create new objects with parts of other objects. Imagine that we have *string_num_df* and want an object with only the information in the **numeric_vector** column. Let's use the following code:

```
# Extract a numeric vector from string_num_df
numeric_extract <- string_num_df$numeric_vector

numeric_extract
```

```
## [1]  2.8  2.0 14.8
```

Knowing how to use the component selector will be especially useful when we discuss making tables for presentation documents in Chapter 9.

`attach()` and `with()`

Using the component selector can create long repetitive code if you want to select many components. You have to write the object name, a dollar sign, and the component name every time you want to select a component. You can streamline your code by using functions such as `attach()` and `with()`.

`attach()` attaches a database to R's search path.[18] R will then search the database for variables you specify. You don't need to use the component selector to tell R again to look in a particular data frame after you have attached it. For example, let's attach the *cars* data that comes with R. It has two variables, **speed** and **dist**.[19]

```
# Attach cars to search path
attach(cars)

# Display speed
head(speed)
```

```
## [1] 4 4 7 7 8 9
```

```
# Display dist
head(dist)
```

```
## [1]  2 10  4 22 16 10
```

We used the `head()` function to see just the first few values of each variable.

[18] You can see what is in your current search path with the `search` function. Just type `search()` into your R console.

[19] For more information on this data set, type `?cars` into your R console.

Now that we are done working with the *cars* data set, we should `detach()` it. Not doing so could confuse R later in our session.

```
detach(cars)
```

A safer alternative to `attach()` is `with()`. It more clearly delineates when to draw from inside a particular object. For example, we can find the mean of **numeric_vector** `with()` the *string_num_df* data frame:

```
with(string_num_df, {
        mean(numeric_vector)
    }
)
```

```
## [1] 6.533
```

In the `with()` call the data frame object goes first and then the `mean()` function[20] goes second in curly brackets (`{}`).

In this book I avoid using the `attach()` and `with()` functions. Instead, I use the component selector. Though it creates longer code, I find that code written with the component selector is less ambiguous. It's always clear which object we are selecting a component from.

Subscripts

Another way to select parts of an object is to use subscripts. You have already seen subscripts in the output from our examples so far. They are denoted with square braces (`[]`). We can use subscripts to select not only columns from data frames but also rows and individual values. As we began to see in some of the previous output, each part of a data frame has an address captured by its row and column number. We can tell R to find a part of an object by putting the row number/name, column number/name, or both in square braces. The first part denotes the rows and separated by a comma (`,`) are the columns.

To give you an idea of how this works, let's use the *cars* data set again. Use `head()` to get a sense of what this data looks like.

```
head(cars)
```

```
##   speed dist
```

[20] Using R terminology, the second 'argument' value, the code after the comma, of the `with()` function is called an 'expression', because it can contain more than one R function or statement. See Section 3.1.2 for a more comprehensive discussion of R function arguments.

```
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

We can see a data frame with information on various car speeds (**speed**) and stopping distances (**dist**). If we want to select only the third through seventh rows, we can use the following subscript function call:

```
cars[3:7, ]
```

```
##   speed dist
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
## 7    10   18
```

The colon (`:`) creates a sequence of whole numbers from 3 to 7. To select the fourth row of the **dist** column, we can type:

```
cars[4, 2]
```

```
## [1] 22
```

An equivalent way to do this is:

```
cars[4, "dist"]
```

```
## [1] 22
```

Finally, we can even include a vector of column names to select:

```
cars[4, c("speed", "dist")]
```

```
##   speed dist
## 4     7   22
```

3.1.2 Functions

If objects are the nouns of the R language, functions are the verbs. They do things to objects. Let's use the `mean` function as an example. This function

takes the mean of a numeric vector object. Remember our *numeric_vector* object from before:

```
numeric_vector
```

```
## [1]  2.8  2.0 14.8
```

To find the mean of this object, type:

```
mean(x = numeric_vector)
```

```
## [1] 6.533
```

We use the assignment operator to place a function's output into an object. For example:

```
numeric_vector_mean <- mean(x = numeric_vector)
```

Notice that we typed the function's name then enclosed the object name in parentheses immediately afterwards. This is the basic syntax that all functions use, i.e. `FUNCTION(ARGUMENTS)`. Even if you don't want to explicitly include an argument, *you still need to type the parentheses after the function.*[21]

Arguments

Arguments modify what functions do. In our most recent example, we gave the `mean` function one argument (`x = numeric_vector`) telling it that we wanted to find the mean of *numeric_vector*. Arguments use the `ARGUMENT_LABEL = VALUE` syntax.[22] In this case, `x` is the argument label.

To find all of the arguments that a function can accept, look at the **Arguments** section of the function's help file. To access the help file, type: `?FUNCTION`. For example:

```
?mean
```

The help file will also tell you the default values that the arguments are set to. You do not need to explicitly set an argument if you want to use its default value.

[21] If you don't include the parentheses after the function name, R will return the source code for the function just like when you enter an object name into your console returns the contents. This is because in R, functions are actually also objects!

[22] Note: you do not have to put spaces between the argument label and the equals sign or the equals sign and the value. However, having spaces can make your code easier to read.

You do need to be fairly precise with the syntax for your argument's values. Values for logical arguments must be written as `TRUE` or `FALSE`.[23] Arguments that accept character strings require quotation marks.

Let's see how to use multiple arguments with the `round()` function. This function rounds a vector of numbers. We can use the `digits` argument to specify how many decimal places we want the numbers rounded to. To round the object *numeric_vector_mean* to one decimal place, type:

```
round(x = numeric_vector_mean, digits = 1)
```

```
## [1] 6.5
```

Note that *arguments are always separated by commas.*

Some arguments do not need to be explicitly labeled. For example, we could write:

```
# Find mean of numeric_vector
mean(numeric_vector)
```

```
## [1] 6.533
```

R will do its best to figure out what you want and will only give up when it can't. This will generate an error message. However, to avoid any misunderstandings between yourself and R, it is good practice to label your argument values. This will also make your code easier for other people to read, i.e. it will be more reproducible.

You can stack functions inside of arguments. For example, have R find the mean of *numeric_vector* and round it to one decimal place:

```
round(mean(numeric_vector), digits = 1)
```

```
## [1] 6.5
```

Stacking functions inside of each other can create code that is difficult to read. Another option that potentially makes more easily understandable code is piping using the pipe function (`%>%`) that you can access from the *magrittr* (Bache and Wickham, 2014) or *dplyr* (Wickham et al., 2019b) packages. The basic idea behind the pipe function is that the output of one function is set as the first argument of the next. For example, to find the mean of *numeric_vector* and then round it to one decimal place use:

[23] They can be abbreviated `T` and `F`.

```
# Load magrittr package
library(magrittr)

# Find mean of numeric_vector and round to 1 decimal place
mean(numeric_vector) %>%
    round(digits = 1)
```

```
## [1] 6.5
```

3.1.3 The workspace and history

All of the objects you create become part of your workspace, alternatively known as the current working environment. Use the `ls()` function to list all of the objects in your current workspace.[24]

```
ls()
```

```
##  [1] "character_vector"    "number"
##  [3] "numeric_extract"     "numeric_vector"
##  [5] "numeric_vector_mean" "pkg_to_install"
##  [7] "si"                  "string_num_df"
##  [9] "string_num_matrix"   "words"
```

You can remove specific objects from the workspace using the `rm()` function. For example, to remove the objects `character_vector` and `words` type:

```
rm(character_vector, words)
```

To save the entire workspace into a binary, not plain-text, RData file use `save.image()`. The main argument of `save.image()` is the location and name of the file in which you want to save the workspace. If you don't specify the file path it will be saved into your current working directory (see Chapter 4 for information on files paths and working directories). To save the current workspace in a file called *workspace-2019-12-22.RData* in the current working directory type:

```
save.image(file = "workspace-2019-12-22.RData")
```

Use `load()` to load a saved workspace back into R:

[24] Note: your workspace will probably include different objects than this example. These are objects created to knit the book.

```
load(file = "workspace-2019-10-22.RData")
```

You should generally avoid having R automatically save your workspace when you quit and reload it when you start R again. Instead, when you return to a project, rerun the source code files. This avoids any complications caused when you use an object in your workspace that is left over from running an older version of the source code.[25] In general, I also recommend against saving data in binary RData formatted files. They are not text files. They are not human readable. They are much less future-proof.

One of the few times when saving your workspace is useful is when it includes an object that was computationally difficult and took a long time to create. In this case, you can save only the large object with `save()`.[26] For example, if we have a very large object called *model-output*, we can save it to a file called *model-output.RData* like this:

```
save(model-output, file = "model-output.RData")
```

3.1.4 R history

When you execute code in the R console, it becomes part of your history. Use the `history()` function to see the most recent functions in your history, You can also use the up and down arrows on your keyboard when your cursor is in the R console to scroll through your history.

3.1.5 Global R options

In R you can set global options with `options()`. This lets you set how R runs and outputs functions through an entire R session. For example, to have output rounded to one decimal place, set the `digits` argument:

```
options(digits = 1)
```

[25] For example, imagine you create an object, then change the source code you used to create the object. However, there is a syntax error in the new version of the source code. The old object won't be overwritten, and you will be mistakenly using the old object in future functions.

[26] `save.image()` is just a special case of `save()`.

3.1.6 Installing new packages and loading functions

Functions are stored in R packages. The functions we have used so far were loaded automatically by default. One of the great things about R is the many user-created packages[27] that expand the number of functions we can use. To install functions that do not come with the basic R installation, you need to install the add-on packages that contain them. To do this, use the `install.packages()` function. By default, this function downloads and installs the packages from the Comprehensive R Archive Network (CRAN).

When you install a package, you will likely be given a list of "mirrors" from which you can download the package. Select the mirror closest to you.

Once you have installed a package, you need to load when you want to use its functions. Use the `library()` function to load a package.[28] For example, the following code loads the popular *ggplot2* plotting package:

```
library(ggplot2)
```

Please note that for the examples in this book I only specify what package a function is from if it is not loaded by default when you start an R session.

Finally, if you want to make sure R uses a function from a specific package, you can use the double-colon operator (`::`). For example, to make sure that we use the `qplot()` function from the *ggplot2* package, we type:

```
ggplot2::qplot(. . .)
```

Using the double-colon ensures that R will use the function from the particular package you want and makes it clear to a source code reader what package a function comes from. If you use the double-colon, you don't need to include `library()` beforehand. Note that it does not load all of the functions in the package, just the one you ask for.

3.2 Using RStudio

As I mentioned in Chapter 1, RStudio is an integrated development environment for R. It provides a centralized and well-organized place to do almost anything

[27] For the latest list, see: `http://cran.r-project.org/web/packages/available_packages_by_name.html`.

[28] You will probably see R packages referred to as "libraries", though this is a misnomer.

you want to do with R. As we will see later in this chapter, it is especially well integrated with literate programming tools for reproducible research. Right now, let's take a quick tour of the basic RStudio window.

The default window

When you first open RStudio, you should see a default window that looks like Figure 3.2. In this figure, you see three window panes. The large one on the left is the *Console/Terminal/Jobs* pane. The *Console* pane is an R console and functions exactly the same as the console discussed so far in this chapter. *Terminal* is a command-line terminal where you can run command-line tools like those we discuss in Chapter 4. The *Jobs* pane allows you to run R scripts in the background. This is very useful if you have computationally time consuming jobs that you would like to run while also doing other work in RStudio.

The *Environment/History/Connections* panes are in the upper right-hand corner. The *Environment* pane shows you all of the objects in your workspace and some of their characteristics, like how many observations a data frame has. You can click on an object in this pane to see a preview of its contents. This is especially useful for quickly looking at a data set in much the same way that you can visually scan a Microsoft Excel spreadsheet. The *History* pane records all of the functions you have run. It also allows you to rerun code and insert it into a source code file. The *Connections* pane allows you to manage connections to databases such as an SQL server.

In the lower right-hand corner, you will see the *Files/Plots/Packages/Help/Viewer* panes. We will discuss the *Files* pane in more detail in Chapter 4. Basically, it allows you to navigate and organize your files. The *Plots* pane is where figures you create in R appear. This pane allows you to see all of the figures you have created in a session using the right and left arrow icons. It also lets you copy and save the figures in a variety of formats. The *Packages* pane shows the packages you have installed, allows you to load individual packages by clicking on the dialog box next to them, access their help files (just click on the package name), update the packages, and even install new packages. The *Help* pane shows you help files. You can search for help files and search within help files using this pane. Finally, the *Viewer* pane allows you to view local web content like JavaScript graphics and Shiny apps.

The Source pane

There is an important pane that does not show up when you open RStudio for the first time. This is the *Source* pane. The *Source* pane is where you create, edit, and run your source code files. It also functions as an editor for your markup files. It is the center of reproducible research in RStudio.

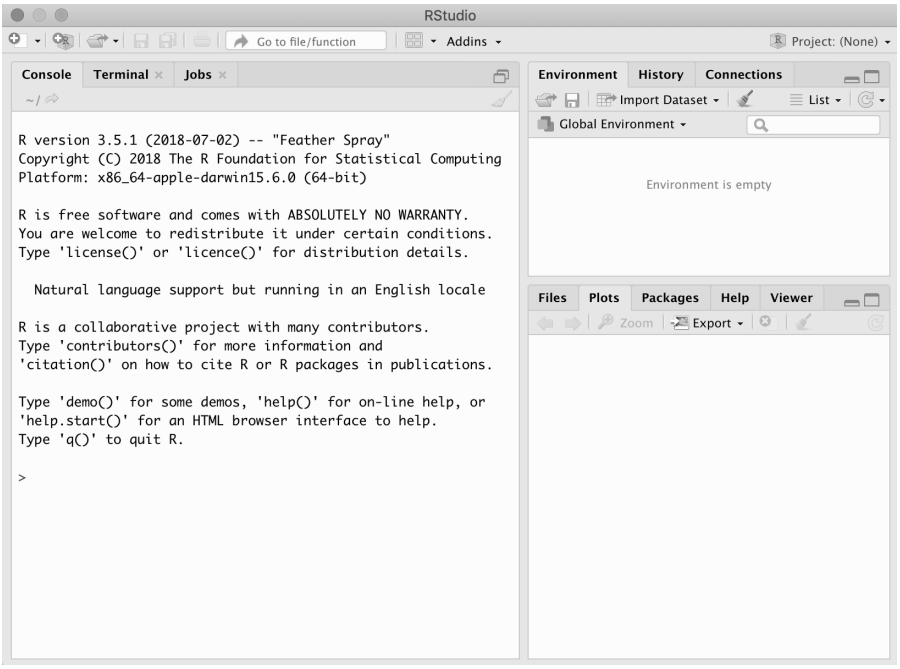


FIGURE 3.2: RStudio at Startup

Let's first look at how to use the *Source* pane with regular R files. We will then cover how it works with *knitr/rmarkdown* in more detail in the next section.

R source code files have the file extension `.R`. When you create a new source code document, RStudio will open a new *Source* pane. Do this by going to the menu bar and clicking on `File New`. In the `New` drop-down menu, you have the option to create a variety of different source code documents. Select the `R Script` option. You should now see a new pane with a bar across the top that looks like Figure 3.3. To run the R code, you have in your source code file highlight it[29] and click the `Run` icon on the top bar. This sends the code to the console where it is run. The icon to the right of `Run` runs the code above where you have highlighted. The `Source` icon next to this runs all of the code in the file using R's `source()` function. When you click on the last icon on the right (it has a series of stacked lines) you will get a navigable table of contents for your file; very useful for working with longer documents, especially markup documents.

[29] If you are only running one line of code, you don't need to highlight the code; you can put your cursor on that line.

FIGURE 3.3: RStudio Source Code Pane Top Bar

3.3 Using knitr and R Markdown: The Basics

To get started with knitr and R Markdown in R or RStudio, we need to learn some of the basic concepts and syntax. The concepts are the same regardless of the markup language we are knitting R code with, but much of the syntax varies by markup language. *rmarkdown* relies on *knitr* and a utility called *Pandoc* to create many different types of presentation documents (HTML, PDF, or MS Word) from one document written largely using knitr's R Markdown syntax.

3.3.1 What *knitr* does

Let's take a quick, abstract look at what the *knitr* package does. As I've mentioned, *knitr* ties together your presentation of results with the creation of those results. The *knitr* process takes three steps (see Figure 3.4). First, we create a knittable markup document. This contains both the analysis code and the presentation document's markup which is the text and rules for how to format the text. *knitr* then *knits*: i.e. it runs the analysis code and converts the output into the markup language you are using according to the rules that you tell it to use. It inserts the marked up results into a document that only contains markup for the presentation document. You *compile* this markup document as you would if you hadn't used *knitr* into your final PDF document or webpage presenting your results.

3.3.2 What *rmarkdown* does

The *rmarkdown* package implements a variation on this process that utilizes a program called Pandoc to create presentation documents in multiple formats from a knittable document written in Markdown. The main difference between pure *knitr* markdown and *rmarkdown* documents is the inclusion of a header specifying how you want to render the document with Pandoc.[30]

The header is written in YAML.[31] The YAML header can include information such as the document's title, author, whether or not to include a table of

[30] Note that you can also create an *rmarkdown* document without a header. *rmarkdown* will just use the default settings when knitting.

[31] YAML is a recursive acronym that means, "YAML Ain't Markup Language".

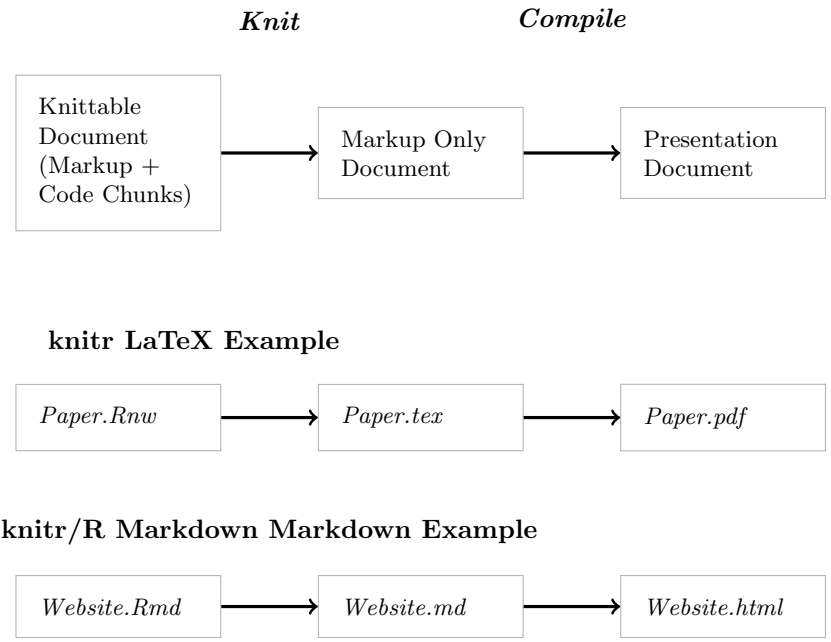


FIGURE 3.4: Knitr/R Markdown Process

contents, and a link to a BibTeX bibliography file. YAML is a straightforward data format that organizes information in a simple hierarchy. The header begins and ends with three dashes (`---`). Information keys–like "title" and "author"–are separated from their associated "values" by a colon (`:`). Sub-values of a hierarchy are denoted by being placed on a new line and indented.[32] Here is a basic R Markdown header that indicates the document's title, author, and date, and that it will be turned into a PDF document (via LaTeX).

```
---
title: "A Basic PDF Presentation Document"
author: "Christopher Gandrud"
date: "2019-12-28"
output: pdf_document:
    toc: true
---
```

The title, author, and date will be placed at the beginning of the output document. The final line (`toc: true`) creates a table of contents near the

[32] It doesn't matter how many spaces you use to indent, as long as all indentations have the same number of spaces.

beginning of the PDF document when we knit it. We will discuss more header options in Chapter 12.

RStudio can automatically create a basic header for the type of output document that you want when you open a new R Markdown file. Simply select `File` then `R Markdown...`. A window will appear that looks like Figure 3.5. In this window select the type of output document you want to create and click `Ok`.

In addition to the header, R Markdown differs from basic knitr files in that you can include Pandoc syntax in your R Markdown document. This can be useful for bibliographies as we will discuss in Chapter 12. Nonetheless, remember that apart from the header and ability to include Pandoc syntax, at the simplest level R Markdown documents are knitr documents written in R Markdown syntax. They have the same code chunk syntax, as we will see shortly.

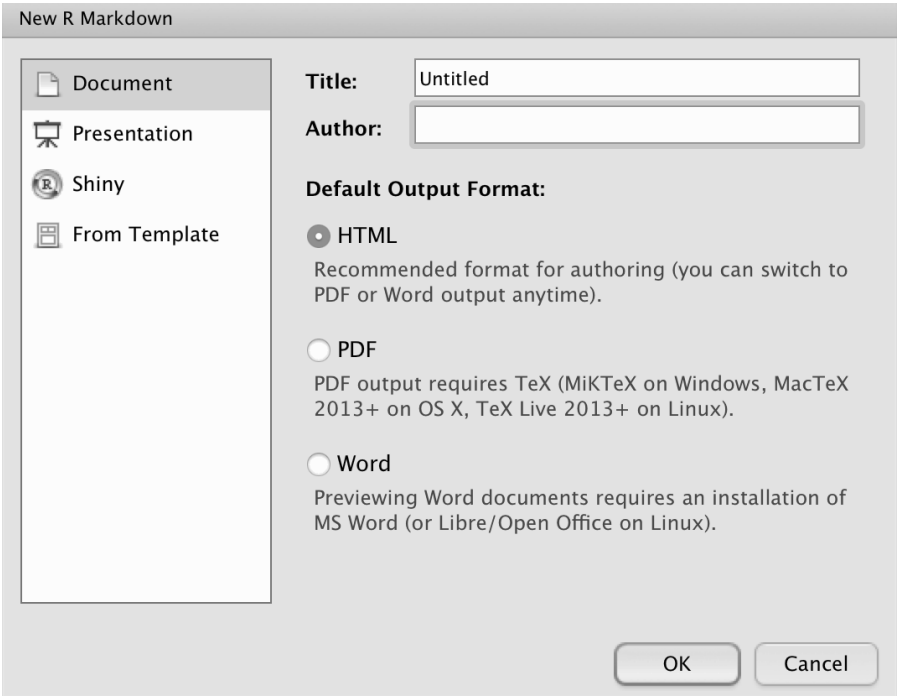


FIGURE 3.5: The New R Markdown Options Window

3.3.3 File extensions

When you save a knittable file, use a file extension that indicates (a) that it is knittable and (b) what markup language it is using. You can use a number of file extensions for R Markdown files including: `.Rmd` and `.Rmarkdown`.[33] LaTeX documents that include *knitr* code chunks are generally called R Sweave files and have the file extension `.Rnw`. This terminology is a little confusing.[34] It is a holdover from *knitr*'s main literate programming predecessor *Sweave*. Note that *rmarkdown* documents can compile to LaTeX PDF documents and support pretty much the full capabilities of LaTeX. Because markdown is generally easier to write than raw LaTeX, `.Rnw` markup is much less commonly used. For example, I converted the third edition of this book from `.Rnw` to `.Rmd`.

3.3.4 Code chunks

Use code chunks to include knittable R code into your markup presentation documents. Code chunk syntax differs depending on the markup language we are using to write our documents. Let's see the syntax for R Markdown and R LaTeX files. If you are unfamiliar with basic LaTeX or Markdown syntax, you might want to skim Chapters 11 and 12 to familiarize yourself with it before reading this section.

R Markdown

In R Markdown files, we begin a code chunk by writing the head: ```` ```{r} ````. A code chunk is closed, ended, with: ```` ``` ````. For example:

````
```{r}
# Example of an R Markdown code chunk
string_num_matrix <- cbind(character_vector, numeric_vector)
```
````

The R Markdown code chunk syntax is exactly the same for markdown files you compile with *knitr* or *rmarkdown*.

[33] R Markdown files that you compile with *knitr* or *rmarkdown* have the same `.Rmd` file extension.

[34] The "nw" refers to the noweb simple literate programming tool that Sweave built on (Leisch, 2002; Ramsey, 2011).

R LaTeX (.Rnw)

Code chunks are delimited in non-R Markdown R LaTeX documents in a way that emulates the long-established *Sweave* syntax. Sweave-style code chunks begin with the following head: `<<>>=`. The code chunk is closed with an at sign (`@`).

```
<< >>=
string_num_matrix <- cbind(character_vector, numeric_vector)
@
```

Code chunk labels

Each chunk has a label. When a code chunk creates a plot or the output is cached, stored for future use, *knitr* uses the chunk label for the new file's name. If you do not explicitly give the chunk a label it will be assigned one like: `unnamed-chunk-1`.

To explicitly assign chunk labels in R Markdown documents, place the label name inside of the braces after the `r`. If we wanted to use the, admittedly not descriptive, label `ex-label` we type:

````
```{r ex-label}
# Example chunk label
```
````

The same general format applies to the two types of LaTeX chunks. In Sweave-style chunks, we type: `<<ex-label>>=`. Try not to use spaces or periods in your label names. Also remember that chunk labels *must* be unique.

Code chunk options

There are many times when we want to change how our code chunks are knitted and presented. Maybe we only want to show the code and not the results. Perhaps we don't want to show the code at all but just a figure that it produces. Maybe we want the figure to be formatted on a page in a certain way. To make these changes and many others, we can specify code chunk options.

Like chunk labels, you specify options in the chunk head. Place them after the chunk label, separated by a comma. Chunk options are written following pretty much the same rules as regular R function arguments. They have a similar `OPTION_LABEL=VALUE` structure as arguments. The option values must be written in the same way that argument values are. Character strings need to be inside of quotation marks. The logical `TRUE` and `FALSE` operators cannot be

written `"true"` and `"false"`. For example, imagine we have a Markdown code chunk called `ex-label`. If we want to run the code chunk, but not show the code in the final presentation document, we can use the option `echo=FALSE`.

````
```{r ex-label, echo=FALSE}
string_num_matrix <- cbind(character_vector, numeric_vector)
```
````

Note that all labels and code chunk options must be on the same line. Options are separated by commas. The syntax for *knitr* options is the same regardless of the markup language.

Throughout this book, we will look at a number of different code chunk options. Many of the chunk options we will use in this book are listed in Table 3.1. For the full list of *knitr* options, see the *knitr* chunk options page maintained by *knitr*'s creator Yihui Xie: `http://yihui.name/knitr/options`.

3.3.5 Global chunk options

So far, we have only looked at how to set local options in *knitr* code chunks, i.e. options for only one specific chunk. If we want an option to apply to all of the chunks in our document, we can set global chunk options. Options are 'global' in the sense that they apply to the entire document. Setting global chunk options helps us create documents that are formatted consistently without having to repetitively specify the same option every time we create a new code chunk. For example, rather than using the `fig.align='center'` option in each code chunk that creates a figure, we can center align all figures in a document by setting the option globally.

To set a global option, first create a new code chunk at the beginning of your document.[35] You will probably want to set the option `include=FALSE` so that *knitr* doesn't include the code in your presentation document. Inside the code chunk, use `opts_chunk$set`. You can set any chunk option as an argument to `opts_chunk$set`. The option will be applied across your document, unless you set a different local option.

Here is an example of how you can center align all of the figures in R Markdown in a chunk placed near the beginning of the document:

````
```{r set-global, include=FALSE}
# Center align all knitr generated figures
```
````

[35] In Markdown, you can put global chunk options at the very top of the document. In `.Rnw` documents, they should be placed after the `\begin{document}` function. See Chapter 11 for more information on how LaTeX documents are structured.

TABLE 3.1: A Selection of *knitr* Code Chunk Options

Chunk Option Label	Type	Description
`cache`	Logical	Whether or not to save results from the code chunk in a cache database. Note: cached chunks are only run when they are changed.
`cache.vars`	Character Vector	Specify the variable names to save in the cache database.
`eval`	Logical	Whether or not to run the chunk.
`echo`	Logical	Whether or not to include the code in the presentation document.
`error`	Logical	Whether or not to include error messages.
`engine`	Character	Set the programming language for *knitr* to evaluate the code chunk with.
`fig.align`	Character	Align figures. (Note: does not work with R Markdown documents.)
`fig.path`	Character	Set the directory where figures will be saved.
`include`	Logical	When `include=FALSE` the chunk is evaluated, but the results are not included in the presentation document.
`message`	Logical	Whether or not to include R messages.
`out.height`	Numeric	Set figures' heights in the presentation document.
`out.width`	Numeric	Set figures' widths in the presentation document.
`results`	Character	How to include results in the presentation document.
`tidy`	Logical	Whether or not to have *knitr* format printed code chunks.
`warning`	Logical	Whether or not to include warnings.

These functions are discussed in more detail in Chapter 8.

```
knitr::opts_chunk$set(fig.align='center')
```

If you want to use `opts_chunk` in a document rendered with *rmarkdown*, you will need to either explicitly call it as in the example using the double colon or load the *knitr* package before calling it.

3.3.6 *knitr* package options

knitr package options affect how the package itself runs. For example, the `progress` option can be set as either `TRUE` or `FALSE` [36] depending on whether or not you want a progress bar to be displayed when you knit a code chunk. You can use `base.dir` to set the directory where you want all of your figures to be saved (see Chapter 4).

You set package options in a similar way as global chunk options with `opts_knitr$set`. For example, include this code at the beginning of a document to turn off the progress bar when it is knitted:

```
```{r set-pkg-opt, include=FALSE}
# Don't show progress bars
knitr::opts_knit$set(progress=FALSE)
```
```

3.3.7 Hooks

You can also set hooks. Hooks come in two types: chunk hooks and output hooks. Chunk hooks run a function before or after a code chunk. Output hooks change how the raw output is formatted. I don't cover hooks in much detail in this book. For more information on hooks, please see Yihui Xie's webpage: `http://yihui.name/knitr/hooks`.

3.3.8 knitr, R Markdown, and RStudio

RStudio is highly integrated with knitr/R Markdown and the markup languages that they work with. RStudio is probably the easiest tool for creating and compiling knitr/R Markdown. Most of the RStudio/knitr/R Markdown features are accessed in the *Source* pane. The *Source* pane's appearance and capabilities change depending on the type of file you have open in it. RStudio uses a file's

[36] It's set as `TRUE` by default.

extension and, if it is an *rmarkdown* document, its header, to determine what type of file you have open.[37] We have already seen some of the features the *Source* pane has for R source code files. Let's now look at how to use *knitr* and *rmarkdown* with R source code files as well as the markup formats we cover in this book: R Markdown and R LaTeX.

Compiling R source code Notebooks

If you want a quick, well-formatted account of the code that you ran and the results that you got you can use RStudio's "Compile Notebook" capabilities. RStudio uses *rmarkdown* to create a standalone file presenting your source code and results. It will include all of the code from an R source file as well as the output. This can be useful for quickly presenting the steps you took to do an analysis. You can see an example RStudio Notebook in Figure 3.6.

If you want to create a Notebook from an open R source code file click the `Compile Notebook` icon ( ) in the *Source* pane's top bar.[38] Then in the window that pops up select the output type you would like (HTML, PDF or MS Word) and click the `Compile` button. For this example I selected HTML. In Figure 3.6 you can see near the top center right a small globe icon next to the word "Publish". Clicking this allows you to publish your Notebook to RPubs (`http://www.rpubs.com/`). RPubs is a site for sharing your Notebooks over the internet. You can publish not only Notebooks, but also any R Markdown document you compile in RStudio.

In this chapter's appendix we discuss interactive Jupyter notebooks. They are popular in the data science and tech industries and use a somewhat different logic from R Markdown notebooks.

My current tech team tends to use either Jupyter notebooks or R Markdown Notebooks to present our detailed analyses. We host and share these via GitHub. GitHub compiles both document types nicely for online access. For R Markdown Notebooks, use `output: github_document` in the header to ensure that the output file is compiled properly on GitHub.

R Markdown

Figure 3.7 is what the *Source* pane looks like when you have an R Markdown file open. You'll notice the familiar `Run` button for running R code. It now includes a drop-down menu for running code chunks. It includes options like `Run Current Chunk`, i.e. run the chunk where your cursor is located, `Run Next`

[37] You can manually set how you want the *Source* pane to act by selecting the file type using the drop-down menu in the lower right-hand corner of the *Source* pane.

[38] Alternatively, `File Compile Notebook...`

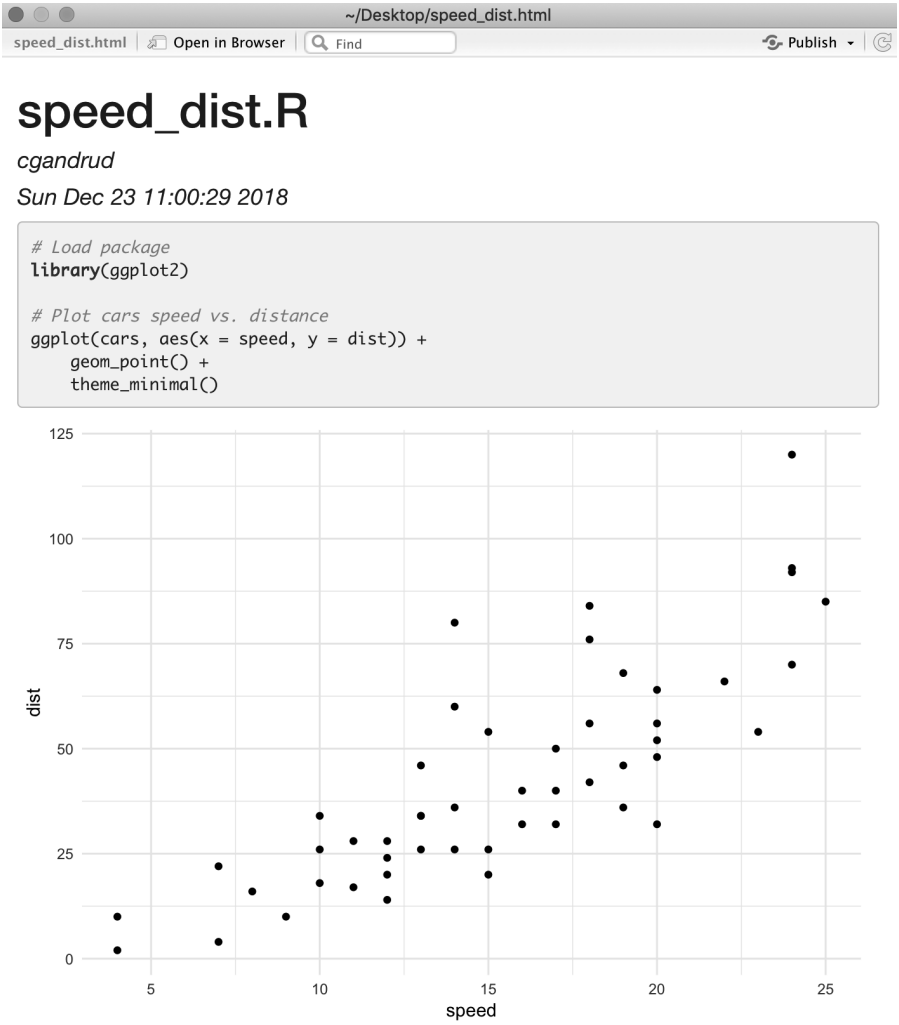


FIGURE 3.6: RStudio Notebook Example

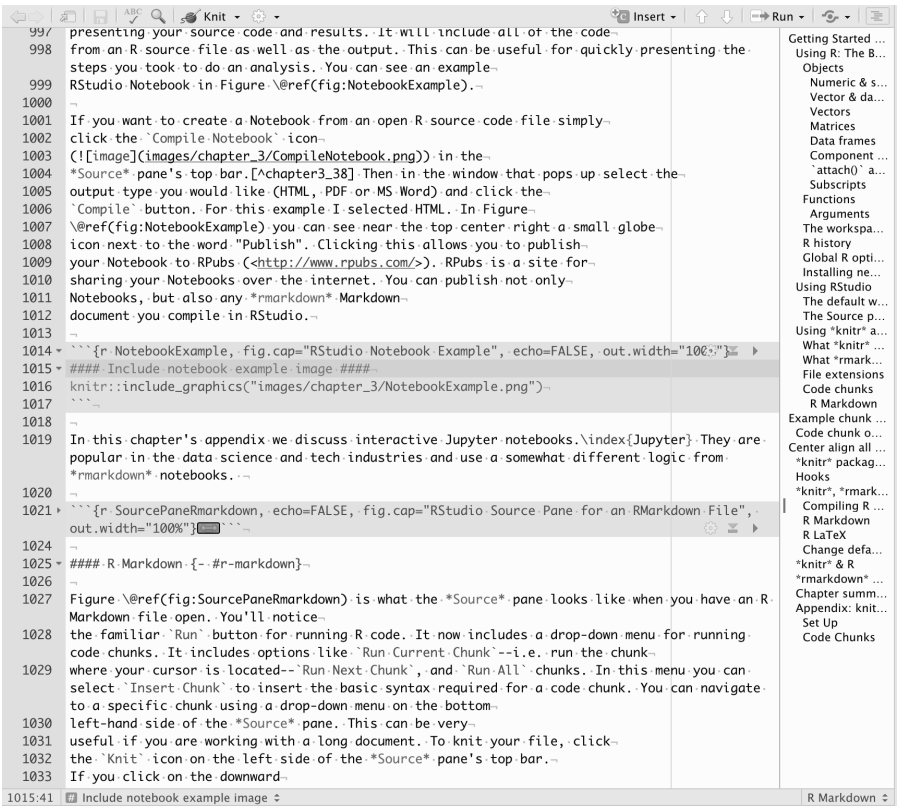


FIGURE 3.7: RStudio Source Pane for an RMarkdown File

`Chunk`, and `Run All` chunks. In this menu, you can select `Insert Chunk` to insert the basic syntax required for a code chunk. You can navigate to a specific chunk using a drop-down menu on the bottom left-hand side of the *Source* pane. This can be very useful if you are working with a long document. To knit your file, click the `Knit` icon on the left side of the *Source* pane's top bar. If you click on the downward arrow on the right of this icon, you will be given the opportunity to knit the document to HTML, PDF, or, MS Word using *rmarkdown*. Helpfully, the R Markdown *Source* pane's top bar also includes the `ABC` spell check icon.

RStudio can properly highlight both the markup language syntax and the R code in the *Source* pane. This makes your source code much easier to read and navigate. RStudio can also fold code chunks. This makes navigating through long documents, with long code chunks, much easier. At line 1014 in Figure 3.7, you can see a small downward facing arrow. If you were to click this arrow, the code chunk would collapse to look like line 1021 in Figure 3.7. To unfold the chunk, just click on the arrow again.

You may also notice that there is a code folding arrow on line 1015 in Figure 3.7. This allows us to fold parts of the code chunk. To enable this option, create a comment line with at least one hash before the comment text and at least four after it like this:

```
#### An RStudio Foldable Comment ####
```

You will be able to fold all of the text after this comment up until the next similarly formatted comment (or the end of the chunk).

R (Sweave) LaTeX

Many of the *Source* pane options for R (`.Rnw`) LaTeX files are the same as R Markdown files, the key differences being that there is a `Compile PDF` icon instead of `Knit`. Clicking this icon knits the file and creates a PDF file in your R LaTeX file's directory. There is also a `Format` icon instead of the question mark icon. This actually inserts LaTeX formatting functions into your document for things such as section headings and bullet lists. These functions can be very tedious to type out by hand otherwise.

By default, RStudio may be set up to use *Sweave* for compiling LaTeX documents. To use *knitr* instead of *Sweave* to knit `.Rnw` files you should click on `Tools` in the RStudio menu bar, then click on `Global Options...`. Once the **Options** window opens, click on the `Sweave` button. Select `knitr` from the drop-down menu for "Weave Rnw files using:". Finally, click `Apply`.[39]

In the `Sweave` options menu, you can also set which LaTeX typesetting engine to use. By default, it is set to the more established engine pdfLaTeX. Another option is XeLaTeX. XeLaTeX has the ability to use many more characters than pdfLaTeX as it works with UTF-8 encoded input. It can also use any font on your computer. XeLaTeX is especially useful compared to pdfLaTeX if you are using characters that are not found in standard English.

3.3.9 knitr and R

As *knitr* is a regular R package, you can of course, knit documents in R (or using the console in RStudio). All of the *knitr* syntax in your markup document is the same as before, but instead of clicking a `Compile PDF` or `knit HTML` button, use the `knit()` function. To knit a hypothetical Markdown file *example.Rmd* you first use the `setwd()` function to set the working directory

[39] In the Mac version of RStudio, you can also access the `Options` window via `RStudio Preferences` in the menu bar.

(for more details see Chapter 4) to the folder where the *example.Rmd* file is located. In this example, it is located in the Documents folder.[40]

```
setwd("/Documents/")
```

Then you `knit()` the file:

```
knit(input = "example.Rmd", output = "example.md")
```

You use the same steps for all other knittable document types. Note that if you do not specify the output file, *knitr* will determine what the file name and extension should be. In this example it would come up with the same name and location as we gave it.

In this example, using the `knit()` function only creates a Markdown file and not an HTML file, as clicking `Knit` in RStudio did. Likewise, if you use on a *.Rnw* file you will only end up with a basic LaTeX *.tex* file and not a compiled PDF. To convert the Markdown file into HTML, you need to further run the *.md* file through the `markdownToHTML()` function from the *markdown* package, i.e.:

```
markdownToHTML(file = "example.md", output = "example.html")
```

This is a bit tedious. Luckily, there is a function in the *knitr* package that combines `markdownToHTML()` and `knit()`. It is called `knit2html()`. You use it like this:

```
knit2html(file = "example.Rmd", output = "example.html")
```

If we want to compile a *.tex* file in R, we run it through the `texi2pdf()` function from the *tools* package. This package will run both LaTeX and BibTeX to create a PDF with a bibliography. See Chapter 11 for more details on using BibTeX for bibliographies. Here is a `texi2pdf()` example:

```
# Load tools package
library(tools)

# Compile pdf
texi2pdf(file = "example.tex")
```

[40] Using the directory name is for Mac computers. Please use alternative syntax discussed in Chapter 4 on other types of systems.

Just like with `knit2html()`, you can simplify this process by using the `knit2pdf()` function to compile a PDF file from a `.Rnw` document.

3.3.10 R Markdown and R

Just as *knitr* is an R package that you can run from the console, you can also run *rmarkdown* from the console. Instead of the `knit()`, function use `render()`. Imagine that *example.Rmd* now has an *rmarkdown* header:

```
---
title: "A Basic PDF Presentation Document"
author: "Christopher Gandrud"
date: "2018-10-28"
output:
    pdf_document:
        toc: true
    html_document:
    toc: false
---
```

This header specifies how the file can be compiled to either PDF or HTML. When compiled to PDF, it will include a table of contents. When compiled to HTML, it won't. Now we use `render()`:

```
render("example.Rmd")
```

This call will compile the document to a PDF in the working directory, because PDF is listed as the first output format in the header. The document will be called *example.pdf*. Alternatively, to compile the R Markdown file to HTML use:

```
render("example.Rmd", "html_document")
```

We could compile to both formats using:

```
render("example.Rmd", "all")
```

or

```
render("example.Rmd", c("pdf_document", "html_document"))
```

In all of these cases, `render()` will create, but not keep the intermediate *.md*

or *.tex* document. You can have these documents saved by adding `keep_md` or `keep_tex` to the header. For example:

```
output:
    pdf_document:
        toc: true
        keep_tex: true
    html_document:
      keep_md: true
    toc: false
---
```

Finally, if you want to output to one format with the default rendering style, for example, the HTML document, use `html_document: default`.

Chapter summary

We've covered a lot of ground in this chapter, including R basics, how to use RStudio, and knitr/R Markdown syntax for multiple markup languages. These tools, especially R and knitr/R Markdown, are fundamental to the reproducible research process we will learn in this book. They enable us to create dynamic text-based files that record our research steps in detail. In the next chapter, we will look at how to organize files created with these types of tools into reproducible research projects.

Appendix: Jupyter Interactive Notebooks

Jupyter notebooks are a commonly used alternative to R Markdown notebooks and knitr generally for displaying and discussing computational analyses. They are especially prevalent in the data science industry. For example, I never used Jupyter notebooks during my academic life in the quantitative social sciences, but after moving to the tech industry I regularly write and read them. A reason for this is that they are useful for fast prototyping data analyses, as they are interactive. You run the code directly in the notebook and see the results printed in the notebook immediately.

Jupyter is often associated with Python, but the name 'Jupyter' actually refers to three languages used in data science **Ju**lia, **Py**thon, and **R** and can be used with other languages as well.

This book is clearly focused on R Markdown, but if you would like to explore launching Jupyter from R, see the *IRkernel* package (Kluyver et al., 2019). Though personally I have been launching Jupyter from Python or Julia, as the installation is more straightforward. In fact, the Python installation is a prerequisite for *IRkernel*. For more details, see:

- Python installation instructions: `http://jupyter.org/install.html`,
- Julia installation instructions: `https://github.com/JuliaLang/IJulia.jl`.

Controversy

In mid-2019 there was a major controversy about Jupyter notebooks (well at least a topic heavily discussed on data science Twitter). Joel Grus started the controversy by giving a talk at the main Jupyter conference, JupyterCon, called, 'I Don't Like Notebooks'.[41]

His critique was multi-pronged, but one critique that resonated with my strong

[41] The presentation is available here: `https://docs.google.com/presentation/d/1n2RlMdmv1p25Xy5thJUhkKGvjtV-dkAIsUXP-AL4ffI/preview?slide=id.g362da58057_0_1`. For a comprehensive discussion of the 'first notebook war' by Yihui Xie, see: `https://yihui.name/en/2018/09/notebook-war/`.

interest in reproducibility (and personal experience using these notebooks) is that you can execute code in Jupyter notebooks in an arbitrary order. Using R Markdown terminology: you could execute the third code chunk before the second and then make changes to and rerun the second chunk without rerunning the third. This is troubling for reproducibility, as it is difficult for a third person (or yourself a few minutes later) to know what order the code was executed in to get the displayed results. Jupyter notebooks do record the order in which code was executed within the same session, but this adds an additional layer of complexity to figuring out results. The order also becomes inconsistent when a notebook is relaunched.

R Markdown vs. Jupyter

A big reason that I personally prefer R Markdown over Jupyter is that it provides the 'best of both worlds'. RStudio allows you to interact with R Markdown documents in a very similar way to Jupyter notebooks (see Figure 3.8). To enable fast prototyping, you can interactively run code chunks in any order and immediately see the results in line with the markup. It also channels you towards running the code in order when you knit the document before you share it with others.[42]

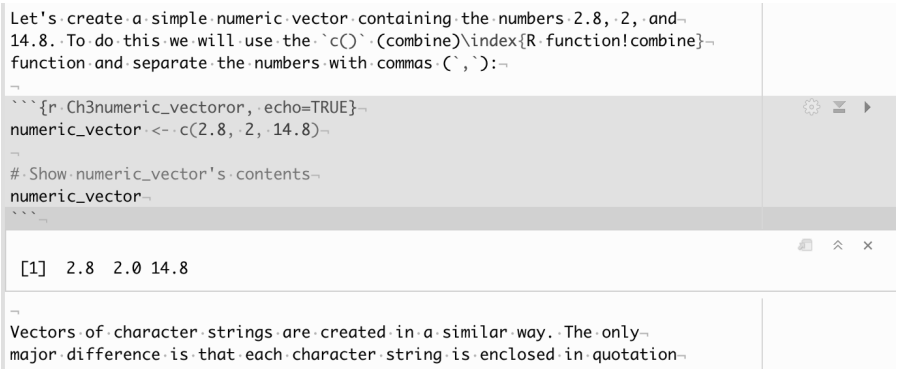


FIGURE 3.8: R Markdown Interactive Behavior Example in RStudio

[42] See Nathan Stephens' 2017 blog post further making the case for R Notebooks: https://rviews.rstudio.com/2017/03/15/why-i-love-r-notebooks/.

Appendix: knitr and Lyx

You may be more comfortable using a what-you-see-is-what-you-get (WYSIWYG) editor, similar to Microsoft Word. Lyx is a WYSIWYG LaTeX editor that can be used with *knitr*. I don't cover Lyx in detail in this book, but here is a little information to get you started.

Setup

To set up Lyx so that it can compile `.Rnw` files, click `Document` in the menu bar, then `Settings`. In the left-hand panel, the second option is `Modules`. Click on `Modules` and select `Rnw (knitr)`. Click `Add`, then `Ok`. Now, compile your LaTeX document in the normal Lyx way.

Code Chunks

Enter code chunks into TeX Code blocks within your Lyx documents. To create a new TeX Code block, select `Insert` from the menu bar, then `TeX Code`.

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

4

Getting Started with File Management

Careful file management is crucial for reproducible research. Remember two of the guidelines from Chapter 2:

- Explicitly tie your files together.
- Have a plan to organize, store, and make your files available.

Apart from the times when you have an email exchange (or even meet in person) with someone interested in reproducing your research, the main information independent researchers have about the procedures is what they access in files you make available: data files, analysis files, and presentation files. If these files are well organized and the way they are tied together is clear, replication will be much easier. File management is also important for you as a researcher, because if your files are well organized, you will be able to more easily make changes, benefit from work you have already done, and collaborate with others.

Using tools such as R, knitr/R Markdown, and markup languages like LaTeX requires fairly detailed knowledge of where files are stored in your computer. Handling files to enable reproducibility may require you to use command-line tools to access and organize your files. R and Unix-like shell programs allow you to control files—creating, deleting, relocating—in powerful and really reproducible ways. By typing these commands you are documenting every step you take. This is a major advantage over graphical user interface-type systems where you organize files by clicking and dragging them with the cursor. However, typed commands require you to know your files' specific addresses, their file paths.

In this chapter we discuss how a reproducible research project may be organized and cover the basics of file path naming conventions in Unix-like operating systems, such as macOS and Linux, and Windows. We then learn how to organize them with RStudio Projects. We'll cover some basic R and Unix-like shell commands for manipulating files as well as how to navigate through files in RStudio in the *Files* pane. The skills you will learn in this chapter will be heavily used in the next chapter (Chapter 5) and throughout the book.

In this chapter we work with locally stored files, i.e. files stored on your

computer. In the next chapter, we will discuss various ways to store and access files remotely stored in the cloud.

4.1 File Paths and Naming Conventions

All of the operating systems covered in this book organize files in hierarchical directories, also known as file trees. To a large extent, directories can be thought of as the folders you usually see on your Windows or Mac desktop.[1] They are called hierarchical because directories are located inside of other directories, as in Figure 4.1.[2]

4.1.1 Root directories

A root directory is the first level in a disk, such as a hard drive. It is the root out of which the file tree 'grows'. All other directories are sub-directories of the root directory.

On Windows computers you can have multiple root directories, one for each storage device or partition of a storage device. The root directory is given a drive letter assignment. If you use Windows regularly, you will most likely be familiar with `C:\` used to denote the C partition of the hard drive. This is a root directory. On Unix-like systems, including Macs and Linux computers, the root directory is denoted by a forward slash (`/`) with nothing before it.

4.1.2 Sub-directories and parent directories

You will probably not store all of your files in the root directory. This would get very messy. Instead, you will store your files in sub-directories of the root directory. Inside of these sub-directories may be further sub-directories and so on. A directory inside of another directory is referred to as a child directory of a parent directory.

On Windows computers, separate sub-directories are indicated with a back slash (\). For example, if we have a folder called *data* inside of a folder called *example-project* which is located in the C root directory,

[1] To simplify things, I use the terms 'directory' and 'folder' interchangeably in this book.

[2] The command line utility *tree* is very useful for visualizing your file trees. For more information, see `https://en.wikipedia.org/wiki/Tree_(command)`.

it has the address `C:\example-project\data`.[3] When you type Windows file paths into R, you need to use two backslashes rather than one: e.g. `C:\\example-project\\data`. This is because the `\` is an escape character in R.[4] Escape characters tell R to interpret the next character or sequence of characters differently. For example, in Section 5.1 you'll see how `\t` can be interpreted by R as a tab rather than the letter "t". Add another escape character to neutralize the escape character so that R interprets it as a backslash. In other words, use an escape character to escape the escape character. Another option for writing Windows file names in R is to use one forward slash (`/`).

On Unix-like systems, including Mac computers, directories are indicated with a forward slash (`/`). The file path of the *data* file on a Unix-like system would be: `/example-project/data`. Remember that a forward slash with nothing before it indicates the root directory. So `/example-project/data` has a different meaning than `example-project/data`. In the former, *example-project* is a sub-directory of the root. In the latter, *example-project* is a sub-directory of the current working directory (see below for details about working directories). This is also true in Windows.

In this chapter, I switch between the two file system naming conventions to expose you to both. In subsequent chapters, I use Unix-like file paths. When you use relative paths (see below), these will work across operating systems in R. We'll get to relative paths in a moment.

4.1.3 Working directories

When you use R, markup languages, and many of the other tools covered in this book, it is important to keep in mind what your current working directory is. The working directory is the directory where the program automatically looks for files and other directories, unless you tell it to look elsewhere. It is also where it will save files. Later in this chapter, we will cover functions for finding and changing the working directory.

4.1.4 Absolute vs. relative paths

For reproducible research, collaborative research, and even if you ever change the computer you work on, it is a good idea to use relative rather than absolute file paths. Absolute file paths give the entire path of a given file or directory on a specific system. For example, `/example-project/data` is an absolute path. It specifies the path of the *data* child directory all the way back to the

[3] For more information on Windows file path names, see this helpful website: `http://msdn.microsoft.com/en-us/library/windows/desktop/aa365247(v=vs.85).aspx`

[4] As we will see in Part IV, it is also a LaTeX and Markdown escape character.

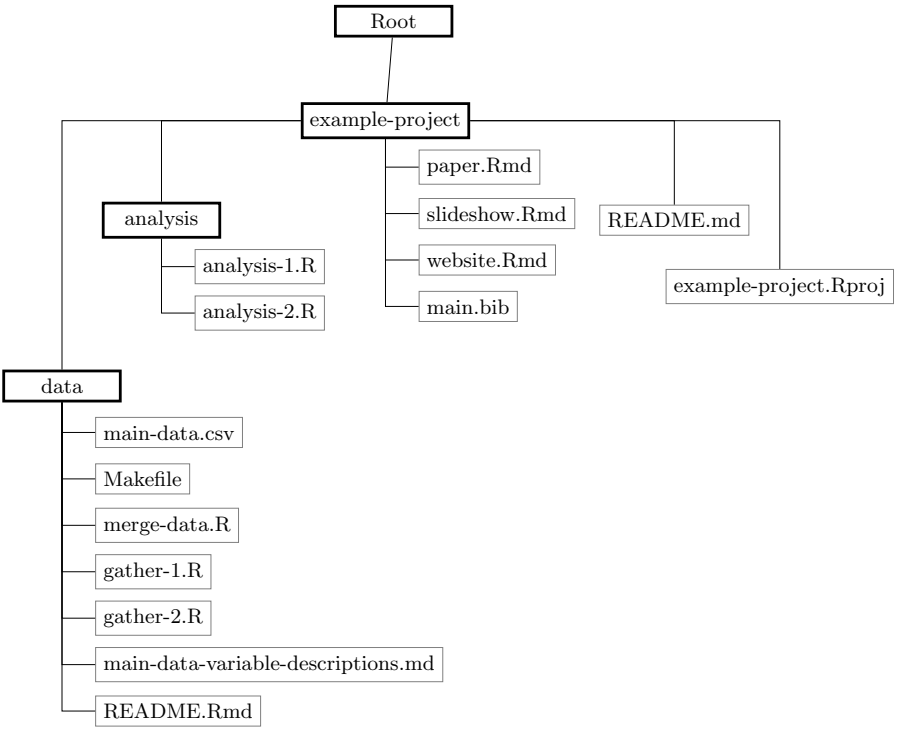


FIGURE 4.1: Example Research Project File Tree

root directory. However, if our current working directory is *example-project* and we want to link to the *data* child directory or a file in it, we don't need the absolute path. We could use `data/`, i.e. the path relative to the working directory.

It is good practice to use relative paths and organize your files such that using relative paths is easy. This makes your code less dependent on the particular file structure of a particular computer. For example, imagine you use `C:\\example-project\\data` in your source code to link to the *data* directory. If someone—a collaborator, a researcher reproducing your work, or even you—then tries to run the code on a different computer, the code will break if they are, for instance, using a Unix-like system or have placed *example-project* in a different partition of their hard drive. This can be fixed relatively by changing the file path in the source. However, this is tedious (often not well documented) and unnecessary if you use relative file paths.

Below we'll see how to use RStudio Projects and also the *here* package (Müller, 2017) to automatically set working directories so that your relative file paths will transport even more easily across computers.

The *ProjectTemplate* package (White, 2019) provides functions to help set up a well structured research project file tree. We don't use it in the following examples, but you may find it useful in your own work.

4.1.5 Spaces in directory and file names

It is good practice to avoid putting spaces in your file and directory names. For example, I called the example project parent directory in Figure 4.1 "example-project" rather than "Example Project". Spaces in file and directory names can sometimes create problems for computer programs trying to read the file path. The program may believe that the space indicates that the path name has ended. To make multi-word names easily readable without using spaces, adopt a consistent naming convention.

One approach is to use a convention that contrasts with the R object naming convention you are using. A contrasting convention helps make it clear if something is an R object or a file name. For example, if we adopt the underscore method for R object names used in Chapter 3 (e.g. `health_data`) we could use hyphens (`-`) to separate words in file names. For example: `example-source.R`. This is sometimes called kebab-case.

4.2 Organizing Your Research Project

Figure 4.1 gives an example of how the files in a simple reproducible research project could be organized. The project's parent directory is called *example-project*. Inside this directory are the primary knittable documents (*paper.Rmd slideshow.Rmd*, and *website.Rmd*). In addition, there is an *analysis* sub-directory with the R files to run the statistical analyses followed by a further *data* child directory.

The nested file structure allows you to use relative file paths. The knittable documents can call *analysis-1.R* with the relative path *analysis/analysis-1.R*.

In addition to the main files and sub-directories in *example-project*, you will notice files called *README.md* and *example-project.Rproj*. We'll discuss the *example-project.Rproj* file in the next section. The *README.md* file is a human readable overview of all the files in the project. It should briefly describe the project including things like its title, author(s), topic, any copyright information, and so on. It should also indicate how the folders in the project are organized and give instructions for how to reproduce the project. The README file should be in the main project folder—in our example this is called *example-*

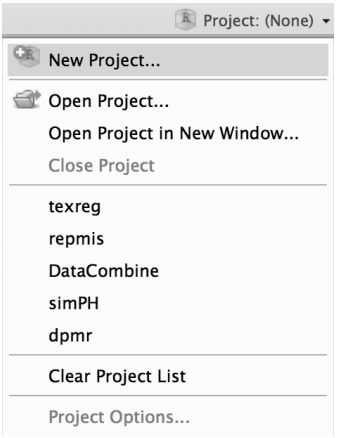


FIGURE 4.2: An Example RStudio Project Menu

project—so that it is easy to find. If you are storing your project as a GitHub repository (see Chapter 5) and the file is called *README*, its contents will automatically be displayed on the repository's main page. If the *README* file is written using Markdown (e.g. *README.md*), it will also be properly formatted. Figure 5.2 shows an example of this.

It is good practice to dynamically include the system information for the R session you used to create the project. To do this, you can write your README file with R Markdown. Simply include the `sessionInfo()` function in a *knitr* code chunk in the R Markdown document. If you knit this file immediately after knitting your presentation document, it will record the information for that session.

You can also dynamically include session info in a LaTeX document. To do this, use the function in a code chunk. The code chunk should have the option `results='asis'`. The code is:

```
toLatex(sessionInfo())
```

4.3 Organizing Research with RStudio Projects

If you are using RStudio, you may want to organize your files as Projects. You can turn a normal directory into an RStudio Project by clicking on `File` in the RStudio menu bar and selecting `New Project...`. A new window will pop

up. Select the option `Existing Directory`. Find the directory you want to turn into an RStudio Project by clicking on the `Browse` button. Finally, select `Create Project`. You will also notice in the Create Project pop-up window that you can build new project directories and create a project from a directory already under version control (we'll do this at the end of Chapter 5). When you create a new project, you will see that RStudio has put a file with the extension `.Rproj` into the directory, like *example-project.Rproj* in Figure 4.1.

Making your research project directories RStudio Projects is useful for a number of reasons:

- The project is listed in RStudio's Project menu where it can be opened easily (see Figure 4.2).
- When you open the project in RStudio, it automatically sets the working directory to the project's directory and can load the source code files you were last working on.
- You can set project specific options like whether PDF presentation documents should be compiled with *Sweave* or *knitr*.
- When you close the project, your R workspace and history are saved in the project directory if you want. However, avoid saving your workspace as this could make reproducibility harder.
- It helps you version control your files.
- You can build your Project—run the files in a specific way—with makefiles.
- It gives you an easy-to-use interface for managing the R packages that your project depends on.

4.4 R File Manipulation Functions

R has a range of functions for handling and navigating through files. Including these functions in your source code files allows you to more easily replicate your actions.

`getwd()`

To find your current working directory use the `getwd()` function:

```
getwd()
```

```
##[1]"/Users/cgandrud/git_repos/Rep-Res-Book/rep-res-3rd-edition"
```

list.files()

Use the `list.files()` function to see all of the files and sub-directories in the current working directory. You can list the files in other directories too by adding the directory path as an argument to the function.

Because my current working directory has a lot of files in it, I will shorten the output for illustration by piping it through `head()`.

```
library(magrittr)
list.files() %>% head()
```

```
## [1] "_book"
## [2] "_bookdown_files"
## [3] "_bookdown.yml"
## [4] "_output.yml"
## [5] "01-author.Rmd"
## [6] "01-stylistic-conventions.Rmd"
```

setwd()

The `setwd()` function is the base R way to set the current working directory. For example, if we are on a Mac or other Unix-like computer, we can set the working directory to the *analysis* directory in our Example Project (see Figure 4.1) like this:

```
setwd("/example-project/analysis/")
```

Now R will automatically look in the *analysis* folder for files and will save new files into this folder, unless we explicitly tell it to do otherwise.

When working with a knittable document, setting the working directory once in a code chunk changes the working directory for all subsequent code chunks.

However . . .

here::set_here()

It is *not* good practice for reproducibility (and just general convenience when using a source code file across multiple computers) to use `setwd()` in your R source code. You, and anyone reproducing your work, will need to tediously set specific file paths for each computer.

Instead, use RStudio Projects, which automatically set the working directory to the one with the *.Rproj* file. If you are not using RStudio Projects, include `set_here()` from the *here* package at the top of your source code. This will create a file called *.here* in the current working directory. It functions similarly to *.Rproj* to automatically flag for *here* what should be the current working directory. Remember when you share your source code to also share the *.Rproj*/*.here* file.

`root.dir` in knittable documents

By default, the root (or working) directory for all of the code chunks in a knittable document is the directory where this document is located. You can reset the directory by feeding a new file path to the `root.dir` option. We can set this globally[5] for all of the chunks in the document by including the following code in the document's first chunk.

```
opts_knit$set(root.dir = "/example-project/analysis")
```

We set the */example-project/analysis* sub-directory as the root directory for all of the chunks in our presentation document.

Note: In general it is preferable to use the knittable file's default directory and file paths relative to it rather than manually specifying `root.dir()`. Setting an alternate root directory will make reproducibility more difficult.

`dir.create()`

Sometimes you may want to create a new directory. You can use the `dir.create()` function to do this.[6] For example, to create an *example-project* file in the root *C* directory on a Windows computer, type:

```
dir.create("C:\\example-project")
```

`file.create()`

Similarly, you can create a new blank file with the `file.create()` function. To add a blank R source code file called *source-code.R* to the *example-project* directory on the *C* drive, use:

[5] See the discussion of global chunk options in Chapter 3, Section 3.3.5.
[6] Note: you will need the correct system permissions to be able to do this.

```
file.create("C:\\example-project\\source-code.R")
```

cat()

If you want to create a new file and put text into it, use the `cat()` (concatenate and print) function. For example, to create a new file in the current working directory called *example-cat.md* that includes the text "Reproducible Research with R and RStudio" type:

```
cat("Reproducible Research with R and RStudio",
    file = "example-cat.md")
```

In this example we created a markdown formatted file by using the `.md` file extension. We could, of course, change the file extension to `.R` to set it as an R source code file, `.Rnw` to create a *knitr* LaTeX file, and so on.

You can use `cat()` to print the contents of one or more objects to a file. **Warning:** the `cat()` function will overwrite existing files with the new contents. To add the text to existing files, use the `append = TRUE` argument.

```
cat("More Text", file = "example-cat.md", append = TRUE)
```

unlink()

You can use the `unlink` function to delete files and directories.

```
unlink("C:\\example-project\\source-code.R")
```

Warning: the `unlink()` function permanently deletes files, so be very careful using it.

file.rename()

You can use `file.rename()` to, obviously, rename a file. It can also be used to move a file from one directory to another. For example, imagine that we want to move the *example-cat.md* file from the directory *example-project* to one called *markdown-files* that we already created.[7]

[7] The `file.rename()` function won't create new directories. To move a file to a new directory, you will need to create the directory first with `dir.create()`.

```
file.rename(from = "C:\\example-project\\example-cat.md",
            to = "C:\\markdown-files\\example-cat.md")
```

`file.copy()`

`file.rename()` fully moves a file from one directory to another. To copy the file to another directory, use the `file.copy()` function. It has the same syntax as `file.rename()`:

```
file.copy(from = "C:\\example-project\\example-cat.md",
          to = "C:\\markdown-files\\example-cat.md")
```

4.5 Unix-like Shell Commands for File Management

Though this book is mostly focused on using R for reproducible research, it can be useful to use a Unix-like shell program to manipulate files in large projects. Unix-like shell programs, including Bash on Linux (and Mac before macOS Catalina), Zsh on Mac (from macOS Catalina onwards), and Windows PowerShell, give you type-able commands to interact with your computer's operating system.[8] We will especially return to shell commands in the next chapter when we discuss Git version control and makefiles for collecting data in Chapter 6, as well as the command-line program[9] Pandoc in Chapter 12. We don't have enough space to fully introduce shell programs or even all of the commands for manipulating files. We are just going to cover some of the basic and most useful commands for file management. For good introductions for Unix and macOS computers, see William E. Shotts Jr.'s (2012) book on the Linux command-line. For Windows users, Microsoft maintains a tutorial on Windows PowerShell at `http://technet.microsoft.com/en-us/library/hh848793`. The commands discussed in this chapter should work in both Unix-like shells and Windows PowerShell.

It's important at this point to highlight a key difference between R and Unix-like shell syntax. Shell command arguments don't have parentheses. For

[8] You can access Bash via the Terminal program on macOS and Linux computers. It is the default shell on Mac (before macOS Catalina) and Linux, so it loads automatically when you open the Terminal. Windows PowerShell comes installed with Windows.

[9] A command-line program is just a program you run from a shell.

example, if I want to change my working directory to my Mac Desktop in a shell using the `cd` command, I type:[10]

```
cd /Users/cgandrud/Desktop
```

In this example `cgandrud` is my user name.

`cd`

As we just saw, use the `cd` (change directory) command to change the working directory in the shell. Here is an example of changing the directory in Windows PowerShell to `C:/`:

```
cd C:/
```

If you are in a child directory and want to change the working directory to the previous working directory you were in, type:

```
cd -
```

If, for example, our current working directory is */User/Me/Desktop* and we typed `cd` followed by a minus sign (`cd -`), then the working directory would change to */User/Me*. Note this will not work in PowerShell.

`pwd`

To find your current working directory, use the `pwd` command (present working directory). This is essentially the same as R's `getwd()` function.

```
pwd
```

`ls`

The `ls` (list) command works very similarly to R's `list.files()` function. It shows you what is in the current working directory.

Again, I have a lot of files in my working directory, so I will shorten the output for this example by piping it through the command line's `head` command. The command line pipe is not `%>%`, as in R, but instead `|`.

[10] Many shell code examples in other sources include the shell prompt, like the `$` in Bash, or `>` in PowerShell. These are like R's `>` prompt. I don't include the prompt in code examples in this book because you don't type them.

```
ls | head
```

```
## 01-author.Rmd
## 01-stylistic-conventions.Rmd
## 02-additional-resources.Rmd
## 03-introduction.Rmd
## 04-getting-started.Rmd
## 05-start-R.Rmd
## 06-file-management.Rmd
## 07-storage.Rmd
## 08-gather.Rmd
## 09-clean.Rmd
```

As we saw earlier, R also has an `ls` command. R's `ls()` function lists items in the R workspace. The shell's `ls` command lists files and directories in the working directory.

`mkdir`

Use `mkdir` to create a new directory. For example, if I wanted to create a sub-directory of my Linux root directory called *new-directory* I would type:

```
mkdir /new-directory
```

`echo`

There are a number of ways to create new files in Unix-like shells. One of the simplest is the `echo` command. This command prints its argument to the Terminal. For example:

```
echo Reproducible Research with R and RStudio
```

```
## Reproducible Research with R and RStudio
```

If you add the greater-than symbol (`>`) after the text you want to print and then a file name, `echo` will create the file (if it doesn't already exist) in the current working directory and then print the text into the file.

```
echo Reproducible Research with R and RStudio > example-echo.md
```

Using only one greater-than sign will completely erase the *example-echo.md* file's contents and replace them with `Reproducible Research with R and RStudio`. To append the text at the end of an existing file, use two greater-than signs (`>>`).

```
echo More text. >> example-echo.md
```

There is also a `cat` shell command. It works slightly differently than the R version of the command and I don't cover it here.

`rm`

The command `rm` removes (deletes) files or directories.

```
rm example-echo.md
```

Add the `d` (directory) option to delete a directory. Note that options are like arguments in an R function. For example:

```
rm -d example-dir
```

Again, be careful when using this command, because it permanently deletes the files or directories.

As we saw in Chapter 3, R also has an `rm()` function. It is different because it removes objects from your R workspace rather than files from your working directory.

`mv`

To move a file from one directory to another from a shell, use the `mv` (move) command. For example, to move the file *example-echo.md* from *example-project* to *markdown-files*, use the following code and imagine both directories are in the root directory:[11]

```
mv /example-project/example-echo.md/markdown-files
```

Note that the *markdown-files* directory must already exist. If it does not exist, the file will just be renamed. This is similar to the R function `file.rename()`.

[11] If they were not in the root directory, we would not place a forward slash at the beginning.

`cp`

The `mv` command completely moves a file from one directory to another. To copy a version of the file to a new directory use the `cp` command. The syntax is similar to `mv`:

```
cp /example-project/ExampleEcho.md /markdown-files
```

`system()` (R function)

You can run shell commands from within R using R's `system()` function. For example, to run the `echo` command from within R type:

```
system("echo Text to Add > ExampleEcho.md")
```

4.6 File Navigation in RStudio

The RStudio *Files* pane allows us to navigate our file tree and do some basic file manipulations. Figure 4.3 shows us what this pane looks like. The pane allows us to navigate to specific files and folders and delete and rename files. To select a folder as the working directory, tick the dialog box next to the file. Then click the `More` button and select `Set As Working Directory`. Under the `More` button, you will also find options to `Move` and `Copy` files (see Figure 4.4).

The *Files* pane is a Graphical User Interface (GUI), so our actions in the *Files* pane are not recorded, as such are not as easily reproducible as the commands we learned earlier in this chapter.

Chapter summary

In this chapter we've learned how to organize our research files to enable dynamic replication. This included not only how they can be ordered in a computer's file system, but also the file path naming conventions—the addresses—that computers use to locate files. Once we know how these addresses work, we can use R and shell commands to refer to and manipulate our files. This skill is particularly useful because it allows us to place code in text-based files to manipulate our project files in highly reproducible ways. In

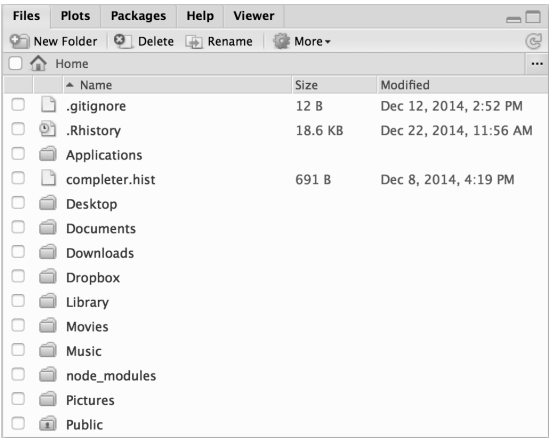


FIGURE 4.3: The RStudio Files Pane

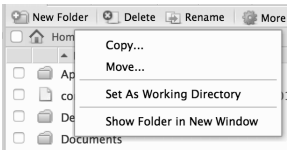


FIGURE 4.4: More Functionality in the RStudio Files Pane

the next few chapters, we will put these skills into practice. We will learn how to store our files and create data files in reproducible ways.

Part II

Data Gathering and Storage

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

5

Storing, Collaborating, Accessing Files, and Versioning

In addition to being well organized, your research files need to be accessible for other researchers to be able to reproduce your findings. A useful way to make your files accessible is to store them on a cloud storage service[1] (Howe, 2012). This chapter describes in detail two different cloud storage services, Dropbox and GitHub, that you can use to make your research files easily accessible to others. Not only do these services enable others to reproduce your research, but they also have a number of benefits for your research workflow. These are certainly not the only services for remote research file storage, but discussing them does cover many common concerns of other services.

Researchers often face a number of data management issues that, beyond making their research difficult to reproduce, can make doing the initial research difficult.

First, there is the problem of **storing** data so that it is protected against computer failure—virus infections, spilling coffee on your laptop, and so on. Storing data locally on your computer or on a flash drive is generally more prone to loss than on remote servers in the cloud.

Second, we may work on a project with different computers and mobile devices. For example, we may use a computer at work to run computationally intensive analysis, while editing our presentation document on a tablet, while riding the train to the office. So, we need to be able to **access** our files from multiple devices in different locations. We often need a way for our **collaborators** to access and edit research files as well.

Finally, we almost never create a data set or write a paper perfectly all at once. We may make changes and then realize that we liked an earlier version, or parts of an earlier version better. This is a particularly important issue in data management where we may transform our data in unintended ways and want to go back to earlier versions. Also, when working on a collaborative project, one of the authors may accidentally delete something in a file that another author needed. To deal with these issues, we need to store our data in a system

[1] These services store your data on remote servers.

that has **version control**. Version control systems keep track of changes we make to our files and allows us to access previous versions if we want to.

You can solve all of these problems in a couple of different ways using free or low cost cloud-based storage formats. In this chapter, we will learn how to use Dropbox and Git/GitHub for research files:

- storage,
- accessing,
- collaboration,
- version control.

5.1 Saving Data in Reproducible Formats

Before getting into the details of cloud-based data storage for all of our research files, let's consider what type of formats you should actually save your data in. A key issue for reproducibility is that others are able to not only get hold of the exact data you used in your analysis, but also be able to understand and use the data now and in the future. Some file formats make this easier than others.

In general, for small to moderately sized data sets[2] plain-text formats like comma-separated values (`.csv`) or tab-separated values[3] (`.tsv`) are good ways to store your data. These formats store a data set as a text file. A row in the data set is a line in the text file. Data is separated into columns with commas or tabs, respectively. These formats are not dependent on a specific program. Any program that can open text files can open them, including a wide variety of statistical programs other than R as well as spreadsheet programs like Microsoft Excel. Using text file formats helps future-proof your research. Version control systems that track changes to text, like Git, are also very effective version control systems for these types of files.

The `write.table()` function is one way to save data in plain-text formats from R. For example, to save a data frame called *data* as a comma-separated-

[2] I don't cover methods for storing and handling very large data sets, with high hundreds of thousands and more observations. For information on large data and R, not just storage, one place to look is this blog post from RDataMining: `http://rdatamining.wordpress.com/2012/05/06/online-resources-for-handling-big-data-and-parallel-computing-in-r/` (posted 6 May 2012). One popular service for large file storage is Amazon S3 (`http://aws.amazon.com/s3/`).

[3] Sometimes this format is called tab-delimited values.

value (CSV) file called *main-data.csv* in our example *data* directory (see Figure 4.1):

```
write.table(Data, "/example-project/data/main-data.csv",
            sep = ",",
            row.names = FALSE)
```

`row.names = FALSE` prevents R from including the row names in the output file.[4] The `sep = ","` argument specifies that we want to use commas to separate values into columns. For CSV files, you can use a modified version of this command called `write.csv()`. This function makes it so that you don't have to write `sep = ","`.[5]

If you want to save your data with values separated by tabs, rather than commas, set the argument `sep = "\t"` and set the file extension to `.tsv`.

R is able to save data in a wide variety of other file formats, mostly through the *foreign* or *rio* (hong Chan and Leeper, 2018) packages (see Chapter 6). These formats may be less future-proof than simple text-formatted data files.

One advantage of many other statistical program file formats is that they include not only the underlying data, but also other information like variable descriptions. If you are using plain-text files to store your data, you will need to include a separate file, preferably in the same directory as the data file describing the variables and their sources. In Chapter 9 we will look at how to automate the creation of variable description files.

5.2 Storing Your Files in the Cloud: Dropbox

In this book we'll cover two (largely) free cloud storage services that allow you to store, access, collaborate on, and version control your research files. These services are Dropbox and GitHub.[6] Though they both meet our basic storage needs, they do so in different ways and require different levels of effort to set up and maintain.

These two services are certainly not the only way to make your research files

[4] Frequently the row names are just the row numbers which may have no substantive meaning.

[5] `write.csv()` is a 'wrapper' for `write.table()`.

[6] Dropbox provides a minimum amount of storage for free, above which they charge a fee. GitHub lets you create publicly accessible repositories—kind of like project folders—for free, but they charge for private repositories.

available. Research-oriented services include Zenodo,[7] the Dataverse Project,[8] figshare,[9] and RunMyCode.[10] These services include good built-in citation systems, unlike Dropbox and GitHub. They also aim to provide persistent URLs for your files. This helps avoid the 'link rot' that threatens reproducibility, i.e. a hosting service changes its URL structure breaking existing links. These services may be a very good place to store research files once the research is completed or close to completion. Many journals now require replication files be uploaded to these sites. However, these sites' ability to store, access, collaborate on, and version control files *during* the main part of the research process is mixed. Services like Dropbox and GitHub are very capable of being part of the research workflow from the beginning.

Zenodo and GitHub have excellent integration, allowing you to actively develop a research project on GitHub then persist it on Zenodo. For details, see `https://guides.github.com/activities/citable-code/`.

The easiest types of cloud storage for your research are services like Dropbox[11] and Google Drive.[12] These services not only store your data in the cloud, but also provide ways to share files. They even include basic version control capabilities. I'm going to focus on Dropbox because it currently offers a complete set of features that allow you to store, version, collaborate, and access your data. I will focus on how to use Dropbox on a computer. Some Dropbox functionality may be different on mobile devices.

5.2.1 Storage

When you sign up for Dropbox and install the program,[13] it creates a directory on your computer's hard drive. When you place new files and folders in this directory and make changes to them, Dropbox automatically syncs the directory with a similar folder on a cloud-based server. Typically when you sign up for the service, you'll receive a limited amount of storage space for free, usually a few gigabytes. This is probably enough storage space for a number of text file-based research projects with smaller data sets.

[7] `https://zenodo.org/`
[8] `https://dataverse.org/`
[9] `http://figshare.com/`
[10] `http://www.runmycode.org/`
[11] `http://www.dropbox.com/`
[12] `https://drive.google.com/`
[13] See `https://www.dropbox.com/downloading` for downloading and installation instructions.

5.2.2 Accessing data

All files stored on Dropbox have a URL address through which they can be accessed from a computer connected to the internet. To access a Dropbox file or directory's URL so that it can be downloaded, right-click on the file icon in your Dropbox folder on your computer. Then click `Copy Dropbox Link`. This copies the URL into your clipboard.

You need to make one small change to the link so that it can be programmatically downloaded. By default, the link will point to the Dropbox website page for the file/directory. To be able to programmatically download it, you need to change the last `0` in the URL to a `1`. For example, change:

`https://www.dropbox.com/s/1xapw69efofpg3b/public.fin.msm.model.csv?dl=0`

to

`https://www.dropbox.com/s/1xapw69efofpg3b/public.fin.msm.model.csv?dl=1`

We changed the download (`dl`) option from false (`dl=0`) to true (`dl=1`). Now you can use the link to download data in your R source code, for example.

Once you have the URL, you can load the file directly into R using the `import()` function from the *rio* package. `import()` works for many different data formats and is generally more robust than `read.table()`. Use the `source_url()` function in the *devtools* package (Wickham et al., 2019c) to download and run R source code files (see Chapter 8).

Let's download data directly into R from Dropbox. The data set's URL is: `https://www.dropbox.com/s/1xapw69efofpg3b/public.fin.msm.model.csv?dl=1`.[14]

```
# Download data on financial regulators stored on Dropbox

# Load rio
library(rio)

# Place the URL into the object fin_url
fin_url <- "https://bit.ly/2xlQ2j5"

# Download data
fin_regulator <- import(fin_url, format = "csv")
```

[14] This data is from (Gandrud, 2013b). I've shortened the URL using Bitly (`https://bitly.com/`) so that it will fit on the page.

```
# Show variables in fin_regulator
names(fin_regulator)
```

```
## [1] "idn"        "country"    "year"       "reg_4state"
```

The argument `format = "csv"` tells `import()` what format the file is in. This isn't necessary if the file path has an informative file extension, e.g. it ends with `.csv`.

5.2.3 Collaboration

Though others can easily access your data and files with Dropbox URL links, you cannot save files through the link. You must save files in the Dropbox folder on your computer or upload them through the website. If you would like collaborators to be able to modify the research files, you will need to 'share' the Dropbox folder with them. Once you create a Dropbox folder, you can share it with your collaborators by right-clicking on the folder's name. Then select `Share`. Enter your collaborator's email address when prompted and select `Can Edit` from the permissions dropdown menu. They will be sent an email that will allow them to accept the share request and, if they don't already have an account, they can sign up for Dropbox.

5.2.4 Version control

Dropbox has a simple version control system. Every time you save a document a new version is created on Dropbox. To view a previous version, navigate to the file on the Dropbox website. Then click on the file. In the upper-right corner, there is a menu where you can select `Version history`. This will take you to a page listing previous versions of the file, who created the version, and when it was created. A new version of a file is created every time you save a file and it is synced to the Dropbox cloud service.

Note that with a free Dropbox account, previous versions of a file are only stored for **30 days**. You need a paid account to save previous versions for more than 30 days.[15]

[15] For more details, see `https://www.dropbox.com/en/help/11`.

5.3 Storing Your Files in the Cloud: GitHub

Dropbox minimally meets our four basic criteria for reproducible data storage. It is easy to set up and use. GitHub meets the criteria and more, especially when it comes to version control. It is, however, less straightforward at first. In this section, we will learn enough of the basics to get you started using GitHub to store, access, collaborate on, and version control your research.

GitHub is an interface and cloud hosting service built on top of the Git version control system.[16] Git does the version control. GitHub stores the data remotely, as well as provides a number of other features, some of which we look at below. GitHub was not explicitly designed to host research projects or even data. It was designed to host "socially coded" computer programs—in what Git calls "repositories", repos for short—by making it easy for a number of collaborators to work together to build computer programs. This seems very far from reproducible research.

Remember that as reproducible researchers, we are building projects out of interconnected text files. In important ways, this is exactly the same as building a computer program. Computer programs are also basically large collections of interconnected text files. Like computer programmers, we need ways to store, version control, access, and collaborate on our text files. Because GitHub is very actively used by people with similar needs (who are also really good programmers), the interface offers many highly developed and robust features for reproducible researchers.

GitHub's extensive features and heart in the computer programming community means that it takes a longer time than Dropbox for novice users to set up and become familiar with. So we need good reasons to want to invest the time needed to learn GitHub. Here is a list of GitHub's advantages over Dropbox for reproducible research that will hopefully convince you to get started using it:[17]

Storage and access

- Dropbox creates folders stored in the cloud which you can share with other people. GitHub makes your projects accessible on a fully featured project website (see Figure 5.2). An example feature is that it automatically renders

[16] I used Git version 2.20.1 for this book.

[17] Because many of these features apply to any service that relies on Git, much of this list of advantages also applies to alternative Git cloud storage services such as Bitbucket (`https://bitbucket.org/`).

Markdown files called *README.md*[18] in a GitHub directory on the repository's website. This makes it easy for independent researchers to find the file and read it.

- GitHub can create and host a website for your research project that you could use to present the results, not just the replication files.
- Its close integration with Zenodo allows you to easily make your full replication material persistently accessible and citable.

Collaboration

- Dropbox allows multiple people to share files and change them. GitHub does this and more.
- GitHub keeps meticulous records of who contributed what to a project.
- Each GitHub repository has an "Issues" area where you can note issues and discuss them with your collaborators. Basically, this is an interactive to-do list for your research project. It also stores the issues so you have a full record.
- Each repository can also host a wiki that, for example, could explain in detail how certain aspects of a research project were done.
- Anyone can suggest changes to files in a public repository. These changes can be accepted or declined by the project's authors. The changes are recorded by the Git version control system. This could be especially useful if an independent researcher notices an error.

Version control

- Dropbox's version control system only lets you see file names, the times they were created, who created them, and revert back to specific versions. Git tracks every change you make. The GitHub website and GUI programs for Mac and Windows provide nice interfaces for examining specific changes in text files.
- Dropbox creates a new version every time you save a file. This can make it difficult to actually find the version you want as the versions quickly multiply. Git's version control system only creates a new version when you tell it to.
- All files in Dropbox are version controlled. Git allows you to ignore specific files. This is helpful if you have large binary files (i.e. not text files) that

[18] You can use a variety of other markup languages as well. See `https://GitHub.com/GitHub/markup`.

you do not want to version control because doing so will use up considerable storage space.

- Unless you have a paid account, previous file versions in Dropbox disappear after 30 days. GitHub stores previous versions indefinitely for all account types.
- Dropbox does not merge conflicting versions of a file together. This can be annoying when you are collaborating on a project and more than one author is making changes to documents at the same time. Git identifies conflicts and lets you reconcile them.
- Git is directly integrated into RStudio Projects.[19]

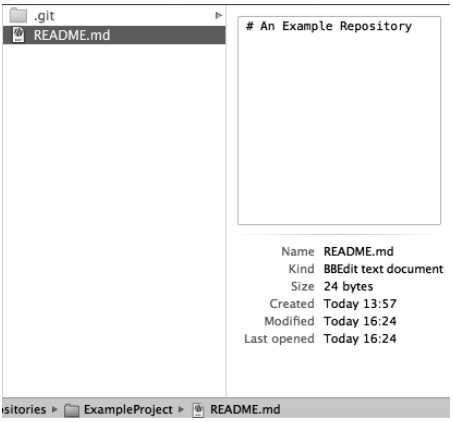


FIGURE 5.1: A Basic Git Repository with Hidden .git Folder Revealed

5.3.1 Setting up GitHub: Basic

There are at least three ways to use Git/GitHub on your computer. You can use the command-line version of Git. It's available for Mac and Linux (in the Terminal) as well as Windows through Git Bash.[20] You can also use the Graphical User Interface GitHub program. Currently, it's only available for Windows and Mac. RStudio also has GUI-style Git functionality for RStudio Projects. In this section, I focus on how to use the command-line version, because it will help you understand what the GUI versions are doing and will allow you to better explore more advanced Git features not covered in this book. In the next section, I will mention how to use Git with RStudio Projects.

[19] RStudio also supports the Subversion version control system, but I don't cover that here.

[20] The interface for Git Bash looks a lot like the Terminal or Windows PowerShell.

The first thing to do to set up Git and GitHub is go to the GitHub website (`https://github.com/`) and sign up for an account. Second, you should go to the following website for instructions on setting up GitHub: `https://help.github.com/articles/set-up-git/`. The instructions on that website are very comprehensive, so I'll direct you there for the full setup information. Note that installing the GUI version of GitHub also installs Git and, on Windows, Git Bash.

5.3.2 Version control with Git

Git is primarily a version control system, so we will start our discussion of how to use it by looking at how to version your repositories.

Setting up Git repositories locally

You can set up a Git repo on your computer with the command-line.[21] I keep my repositories in a folder called *git_repositories*,[22] though you can use Git with almost any directory you like. The *git_repositories* directory has the root folder as its parent. Imagine that we want to set up a repository in this directory for a project called *example_project*. Initially it will have one README file called *README.md*. To do this, we would first type into the Terminal for Mac and Linux computers:

```
# Make new directory 'example-project'
mkdir /git_repositories/example-project

# Change to directory 'example-project'
cd /git_repositories/example-project

# Create new file README.md
echo "# An Example Repository" > README.md
```

So far, we have only made the new directory and set it as our working directory (see Chapter 4). All of the examples in this section assume your current working directory is set to the repo. Then, with the `echo` shell command we created a new file named *README.md* that includes the text `# An Example Repository`. Note that the code is basically the same in Windows PowerShell or Git Bash. Also, you don't have to do these steps in the command-line. You

[21] Much of the discussion of the command-line in this section is inspired by Nick Farina's blog post on Git (see `http://nfarina.com/post/9868516270/git-is-simpler`, posted 7 September 2012).

[22] To follow along with this code, you will first need to create a folder called *git_repositories* in your root directory. Note also that throughout this section I use Unix file path conventions.

could just create the new folders and files the same way that you normally do with your mouse in your GUI operating system.

Now that we have a directory with a file, we can tell Git that we want to treat the directory *example-project* as a repository and that we want to track changes made to the file *README.md*. Use Git's `init` (initialize) command to set the directory as a repository. See Table 5.1 for the list of Git commands covered in this chapter.[23] Use Git's `add` command to add a file to the Git repository. For example,

```
# Initialize the Git repository
git init

# Add README to the repository
git add README.md
```

You probably noticed that you always need to put `git` before the command. This tells the shell what program the command is from. When you initialize a folder as a Git repository, a hidden folder called *.git* is added to the directory (see Figure 5.1). This is where all of your changes are kept. If you want to add all of the files in the working directory to the Git repository type:

```
# Add all files to the repository
git add .
```

When we want Git to track changes made to files added to the repository we can use the `commit` command. In Git language we are "committing" the changes to the repository.

```
# Commit changes
git commit -a -m "First Commit, created README file"
```

Note: the files won't appear on GitHub yet. Later in the chapter, we will learn how to push commits to your remote GitHub repository. The `-a` (all) option commits changes made to all of the files that have been added to the repository. You can include a message with the commit using the `-m` option like: `"First Commit, created README file"`. Messages help you remember general details about individual commits. This is helpful when you want to revert to old versions. **Remember:** Git only tracks changes when you commit them.

Finally, you can use the `status` command for details about your repository,

[23] For a comprehensive guide to Git commands, see `http://git-scm.com/`.

TABLE 5.1: A Selection of Git Commands

Command	Description
`add`	Add a file to a Git repository.
`branch`	Create and delete branches.
`checkout`	Checkout a branch.
`clone`	Clone a repository (for example, the remote GitHub version) into the current working directory.
`commit`	Commit changes to a Git repository.
`fetch`	Download objects from the remote (or another) repository.
`.gitignore`	Not a Git command, but a file you can add to your repository to specify what files/file types Git should ignore.
`init`	Initialize a Git repository.
`log`	Show a repo's commit history.
`merge`	Merge two or more commits/branches together.
`pull`	`fetch` data from a remote repository and try to `merge` it with your commits.
`push`	Add committed changes to a remote Git repository, i.e. GitHub.
`remote add`	Add a new remote repository to an existing project.
`rm`	Remove files from Git version tracking.
`status`	Show the status of a Git repository including uncommitted changes made to files.
`tag`	Bookmark particularly significant commits.

Note: when you use these commands in the shell, you will need to precede them with `git` so the shell knows what program they are from.

including uncommitted changes. Generally it's a good idea to use the `-s` (short) option, so that the output is more readable.

```
# Display status
git status -s
```

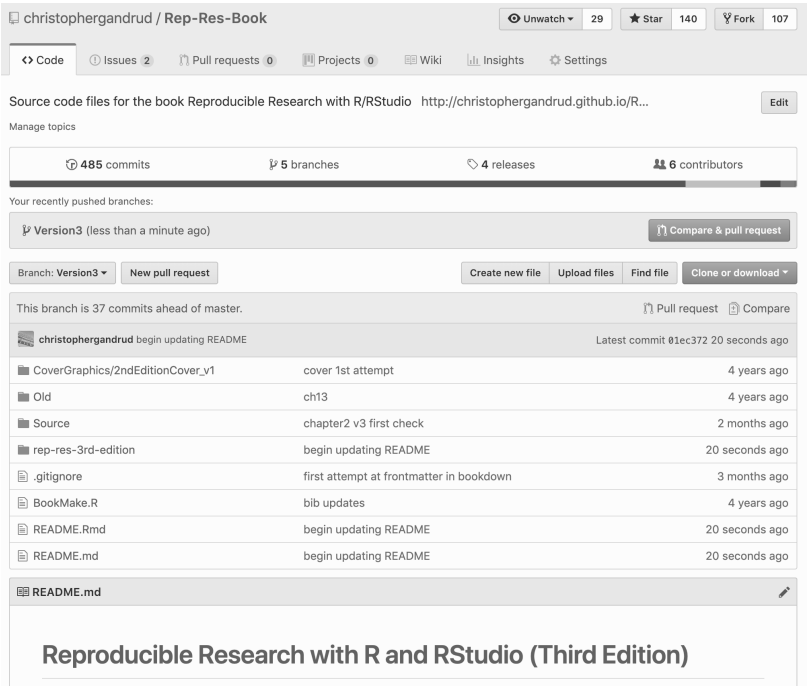


FIGURE 5.2: Part of this Book's GitHub Repository Webpage

Checkout

It is useful to step back for a second and try to understand what Git is doing when you commit your changes. In the hidden *.git*, folder Git is saving all of the information in compressed form from each of your commits into a sub-folder called *objects*. Commit objects[24] are everything from a particular commit. I mean everything. If you delete all of the files in your repository (except for the *.git* folder), you can completely recover all of the files from your most recent commit with the `checkout` command:

[24] Other Git objects include trees (sort of like directories), tags (bookmarks for important points in a repo's history), and blobs (individual files).

```
# Checkout latest
commit git checkout -- .
```

Note that there is a space between the two dashed lines and the period. You can also change to any other commit or any committed version of a particular file with `checkout`. Simply replace the `--` with the commit reference. Note that the period at the end is still very important to include after the commit reference. The commit reference is easy to find and copy from a repository's GitHub webpage (see below for more information on how to create a GitHub webpage).[25] For an example of a GitHub repo webpage, see Figure 5.2. Click on the link that lists the number of repo commits on the left-hand side of the repo's webpage. This will show you all of the commits. A portion of this book's commit history is shown in Figure 5.3. By clicking on the code icon (<>), you can see what the files at any commit looked like. Next to this button is another with a series of numbers and letters. This is the commit's SHA-1 hash.[26] For our purposes, it is the commit's reference number. Click on the button to the left of the SHA to copy it. You can then paste it as an argument to your command. This will revert you to that particular commit. Also include the file name if you want to revert to a particular version of a particular file.

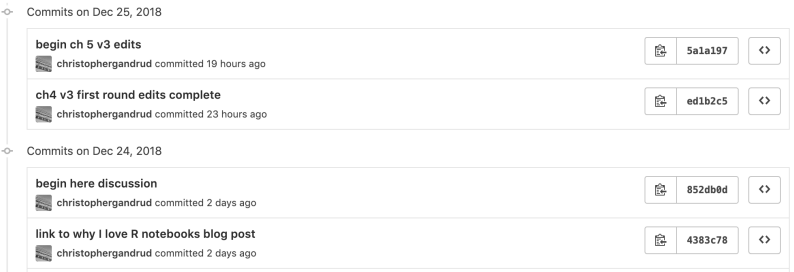


FIGURE 5.3: Part of this Book's GitHub Repository Commit History

Tags

SHA-1 hashes are a bit cumbersome to use as references. What was the hash number for that one commit? To solve this problem you can add bookmarks, known as "tags", to particularly important commits. Imagine we just committed

[25] You can also search your commit history and roll back to a previous commit using only the command-line. To see the commit history, use the `log` command (more details at `http://git-scm.com/book/en/Git-Basics-Viewing-the-Commit-History`). When a repo has many commits, this can be a very tedious command to use, so I highly recommend the GUI version of GitHub or the repo's GitHub website.

[26] Secure Hash Algorithm. This is a unique identifier for each commit.

our first full draft of a project. We want to tag it as version 0.1, i.e. "v0.1". To do this, use Git's `tag` command:

```
# Tag most recent commit v0.1
git tag -a v0.1 -m "First draft"
```

The `-a` option adds the tag `v0.1` and `-m` lets us add a message. Now we can check out this particular commit by using its tag, i.e.:

```
# Checkout v0.1
git checkout v0.1
```

This will create a new "branch" with a generic name *(detached from v0.1)* where you can make changes and commit them. If you plan to check out a previous tagged version and make changes to it, it is a good idea to specifically name the branch using the `-b` argument.[27] For example, to give it the name *v0.1_branch* type:

```
# Checkout v0.1 as v0.1_branch
git checkout v0.1 -b v0.1_branch
```

What is a branch?

Branches

Sometimes you may want to work on an alternative version of your project and then merge changes made to this version back into the main one. For example, the main version could be the most stable current copy of your research, while the alternative version could be a place where you test out new ideas. Git allows you to create a new *branch* (alternative version of the repo) which can be merged back into the *master* (main) branch. To see what branch you are using, type:

```
# Show git branch
git branch
```

```
##   Version3
## * master
##   v3janMinor
```

[27] If you don't, then the new branch will have a "detached head" which will create problems using the branch in the future.

To create a new branch, use the `branch` command. For example, to create a new branch called *test*:

```
# Create test branch
git branch test
```

You can now use `checkout` to switch to this branch.[28] Here is a shortcut for creating and checking out the branch:

```
# Create and checkout test branch
git checkout -b test
```

The `-b` (branch) option for `checkout` creates the new *test* branch before switching to it.

To merge changes you commit in the *test* branch to the *master*, `add` and `commit` your changes, `checkout` the *master* branch, then use the `merge` command.[29]

```
# Add files
git add .

# Commit changes to test branch
git commit -a -m "commit changes to test"

# Checkout master branch
git checkout master

# Merge master and test branches
git merge test
```

Note, when you merge a branch, you may encounter conflicts in the files that make it impossible to smoothly merge the files together. Git will tell you what and where these are; you then need to decide what to keep and what to delete.

Having Git ignore files

There may be files in your repository that you do not want to keep under version control. Maybe this is because they are very large files or cached files from *knitr* or other files that are byproducts of compiling a LaTeX document (see Chapter 8). You also want to ignore files that contain private information. Make

[28] To delete the *test* branch, use the `-d` argument, i.e. `git branch -d Test`.
[29] Any uncommitted changes are merged with a branch when it is checked out.

sure to **never include private information** (e.g. passwords or confidential data) in your Git history. Once they are committed, it will be very difficult to definitively remove them. Once they are on GitHub, they will be publicly accessible.

To have Git ignore particular files, create a file called *.gitignore*.[30] You can either put this file in the repository's parent directory to create a *.gitignore* file for the whole repository or in a sub-directory to ignore files in that sub-directory. In the *.gitignore* file, add ignore rules by including the names of the files that you want to have Git ignore. For example, GitHub has a *.gitignore* file that is useful for ignoring files[31] that we often don't want to commit to our git history when using R and R Markdown:

```
# History files
.Rhistory
.Rapp.history

# Session Data files
.RData

# Example code in package build process
*-Ex.R

# Output files from R CMD build
/*.tar.gz

# Output files from R CMD check
/*.Rcheck/

# RStudio files
.Rproj.user/

# produced vignettes
vignettes/*.html
vignettes/*.pdf

# OAuth2 token
.httr-oauth

# knitr and R markdown default cache directories
/*_cache/
/cache/
```

[30] Note that like *.git*, *.gitignore* files are hidden.

[31] From: `https://github.com/github/gitignore/blob/master/R.gitignore` as of 26 December 2018.

```
# Temporary files created by R markdown
*.utf8.md
*.knit.md
```

The asterisk (`*`) is a "wildcard" and stands for any character. In other words, it tells Git to look for files with any name that end with a specified file extension. This is faster than writing out the full name of every file you want to ignore individually. It also makes it easy to copy the rules into new repos. For example, you'll notice the `*-Ex.R` and `/*_cache/` rules. These tell Git to ignore all of the files with a name ending in *-Ex.R* and all files in subdirectories with a name ending in *_cache*.

Git will not ignore files that have already been committed to a repository. To ignore these files, you will first need to remove them from Git with Git's `rm` (remove) command. If you wanted to remove a file called *example-project.tex* from version tracking type:

```
# Remove example-project.tex from Git version tracking
git rm --cached example-project.tex
```

Using the `-cached` argument tells Git not to track the file, but not delete it.

For more information on *.gitignore* files, see GitHub's reference page on the topic at: `https://help.github.com/articles/ignoring-files/`.

5.3.3 Remote storage on GitHub

So far we've been using repos stored locally. Let's now look at how to also store a repository remotely on GitHub. You can either create a new repository on GitHub and download (`clone`) it to your computer or upload (`push`) an existing repository to a new GitHub remote repo. In both cases, you need to create a new repository on GitHub.

To create a new repository on GitHub, go to your main GitHub account webpage and click the `New repository` button. On the next page that appears, give the repository a name, brief description, and choose whether to make it public or private. If you want to store an existing repository on GitHub, give it the same name as the one that already exists on your computer. If you already have files in your local repository do not check the boxes for creating *README.md*, *LICENSE*, and *.gitignore* files. When you then click `Create Repository`, you will be directed to the repository's GitHub webpage.[32]

[32] Before the repo has any files in it, the webpage will include instructions for how to set it up on your computer.

Clone a new remote repository

If you are working with a new repository and do not have an existing version on your computer, you need to "clone" the GitHub repo to your computer. The repo's GitHub page contains a button called `Clone in Desktop`. Clicking this will open GUI GitHub (if it is installed) and prompt you to specify what directory on your computer you would like to clone the repository into. You can also use the `clone` command in the shell. Imagine that the URL for a repo called *Example Project* is `https://GitHub.com/USER/example-project.git`. To clone it into the */git_repositories* directory type:[33]

```
# Change working directory
cd /git_repositories/

# Clone example-project
git clone https://GitHub.com/USER/example-project.git
```

Push an existing repository to a new GitHub repo

If you already have a repository with files in it on your computer and you want to store them remotely in a new GitHub repo, you need to add the remote repository and `push` your files to it. Type Git's `remote add` command. For example, if your repository's GitHub URL is `https://github.com/USER/example-project.git`, then type:

```
# Change working directory to existing local repo
cd /git_repositories/example-project

# Add a remote (GitHub) repository to an existing repo
git remote add origin https://github.com/USER/example-project.git
```

This will tell your local repository where the remote one is. Finally, push the repository to GitHub:

```
# Push local repository to GitHub for the first time
git push -u origin master
```

The `-u` (upstream tracking) option adds a tracking reference for the upstream (GitHub) repository branches.

[33] If you are on the repo's webpage the URL to copy is under `HTTPS clone URL`.

Pushing commits to a GitHub repo

Once you have your local repository connected to GitHub, you can add new commits with the `push` command. For example, if your current working directory is the Git repo you want to push and you have already added/committed the changes you want to include in the remote repo, type:

```
# Add changes to the GitHub remote master branch
git push origin master
```

The `origin` is the remotely stored repository on GitHub and `master` is the master branch. You can change this to another branch if you'd like. If you have not set up password caching[34] you will now be prompted to give your GitHub username and password.

You can also push your tags to GitHub. To push all of the tags to GitHub, type:

```
git push --tags
```

Now on the repo's GitHub page, there will be a `Tags` section that will allow you to view and download the files in each tagged version of the repository.

5.3.4 Accessing on GitHub

Downloading into R

In general, the process of downloading data directly into R is similar to what we saw earlier for loading data from Dropbox Public folders. We can use the `import()` function. First, we need to find our plain-text data file's *raw* URL. To do this, go to your repository's GitHub site, navigate to the file you want to load, and click the `Raw` button on the right just above the file preview. I have data in comma-separated values format stored in a GitHub repository.[35] The URL for the raw (plain-text) version of the data is `https://raw.githubusercontent.com/christophergandrud/Disproportionality_Data/master/Disproportionality.csv`.[36]

[34] See `https://help.github.com/articles/set-up-git/` for more details.

[35] For full information about the disproportionality data set, please see `http://christophergandrud.github.io/Disproportionality_Data/`.

[36] It has been shortened with Bitly in the example.

```
# Place shortened URLinto url object
url <- "http://bit.ly/14aSjxB"

# Download data
disprop_data <- rio::import(url, format = "csv")

# Show variable names
names(disprop_data)
```

```
## [1] "country"            "iso2c"
## [3] "year"               "disproportionality"
```

`import()` downloaded the most recent version of the file from the master branch.

We can actually use `import()` to download a particular version of a file—from a particular Git commit—directly into R. This makes reproducing a specific result much easier. To do this, you just need to use a file's raw URL from a particular commit. To find a file's particular commit raw URL first click on the file on GitHub's website. Then click the `History` button. This will take you to a page listing all of the file's versions. Click on the git commit hash button next to the version of the file that you want to use. Then click `View file` and finally the `Raw` button to be taken to the text-only version of the file. Copy this page's URL address and use it with `import()`.

For example, I have an old version of the disproportionality data. To download it, I find this particular version of the file's URL and use it in `import()`:

```
# Create object containing the file's URL
old_url <- paste0("https://raw.githubusercontent.com/",
                  "christophergandrud/",
                  "Disproportionality_Data/",
                  "1a59d360b36eade3b183d6336a",
                  "2262df4f9555d1/",
                  "Disproportionality.csv")

# Download old disproportionality data
disprop_old <- rio::import(old_url, format = "csv")
```

In this example I did not shorten the URL, but instead used the `paste0()` function to paste it together.[37] You do not have to do this. I did it here so that the URL would fit on the printed page.

[37] `paste0` is the same as `paste`, but has the argument `sep = ""` so that white space is not placed between the pasted elements.

Viewing files

The GitHub web user interface also allows you, your collaborators (see below) or, if the repo is public, anyone to look at text files from a web browser. Collaborators can actually also create, modify, and commit changes in the web user interface. This can be useful for making small changes, especially from a mobile device without a Git installation. Anyone with a GitHub account can suggest changes to files in a public repository on the repo's website. Simply click the `Edit` button (it looks like a pencil) above the file and make edits. If the person making the edits is not a designated collaborator, their edits will be sent to the repository's owner for approval.[38] This can be a useful way for independent researchers to fix errors.

Collaboration with GitHub

Repositories can have official collaborators that can make changes to files in the repo. Public repositories can have unlimited collaborators. Anyone with a GitHub account can be a collaborator. To add a collaborator to a repository you created, click on the `Settings` button on the repository's website (see Figure 5.2). Then click the `Collaborators` button on the left-hand side of the page. You will be given a box to enter your collaborator's GitHub username. If your collaborator doesn't have a GitHub account, they will have to create a new one. Once you add someone as a collaborator, they can clone the repository onto their computer as you did earlier and push changes.

Syncing a repository

If you and your collaborators are both making changes to the files in a repo you might create conflicting changes, i.e. different changes to the same part of a file. To avoid too many conflicts, it is a good idea to sync your local repository with the remote repository **before** you push your commits to GitHub. Use the `pull` command to sync your local and remote repository. First add and commit your changes, then type:

```
git pull
```

If the files you are pulling conflict with your local files, you will probably want to resolve these in the individual files and commit the changes. When there are merge conflicts, Git adds both versions of a piece of text to the file. You then open the file and decide which version to keep and which one to delete. When the conflicts are resolved and changes committed, push your merged changes up to the remote repository as we did before.

[38] This is called a `pull` request in Git terminology. See the next section for more details.

5.3.5 Summing up the GitHub workflow

We've covered a lot of ground in this section. Let's sum up the basic GitHub workflow you will probably follow once your repo is set up.

1. Add any changes you've made with `git add`.
2. `commit` the changes.
3. `pull` your collaborators' changes from the GitHub repo, resolve any merge conflicts, and `commit` the changes.
4. `push` your changes to GitHub.

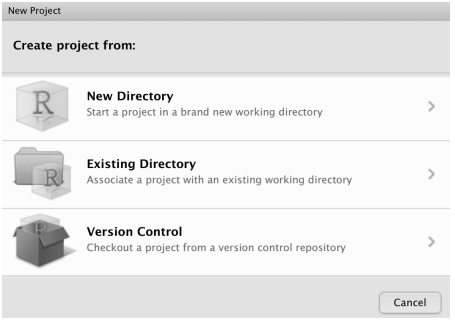


FIGURE 5.4: Creating RStudio Projects

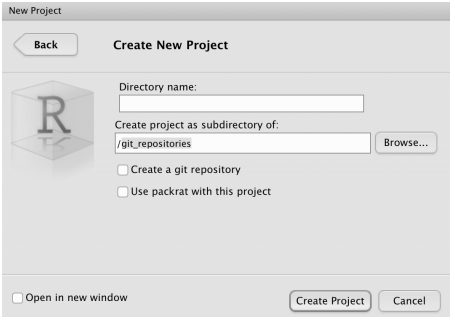


FIGURE 5.5: Creating RStudio Projects in New Directories

5.4 RStudio and GitHub

When you open a Project with a Git repository in RStudio, you will see a new *Git* tab next to *Environment*, *History*, and *Connections* (see Figure 5.6). From here, you can do many of the things we covered in the previous section. Let's look at how to set up and use Git in RStudio Projects.

5.4.1 Setting up Git/GitHub with Projects

You can Git initialize new RStudio Projects, Git initialize existing projects, and create RStudio Projects from cloned repos. When you do any of these things, RStudio automatically adds a *.gitignore* file telling Git to ignore *.Rproj.user*, *.Rhistory*, and *.RData* files.

Git with a new project

To create a new project with Git version control, go to `File` in the RStudio menu bar. Then click `New Project...`. In the box that appears (see Figure 5.4) select `New Directory` `Empty Project`. Enter the Project's name and desired directory. Make sure to check the dialog box for `Create a git repository` (see Figure 5.5).

Git initialize existing projects

If you have an existing RStudio Project and want to add Git version control to it, first go to `Tools` in the RStudio menu bar. Then select `Project Options ...`. Select the `Git/SVN` icon. Finally, select `Git` from the drop-down menu for `Version Control System:`.

Clone repository into a new project

Again go to `File` in the RStudio menu bar to create a new project from a cloned GitHub repository. Then click `New Project...`. Select the `Version Control` option and then `Git`. Finally, paste the repository's URL in the field called `Repository URL:`, enter the directory you would like to locate the cloned repo in, and click `Create Project`.

Add existing Project repository to GitHub

You can push an existing Project repository stored on your computer to a new remote repository on GitHub. To do this, first create a new repo on GitHub with the same name as your RStudio Project (see Section 5.3.3). Then copy the remote repository's URL like we saw before when we cloned a repository from GitHub (see Section 5.3.3). Open a new shell from within RStudio. To do this, click the `Shell` button in the *Git* tab's `More` drop-down menu (it looks like a gear). Now follow the same steps that we used in Section 5.3.3 to connect a locally stored repository to GitHub for the first time.

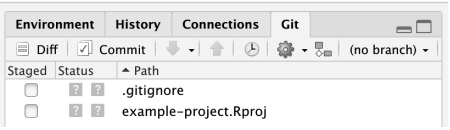


FIGURE 5.6: The Git Repository Tab in RStudio

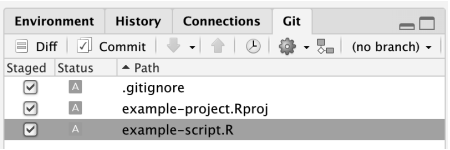


FIGURE 5.7: Adding Changes to the Repository

5.4.2 Using Git in RStudio Projects

The RStudio *Git* tab allows you to do many of the same things with Git that we covered in the previous section. In Figure 5.6 you will see the *Git* tab for a new RStudio Project called *example-project*. It has two files that have not been added or committed to Git. To add and commit the files to the repository, click on the dialog boxes next to the file names. In Figure 5.7 you can see that I've created a new R file called *example-script.R* and clicked the dialog box next to it, along with the other files. The yellow question marks in the top panel have now become green A's for "add". Clicking `Commit` opens a new window called **Review Changes** where you can commit the changes. Simply write a commit message in the box called *Commit Message* in the **Review Changes** window and click `Commit`. If you add file names to the *.gitignore* files, they will not show up in RStudio's *Git* tab.

If you are using a GitHub repo that is associated with a remote repository on GitHub, you can push and pull it with the `Pull Branches` and `Push Branch` buttons in Git menu bar (the down and up arrows respectively). You can use the same buttons in the **Review Changes** window. The *Git* tab also allows

you to change branches, revert to previous commits, add files to `.gitignore`, and view your commit history. You can always use the `More -> Shell...` option to open a new shell with the Project set as the working directory to complete any other Git task you might want to do.

Chapter summary

In this chapter we have primarily learned how to store text-based reproducible research files in ways that allow us and others to access them easily from many locations, enable collaboration, and keep a record of previous versions. In the next chapter, we will learn how to use text-based files to reproducibly gather data that we can use in our statistical analyses.

6

Gathering Data with R

How you gather your data directly impacts how reproducible your research will be. You should try your best to document every step of your data gathering process. Reproduction will be easier if your documentation—especially, variable descriptions and source code—makes it easy for you and others to understand what you have done. If all of your data gathering steps are tied by your source code, then independent researchers (and you) can more easily regather the data. Regathering data will be easiest if running your code allows you to get all the way back to the raw data files, the rawer the better. Of course, this may not always be possible. You may need to conduct interviews or compile information from paper-based archives, for example. Data hosted online may disappear when the host ceases operation. The best you can sometimes do is describe your data gathering process in detail or rehost an original data set. Nonetheless, R's automated data gathering capabilities for internet-based information is extensive. Learning how to take full advantage of these capabilities greatly increases reproducibility and can save you considerable time and effort over the long run.

In this chapter we'll learn strategies for how to gather quantitative data in a fully reproducible way. We'll start by learning how to use data gathering makefiles to organize your whole data gathering process so that it can be completely reproduced. Then we will learn the details of how to actually load data into R from various sources, both locally on your computer and remotely via the internet. In the next chapter (Chapter 7), we'll learn the details of how to clean up raw data so that it can be merged into data frames that can be used for statistical analyses.

6.1 Organize Your Data Gathering: Makefiles

Before getting into the details of using R to gather data, let's start by creating a plan to organize the process. Organizing your data gathering process from the beginning of a research project improves the possibility of reproducibility

and can save you significant effort over the course of the project by making it easier to add and regather data later on.

A key part of reproducible data gathering with R, like reproducible research in general, is segmenting the process into modular files that can all be run in sequence by a common "makefile". In this chapter we'll learn how to create make-like files run exclusively from R as well as GNU Make makefiles,[1] which you run from a shell.[2] Learning how to create R make-like files is fairly easy. Using GNU Make does require learning some more new syntax. However, it has one very clear advantage: it only runs a source code file that has been updated since the last time you ran the makefile. This is very useful if part of your data-gathering process is very computationally and time intensive.

Segmenting your data gathering into modular files and tying them with some sort of makefile allows you to more easily navigate research text and find errors in the source code. The makefile's output is the data set that you'll use in the statistical analyses. There are two types of source code files that the makefile runs: data gathering/cleanup files and merging files. Data cleanup files bring raw individual data sources into R and transform them so that they can be merged with data from the other sources. Many of the R tools for data cleanup and merging will be covered in Chapter 7. In this chapter, we mostly cover the ways to bring raw data into R. Merging files are executed by the makefile after it runs the data gathering/cleanup files.

It's a good idea to have the source code files use very raw data as input. Your source code should avoid directly changing these raw data files. Instead, changes should be put into new objects and data files. Doing this makes it easier to reconstruct the steps you took to create your data set. Also, while cleaning and merging your data you may transform it in unintended ways, for example, accidentally deleting some observations that you wanted to keep. Having the raw data makes it easy to go back and correct your mistakes.

The files for the examples used in this section can be downloaded from GitHub at: `https://github.com/christophergandrud/rep-res-book-v3-examples`.

6.1.1 R Make-like files

When you create make-like files in R to organize and run your data gathering, you usually only need one or two functions, `setwd()` and `source()`. As we talked about in Chapter 4, `setwd()` tells R where to look for and place files. `source()` tells R to run code in an R source code file.[3] Let's see what an R

[1] GNU stands for "GNU's Not Unix", indicating that it is Unix-like.

[2] To standardize things, I use the terms "R make-like file" for files created and run in R and the standard "makefile" for files run by Make.

[3] We use the command more in Chapter 8.

data makefile might look like for a project with a file structure similar to the example project in Figure 4.1. The file paths in this example are for Unix-like systems and the make-like file is called *Makefile.R*.

```
################ # Example R make-like file
# Christopher Gandrud
# Updated 12 January 2019
#################

# Set working directory
setwd("/example-project/data/")

# Gather and cleanup raw data files with a for loop
gatherers <- c("gather-1.R", "gather-2.R", "gather-3.R")
for (i in gatherers) source(i)

# Merge cleaned data frames into data frame object cleaned_data
source("merge-data.R")
```

This code first sets the working directory. Then it runs three source code files to gather data from three different sources. These files gather the data and clean it so that it can be merged. The cleaned data frames are available in the current workspace. Next the code runs the *merge-data.R* file that merges the data frames and saves the output data frame as a CSV formatted file. The CSV file could be the main file we use for statistical analysis. *merge-data.R* also creates a Markdown file with a table describing the variables and their sources. We'll come back to how to create tables in Chapter 9.

You can run the commands in this file one-by-one or run the make-like file by putting it through the `source()` function so that it will run it all at once.

6.1.2 GNU Make

R make-like files are a simple way to tie together a segmented data gathering process. If one or more of the source files that our previous example runs is computationally intensive it is a good idea to run them only when they are updated. However, this can become tedious, especially if there are many segments. The well-established GNU Make command-line program[4] deals with this problem by comparing the output files' time stamps[5] to time stamps of

[4] GNU Make was originally developed in 1977 by Stuart Feldman as a way to compile computer programs from a series of files, its primary use to this day. For an overview, see `http://en.wikipedia.org/wiki/Make_(software)`. For installation instructions, please see Section 1.5.2.

[5] A file's time stamp records the time and date when it was last changed.

the source files that created them. If a source file has a time stamp that is newer than its output, Make will run it. If the source's time stamp is older than its output, Make will skip it.

In Make terminology the output files are called "targets" and the files that create them are called "prerequisites". You specify a "recipe" to create the targets from the prerequisites. The recipe is basically just the code you want to run to make the target file. The general form is:

```
TARGET ... : PREREQUISITE ...
    RECIPE
    ...
    ...
```

Note that, unlike in R, tabs are important in Make. They indicate what lines are the recipe. Make uses the recipe to ensure that targets are newer than prerequisites. If a target is newer than its prerequisite, Make does not run the prerequisite.

The basic idea of reproducible data gathering with Make is similar to what we saw before, with a few twists and some new syntax. Let's see an example that does what we did before: gather data from three sources, clean and merge the data, and save it in CSV format.

Example makefile

The first thing we need to do is create a new file called *Makefile*[6] and place it in the same directory as the data gathering files we already have. The makefile we are going to create runs prerequisite files by the alphanumeric order of their file names. So we need to ensure that the files are named in the order that we want to run them. Now let's look at the actual makefile:

```
#################
# Example Makefile
# Christopher Gandrud
# Updated 1 July 2013
# Influenced by Rob Hyndman (31 October 2012)
# See: http://robjhyndman.com/researchtips/makefiles/
#################

# Key variables to define
RDIR = .
MERGE_OUT = merge-data.Rout
```

[6] Alternatively, you can call the file *GNUmakefile* or *makefile*.

```
# Create list of R source files
RSOURCE = $(wildcard $(RDIR)/*.R)

# Files to indicate when the RSOURCE file was run
OUT_FILES = $(RSOURCE:.R=.Rout)

# Default target
all: $(OUT_FILES)

# Run the RSOURCE files
$(RDIR)/%.Rout: $(RDIR)/%.R
    R CMD BATCH $<

# Remove Out Files
clean:
    rm -fv $(OUT_FILES)

# Remove merge-data.Rout
cleanMerge:
    rm -fv $(MERGE_OUT)
```

Ok, let's break down the code. The first part of the file defines variables that will be used later on. For example, in the first line of executable code (`RDIR = .`) we create a simple variable[7] called `RDIR` with a period (`.`) as its value. In Make and Unix-like shells, periods indicate the current directory. The next line allows us to specify a variable for the outfile created by running the *merge-data.R* file. This will be useful later when we create a target for removing this file to ensure that the *merge-data.R* file is always run.

The third executed line (`RSOURCE:= $(wildcard $(RDIR)/*.R)`) creates a variable containing a list of all the names of files with the extension `.R`, i.e. our data gathering and merge source code files. This line has some new syntax, so let's work through it. In Make (and Unix-like shells generally) a dollar sign (`$`) followed by a variable name substitutes the value of the variable in place of the name.[8] For example, `$(RDIR)` inserts the period `.` that we defined as the value of `RDIR` previously. The parentheses are included to clearly demarcate where the variable name begins and ends.[9]

You may remember the asterisk (`*`) from the previous chapter. It is a "wildcard",

[7] Simple string variables are often referred to as "macros" in GNU Make. A common convention in Make and Unix-like shells generally is to use all caps for variable names.

[8] This is a kind of parameter expansion. For more information about parameter expansion, see Frazier (2008).

[9] Braces (`{}`) are also sometimes used for this.

a special character that allows you to select file names that follow a particular pattern. Using `*.R` selects any file name that ends in `.R`.

Why did we also include the actual word `wildcard`? The `wildcard` function is different from the asterisk wildcard character. The function creates a list of files that match a pattern. In this case the pattern is `$(RDIR)/*.R`. The general form for writing the `wildcard` function is: `$(wildcard PATTERN)`.

The third line (`OUT_FILES = $(RSOURCE:.R=.Rout)`) creates a variable for the `.Rout` files that Make will use to tell how recently each R file was run.[10] `$(RSOURCE:.R=.Rout)` is a variable that uses the same file name as our RSOURCE files, but with the file extension `.Rout`.

The second part of the makefile tells Make what we want to create and how to create it. In the line `all: $(OUT_FILES)` we are specifying the makefile's default target. Targets are the files that you instruct Make to make. `all:` sets the default target; it is what Make tries to create when you enter the command `make` in the shell with no arguments. We will see later how to instruct Make to compile different targets.

The next two executable lines (`$(RDIR)/%.Rout: $(RDIR)/%.R` and `R CMD BATCH $<`) run the R source code files in the directory. The first line specifies that the `.Rout` files are the targets of the `.R` files. The percent sign (`%`) is another wildcard. Unlike the asterisk, it replaces the selected file names throughout the command used to create the target.

The dollar and less-than signs (`$<`) indicate the first prerequisite for the target, i.e. the `.R` files. `R CMD BATCH` is a way to call R from a Unix-like shell, run source files, and output the results to other files.[11] The out-files it creates have the extension `.Rout`.

The next two lines specify another target: `clean`. When you type `make clean` into your shell, Make will follow the recipe: `rm -fv $(OUT_FILES)`. This removes (deletes) the `.Rout` files. The `f` argument (force) ignores files that don't exist and the `v` argument (verbose) instructs Make to tell you what is happening when it runs. When you delete the `.Rout` files, Make will run all of the `.R` files the next time you call it.

The last two lines help us solve a problem created by the fact that our simple makefile doesn't push changes downstream. For example, if we make a change to *gather-2.R* and run `make`, only *gather-2.R* will be rerun. The new data frame will not be added to the final merged data set. To overcome this problem, the

[10] The R out-file contains all of the output from the R session used while running the file. These can be a helpful place to look for errors if your makefiles give you an error like `make: *** [gather.Rout] Error 1`.

[11] You will need to make sure that R is in your PATH. Setting this up is different on different systems. If on Mac and Linux you can load R from the Terminal by typing `R`, R is in your PATH. The R installation usually sets this up correctly. There are different methods for changing the file path on different versions of Windows.

last two lines of code create a target called `cleanMerge`; this removes only the *merge-data.Rout* file.

Running the Makefile

To run the makefile for the first time, change the working directory to where the file is and type `make` into your shell. It will create the CSV final data file and four files with the extension `.Rout`, indicating when the segmented data gathering files were last run.[12]

When you run `make` in the shell for the first time, you should get the output:

```
## R CMD BATCH gather-1.R
## R CMD BATCH gather-2.R
## R CMD BATCH gather-3.R
## R CMD BATCH merge-data.R
```

If you run it a second time without changing the R source files, you will get the following output:

```
## make: Nothing to be done for 'all'.
```

To remove all of the `.Rout` files, set the make target to `clean`:

```
make clean

## rm -fv ./gather-1.Rout ./gather-2.Rout ./gather-3.Rout
## ./merge-data.Rout
## ./gather-1.Rout
## ./gather-2.Rout
## ./gather-3.Rout
## ./merge-data.Rout
```

If we run the following code:

```
# Remove merge-data.Rout and make all R source files
make cleanMerge all
```

then Make will first remove the *merge-data.Rout* file (if there is one) and then run all of the R source files as need be. *merge-data.R* will always be run. This ensures that changes to the gathered data frames are updated in the final merged data set.

[12] If you open these files, you will find the output from the R session used when their source file was last run.

Makefiles and RStudio Projects

You can run makefiles from RStudio's *Build* tab. For the type of makefile we have been using, the main advantage of running it from within RStudio is that you don't have to toggle between RStudio and the shell. Everything is in one place. Imagine that the directory with our makefile is an RStudio Project. If a Project already contains a makefile, RStudio will automatically open a *Build* tab on the *Environment/History/Connections* pane, the same place where the *Git* tab appears (see Figure 6.1).[13]

The *Build* tab has buttons you can click to `Build All` (this is equivalent to `make all`), and, in the `More` drop-down menu, `Clean all` (i.e., `make clean`) and `Clean and Rebuild` (i.e., `make clean all`). As you can see in Figure 6.1, the *Build* tab shows you the same output you get in the shell.

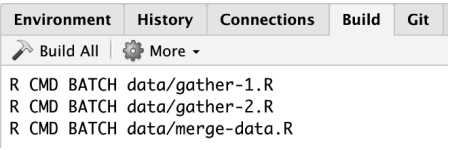


FIGURE 6.1: The RStudio Build Pane

Other information about makefiles

Note that Make relies heavily on commands and syntax of the shell program that you are using. The above example was written and tested on a Mac. It should work on other Unix-like computers without modification.

You can use Make to build almost any project from the shell, not just to run R source code files. It was an integral part of early reproducible computational research (Fomel and Claerbout, 2009; Buckheit and Donoho, 1995). Rob Hyndman more recently posted a description of the makefile he uses to create a project with R and LaTeX.[14] The complete source of information on GNU Make is the official online manual. It is available at: `http://www.gnu.org/software/make/manual/`.

[13] If a project doesn't have a makefile, you can still set up RStudio Build. Click on `Build` in the Menu bar then `Configure Build Tools . . ..` Select `Makefile` from the drop-down menu, then `Ok`. You will still need to manually add a Makefile in the Project's root directory.

[14] See his blog at: `https://robjhyndman.com/hyndsight/makefiles/`, which was posted 31 October 2012. This method largely replicates what we do in this book with *knitr*. Nonetheless, it has helpful information about Make that can be used in other tasks. It was in fact helpful for writing this section of the book.

6.2 Importing Locally Stored Data Sets

Now that we've covered the big picture, let's learn the different tools you will need to know to gather data from different types of sources. The most straightforward place to load data from is a local file, e.g. one stored on your computer. Though storing your data locally does not really encourage reproducibility, most research projects will involve loading data this way at some point. The tools you will learn for importing locally stored data files will also be important for most of the other methods further on.

Data stored in plain-text files on your computer can be loaded into R using the `read.table()` function. For example, imagine we have a CSV file called *test-data.csv* stored in the current working directory. To load the data set into R, type:

```
test_data <- read.table("test-data.csv", sep = ",",
                        header = TRUE)
```

If you are using RStudio, you can do the same thing with drop-down menus. To open a plain-text data file, click on `Environment Import Dataset... From Text File...`. In the box that pops up, specify the column separator, whether or not you want the first line to be treated as variable labels, and other options. This is initially easier than using `read.table()`, but it is much less reproducible.

If the data is not stored in plain-text format but is instead saved in a format created by another statistical program such as SPSS, SAS, or Stata, we can import it using commands in the *foreign* package. For example, imagine we have a data file called *data-1.dta* stored in our working directory. This file was created by the Stata statistical program. To load the data into an R data frame object called *stata_data*, type:

```
# Load foreign package
library(foreign)

# Load Stata formatted data
stata_data <- read.dta(file = "data-1.dta")
```

As you can see, functions in the *foreign* package have similar syntax to `read.table()`. To see the full range of commands and file formats that the *foreign* package supports, use the following:

```
library(help = "foreign")
```

Typically an even simpler solution is to use `import()` from the *rio* package. It will automatically try to find the right way to parse whatever data format you give it. For example:

```
stata_data <- rio::import("data-1.dta")
```

If you have data stored in a spreadsheet format such as Excel's *.xlsx*, it may be best to first clean up the data in the spreadsheet program by hand and then save the file in plain-text format. When you clean up the data, make sure that the first row has the variable names and that observations are in the following rows. Also, remove any extraneous information such as notes, colors, and so on that will not be part of the data frame. `import()` can also attempt to import *.xlsx* files. This is much easier if they are cleaned up to resemble text files.

To aid reproducibility, locally stored data should include careful documentation of where the data came from and how, if at all, it was transformed before it was loaded into R. Ideally, the documentation would be written in a text file saved in the same directory as the raw data file.

6.3 Importing Data Sets from the Internet

There are many ways to import data that is stored on the internet directly into R. We have to use different methods depending on where and how the data is stored.

6.3.1 Data from non-secure (*http*) URLs

Importing data into R that is located at a non-secure URL[15]—ones that start with *http*—is straightforward, provided that:

- the data is stored in a simple format, e.g. plain-text,
- the file is not embedded in a larger HTML website.

We already discussed the first issue in detail. You can determine if the data file is embedded in a website by opening the URL in your web browser. If you

[15] URL stands for "Uniform Resource Locator".

only see the raw plain-text data, you are probably good to go. To import the data, include the URL as the file's name in your `read.table()` function.

6.3.2 Data from secure (*https*) URLs

Storing data at non-secure URLs is now very uncommon. Services like Dropbox, GitHub, and Dataverse store their data at secure URLs. You can tell if the data is stored at a secure web address if it begins with `https` rather than `http`. We have to use a different function to download data from secure URLs.

As we saw last chapter, in Section 5.2.2, `import()` has no problem gathering data from secure URLs, e.g.:

```
# Place the URL into the object fin_url
fin_url <- "https://bit.ly/2xlQ2j5"

# Download data
fin_regulator <- import(fin_url, format = "csv")
```

6.3.3 Compressed data stored online

Sometimes data files are large, making them difficult to store and download without compressing them. There are a number of compression methods such as Zip and Tar.[16] Zip files have the extension `.zip` and Tar files use extensions such as `.tar` and `.gz`. In most cases[17] you can download, decompress, and create data frame objects from these files directly in R. To do this, you need to[18]

- create a temporary file with `tempfile()` to store the zipped file, which you will later remove with `unlink()` at the end,
- download the file with `download.file()`,
- decompress the file with one of the commands in base R,[19]
- read the file with `read.csv()` or `import()`.

[16] Tar archives are sometimes referred to as 'tar balls'.

[17] Some formats that require the *foreign* package to open are more difficult. This is because functions such as for opening Stata files only accept file names or URLs as arguments, not connections, which you create for unzipped files.

[18] The description of this process is based on a Stack Overflow comment by Dirk Eddelbuettel (see `http://stackoverflow.com/questions/3053833/using-r-to-download-zipped-data-file-extract-and-import-data?answertab=votes#tab-top`, posted 10 June 2010.)

[19] To find a full list of functions, type `?connections` into the R console.

The reason that we have to go through so many extra steps is that compressed files are more than just a single file and contain a number of files as well as metadata.

Let's download a compressed file called *uds_summary.csv* from (Pemstein et al., 2010). It's in a compressed file called *uds_summary.csv.gz*. At the time of writing, the file's URL address is `http://www.unified-democracy-scores.org/files/20140312/z/uds_summary.csv.gz`.

```
# For simplicity, store the URL in an object called 'URL'
URL <- paste0("http://www.unified-democracy-scores.org/",
              "files/20140312/z/uds_summary.csv.gz")

# Create a temporary file called 'temp' to put the zip file into.
temp <- tempfile()

# # Download the compressed file into the temporary file
download.file(URL, temp)

# # Decompress the file and convert it into a data frame
uds_data <- read.csv(gzfile(temp, "uds_summary.csv"))

# # Delete the temporary file
unlink(temp)

# # Show variables in data
names(uds_data)
```

Note I used `paste0()` to split the URL over two lines so I could print the whole URL on this page.

6.3.4 Data APIs and feeds

There are a growing number of packages that can gather data directly from a variety of internet sources and import them into R. Most of these packages use the sources' web application programming interfaces (APIs). Web APIs allow programs to interact with a website. Needless to say, this is great for reproducible research. It not only makes the data gathering process easier as you don't have to download many Excel files and fiddle around with them before even getting the data into R, but it also makes replicating the data gathering process much more straightforward and makes it easy to update data sets when new information becomes available.

Warning: An R package that downloads data from an API will only work as

long as the package maintainer keeps up with changes made to the API and the service the API calls still exists. If one of these conditions doesn't hold, the function call will break. It will not be possible to easily reproduce the data gathering process. Because of these threats to reproducibility, I recommend saving a copy of the data you download and considering making it available for replication.

API R package example

Each of these packages has its own syntax and it isn't possible to go over all of them here. Nonetheless, let's look at an example of accessing World Bank data with the *WDI* to give you a sense of how these packages work. Imagine that we want to gather data on fertilizer consumption. We can use *WDI*'s `WDIsearch()` function to find fertilizer consumption data available at the World Bank:

```
# Load WDI package
library(WDI)

# Search World Bank for fertilizer consumption data
WDIsearch("fertilizer consumption")
```

```
##      indicator
## [1,] "AG.CON.FERT.ZS"
## [2,] "AG.CON.FERT.PT.ZS"
## [3,] "AG.CON.FERT.MT"
##      name
## [1,] "Fertilizer consumption (kilograms per hectare of arable land)"
## [2,] "Fertilizer consumption (% of fertilizer production)"
## [3,] "Fertilizer consumption (metric tons)"
```

This call returns a selection of indicator numbers and their names.[20] Let's gather data on countries' fertilizer consumption in kilograms per hectare of arable land. The indicator number for this variable is: `AG.CON.FERT.ZS`. We can use the function `WDI()` to gather the data and put it in an object called *fert_cons_data*.

```
fert_cons_data <- WDI(indicator = "AG.CON.FERT.ZS",
                      start = 2010, end = 2016)
```

The `start` and `end` arguments allow us to set the starting and ending year of the data to download.

[20] You can also search the World Bank Development Indicators website. The indicator numbers are at the end of each indicator's URL.

The data we downloaded looks like this:

```
head(fert_cons_data)
```

```
##   iso2c    country AG.CON.FERT.ZS year
## 1    1A Arab World          68.36 2016
## 2    1A Arab World          73.26 2015
## 3    1A Arab World          68.16 2014
## 4    1A Arab World          62.40 2013
## 5    1A Arab World          64.10 2012
## 6    1A Arab World         104.90 2011
```

You can see that `WDI` has downloaded data for four variables: **iso2c**,[21] **country**, **AG.CON.FERT.ZS** and **year**.

6.4 Advanced Automatic Data Gathering: Web Scraping

If a package does not already exist to access data from a particular website, there are other ways to automatically "scrape" data with R. This section briefly discusses some of R's web scraping tools and techniques to get you headed in the right direction to do more advanced data gathering.

The general process

Simple web scraping involves downloading a file from the internet, parsing it (i.e. reading it), and extracting the data you are interested in then putting it into a data frame object. We already saw a simple example of this when we downloaded data from a secure HTTPS website.

The complexity of this process depends on how structured the data is. If the data is in a CSV file, then all we need is the `import()` function. Less structured data requires more effort to download and parse. For example, data may be stored in an HTML formatted table within a more complicated HTML marked up webpage. The *XML* package (Temple Lang, 2020) has a number of useful functions such as `readHTMLTable()` for parsing and extracting this kind of data. The *XML* package also clearly has functions for handling XML—Extensible Markup Language—formatted data. In addition, the helpful *rvest* (Wickham, 2019b) package provides set of functions with capabilities similar to

[21] These are the countries' or regions' International Standards Organization's two-letter codes. For more details, see `https://www.iso.org/iso-3166-country-codes.html`.

and often more capable than *XML*. If the data is stored in JSON—JavaScript Object Notation—you can read it with a package like *jsonlite* (Ooms et al., 2018).

There are more websites with APIs than R packages designed specifically to access each one. If an API is available, the *httr* package (Wickham, 2019a) may be useful.

More tools to learn for web scraping

Beyond learning about the various R packages that are useful for R web scraping, an aspiring web scraper should probably invest time learning a number of other skills:

- HTML: Obviously you will encounter a lot of HTML markup when web scraping. Having a good understanding of the HTML markup language will be very helpful. W3 Schools (`https://www.w3schools.com/`) is a free resource for learning HTML as well as JSON, JavaScript, XML, and other languages you will likely come across while web scraping.
- Regular Expressions: Web scraping often involves finding character patterns. Some of this is done for you by the R packages above that parse text. There are times, however, when you are looking for particular patterns, like tag IDs, that are particular to a given website and change across the site based on a particular pattern. You can use regular expressions to deal with these situations. R has a comprehensive, if bare-bones, introduction to regular expressions. To access it, type `?regex` into your R console.
- Looping: Web scraping often involves applying a function to multiple things, e.g. tables or HTML tags. To do this in an efficient way, you will need to use loops and `apply` functions. Matloff (2011) provides a comprehensive overview. The *dplyr* (Wickham et al., 2019b) and *purrr* (Henry and Wickham, 2019) packages are useful for data frame and vector manipulation.
- Finally, Munzert et al. (2015) provide a comprehensive overview of web scraping and text mining with R.

Chapter summary

In this chapter, we learned how to reproducibly gather data from a number of sources. If the data we are using is available online, we may be able to create really reproducible data gathering files. These files have commands that others can execute with makefiles that allow them to actually regather the exact data we used. The techniques we can use to gather online data also make it easy to update our data when new information becomes available. Of course, it may not always be possible to have really reproducible data gathering.

Nonetheless, you should always aim to make it clear to others (and yourself) how you gathered your data. In the next chapter, we will learn how to clean and merge multiple data files so that they can easily be used in our statistical analyses.

7

Preparing Data for Analysis

Once we have gathered the raw data that we want to include in our statistical analyses, we generally need to clean it up so that it can be merged into a single data set that we can easily use for statistical analysis. In this chapter we will learn how to create the data gathering and merging files we saw in the last chapter. This includes recoding and transforming variables in the data set so that the data sets can be easily merged. This will also be useful information in later chapters as well. If you are very familiar with data transformations in R, you may want to skip to the next chapter.

7.1 Cleaning Data for Merging

In order to successfully merge two or more data frames, we need to make sure that they are in the same format. Let's look at some of the important formatting issues and how to reformat your data frames so that they can be easily merged.

7.1.1 Get a handle on your data

Before doing anything to your data, it is a good idea to 'look at it' to see what needs to be done. Taking a little time to become acquainted with your data will help you avoid many error messages and much frustration.

You could type a data frame object's name into the R console. This will print the entire data frame in your console. For data frames with more than a few variables and observations, this is impractical. We have already seen a number of functions that are useful for looking at parts of your data. As we saw in Chapter 3, the `names()` function shows you the variable names in a data frame object. The `head()` function shows the names plus the first few observations in a data frame. `tail()` shows the last few. `str()` returns a summary of a data frame, including the number of observations and variables as well as the variable types.

Use the `dim()` (dimensions) function to quickly see the number of observations and variables (the number of rows and columns) in a data frame object. For example, let's test out `dim()` with the *fert_cons_data* object we created in Chapter 6:

```
dim(fert_cons_data)
```

```
## [1] 1848    4
```

The first number is the number of rows in the data frame (1848), and the second is the number of columns (4). You can also use the `nrow()` function to find just the number of rows and `ncol()` to see only the columns.

The `summary()` function is especially helpful for seeing basic descriptive statistics for all of the variables in a data frame and also the variable types. Here is an example:

```
summary(fert_cons_data)
```

```
##     iso2c              country           AG.CON.FERT.ZS
##  Length:1848        Length:1848        Min.   :    0
##  Class :character   Class :character   1st Qu.:   27
##  Mode  :character   Mode  :character   Median :  107
##                                        Mean   :  275
##                                        3rd Qu.:  181
##                                        Max.   :33067
##                                        NA's   :414
##       year
##  Min.   :2010
##  1st Qu.:2011
##  Median :2013
##  Mean   :2013
##  3rd Qu.:2015
##  Max.   :2016
##
```

We can immediately see that the variables **iso2c** and **country** are character strings. Because `summary()` is able to calculate means, medians, and so on for **AG.CON.FERT.ZS** and **year**, we know they are numeric. Have a look over the summary to see if there is anything unexpected like lots of missing values (**NA's**) or unusual maximum and minimum values. You can of course, run `summary()` on a particular variable by using the component selector (`$`):

```
# Summarize fertilizer consumption variable from fert_cons_data
summary(fert_cons_data$AG.CON.FERT.ZS)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0      27     107     275     181   33067
##    NA's
##     414
```

We'll come back to why knowing this type of information is important for merging and data analysis later in this chapter.

Another important function for quickly summarizing a data frame is `table()`. This creates a contingency table with counts of the number of observations per combination of factor variables.

You can view a portion of a data frame object with `View()` This will open a new window that lets you see a selection of the data frame. If you are using RStudio, you can click on the data frame in the *Environment* tab and you will get something similar. Note that neither of these viewers are interactive in that you can't use them to manipulate the data. They are only data viewers. To be able to see similar windows that you can interactively edit, use the `fix()` function in the same way that you use `View()`. This can be useful for small edits, but remember that the edits are not reproducible.

Tibbles

Most of these data summary capabilities come "for free" when you use an alternate type of data frame called a "tibble" (Müller and Wickham, 2019) For example:

```
# Create example tibble data frame
tbl_ex <- tibble::tibble(numbers = 1:26, letters = letters)

tbl_ex
```

```
## # A tibble: 26 x 2
##    numbers letters
##      <int> <chr>
##  1       1 a
##  2       2 b
##  3       3 c
##  4       4 d
##  5       5 e
##  6       6 f
##  7       7 g
##  8       8 h
##  9       9 i
## 10      10 j
```

```
## # ... with 16 more rows
```

Entering a tibble's object name in the console returns the condensed output, the data dimmensions, and the variable types with the first 10 entries.

Tibbles are the data structure favored by the tidy data/tidyverse R data paradigm (Wickham, 2014b). We will work with other packages of the Tidyverse, e.g. *dplyr* and *ggplot2*, in later chapters. Note that these packages often work with traditional data frames as well (or will convert data frames to tibbles automatically).

7.1.2 Reshaping data

It is often a good idea if your data sets are kept in data frame type objects if that is the format you will use for analysis. See Chapter 3 for how to convert objects into data frames with the `data.frame()` function. Not only do data sets (generally) need to be stored in data frame objects, they also need to have the same layout before they can be merged. Most R statistical analysis tools assume that your data is in "long" format. For an excellent discussion of ideal data formats for statistical analysis, see Wickham (2014b). Long formatted data usually has columns that represent variables. Rows contain specific observations. For example:

TABLE 7.1: Long-Formatted Data Example

Subject	Variable1
Subject1	
Subject2	
Subject3	
...	

In this chapter we will mostly use examples of time-series cross-sectional data (TSCS) that we want to have in long-format. Long-formatted TSCS data is a data frame where rows identify observations of a particular subject at particular points in time and there are multiple observations per subject (see Table 7.2). In this chapter our TSCS data is specifically going to be countries that are observed in multiple years.

If one of our raw data sets is not in this format, then we will need to reshape or, using Wickham's (2014b) terminology, "tidy" it. Some data sets are in "wide" format, where one of the columns in what would be long formatted data

TABLE 7.2: Long-Formatted Time-Series Cross-Sectional Data Example

Subject	Time	Variable1
Subject1	1	
Subject1	2	
Subject1	3	
Subject2	1	
Subject2	2	
Subject2	3	
...		

is "widened" to cover multiple columns. This is confusing to imagine without an example. Table 7.3 shows how Table 7.2 looks when we widen the time variable.

TABLE 7.3: Wide-Formatted Data Example

Subject	Time1	Time2	Time3
Subject1			
Subject2			
...			

The process of tidying data often causes confusion and frustration. Though probably never easy, there are a number of useful R functions for changing data from wide-format to long and vice versa. These include the matrix transpose function (`t()`)[1] and the `reshape()` function, both are loaded in R by default. *tidyr* (Wickham and Henry, 2019) is a very helpful package for reshaping data. This package has more general tools for reshaping data and is worth investing some time to learn well. In this section, we will look at *tidyr*'s `pivot_longer()` function and use it to reshape a TSCS data frame from wide- to long-format.

[1] See this example by Rob Kabacoff: `http://www.statmethods.net/management/reshape.html`. Note also that because the matrix transpose function is denoted with `t`, you should not give any object the name *t*.

We will also encounter this function again in Chapter 10 when we want to transform data so that it can be graphed. Note that if you want to go from long to wide-format, use *tidyr*'s `pivot_wider()` function.

For illustration, let's imagine that the fertilizer consumption data we previously downloaded from the World Bank is in wide, rather than long, format and is in a data frame object called *fert_wide*. It looks like this:

```
fert_wide[, 1:4]
```

```
## # A tibble: 264 x 4
##    iso2c country             `2016` `2015`
##    <chr> <chr>                <dbl>  <dbl>
##  1 AF    Afghanistan          12.2   12.1
##  2 AL    Albania             126.   108.
##  3 DZ    Algeria              22.3   23.4
##  4 AS    American Samoa       NA     NA
##  5 AD    Andorra              NA     NA
##  6 AO    Angola                7.98   8.05
##  7 AG    Antigua and Barbuda  13.9    5.48
##  8 1A    Arab World           68.4   73.3
##  9 AR    Argentina            50.3   27.6
## 10 AM    Armenia             110.    53.0
## # ... with 254 more rows
```

See the chapter's Appendix for the full code I used to reshape the data from long- to wide-format.

Let's think about how we want to tidy the data. We want to create two new columns from the many columns that are now labeled by year. Let's call the new columns **Year** and **Fert**. The **Year** column will clearly contain the year of each observation and **Fert** will contain the fertilizer consumption. **Year** will be what `pivot_longer()` calls the variable's "name" and **Fert** is the "value". In our *fert_wide* data, we don't want the **iso2c** and **country** variables to be gathered. These variables identify the data set's subjects. So we can tell `pivot_longer()` that we only want the columns with the between **2016** and **2010** to be used for the long variable. Note that the back ticks in the code below allow us to specify numeric values as column names.

```
# Gather fert_wide
fert_long <- tidyr::pivot_longer(fert_wide,
                                 cols = `2016`:`2010`,
                                 names_to = "Year",
                                 values_to = "Fert")
```

```
fert_long
```

```
## # A tibble: 1,848 x 4
##    iso2c country     Year    Fert
##    <chr> <chr>       <chr>  <dbl>
##  1 AF    Afghanistan 2016   12.2
##  2 AF    Afghanistan 2015   12.1
##  3 AF    Afghanistan 2014   12.1
##  4 AF    Afghanistan 2013   14.9
##  5 AF    Afghanistan 2012   28.1
##  6 AF    Afghanistan 2011    6.61
##  7 AF    Afghanistan 2010    4.25
##  8 AL    Albania     2016  126.
##  9 AL    Albania     2015  108.
## 10 AL    Albania     2014   88.4
## # ... with 1,838 more rows
```

7.1.3 Renaming variables

Frequently, in the data cleaning process we want to change the names of our variables. This will make our data easier to understand and may even be necessary to properly combine data sets (see below). In the previous example, for instance, our `fert_long` data frame has two variables: **Year** and **Fert**. Imagine, for the sake of demonstration, that we want to rename them **year** and **fert_cons**. Renaming data frame variables is straightforward with the `rename()` function in the *dplyr* package (Wickham et al., 2019b). To rename both **variable** and **value** with the `rename()` function type:

```
fert_long <- dplyr::rename(fert_long,
                           year = Year,
                           fert_cons = Fert)

fert_long
```

```
## # A tibble: 1,848 x 4
##    iso2c country     year  fert_cons
##    <chr> <chr>       <chr>     <dbl>
##  1 AF    Afghanistan 2016       12.2
##  2 AF    Afghanistan 2015       12.1
##  3 AF    Afghanistan 2014       12.1
##  4 AF    Afghanistan 2013       14.9
##  5 AF    Afghanistan 2012       28.1
```

```
##  6 AF    Afghanistan 2011       6.61
##  7 AF    Afghanistan 2010       4.25
##  8 AL    Albania     2016     126.
##  9 AL    Albania     2015     108.
## 10 AL    Albania     2014      88.4
## # ... with 1,838 more rows
```

7.1.4 Ordering data

You may have noticed that as a result of gathering *fert_wide* the data is now ordered by country-year. Imagine that for some substantive reason that makes the data easier to read, we rather want it ordered by year-country. Though not required for merging in R, some statistical analyses assume that the data is ordered in a specific way.

We can order observations in our data set using the `order()` function. For example, to order *fert_long* by year-country, we type:

```
# Order fert_long by year-country
fert_long <- fert_long[order(fert_long$year,
                                    fert_long$country), ]

head(fert_long)
```

```
## # A tibble: 6 x 4
##   iso2c country        year  fert_cons
##   <chr> <chr>          <chr>     <dbl>
## 1 AF    Afghanistan    2010       4.25
## 2 AL    Albania        2010      97.3
## 3 DZ    Algeria        2010      19.5
## 4 AS    American Samoa 2010      NA
## 5 AD    Andorra        2010      NA
## 6 AO    Angola         2010       8.43
```

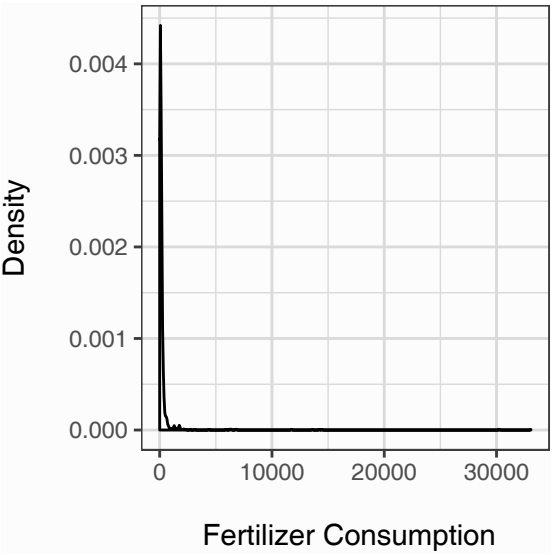
dplyr has a function called `arrange()` that can also be useful for ordering your data. `arrange()`'s syntax is much cleaner and easier to remember for data frames than the operation we did with `order()`. To arrange the *fert_long* data back to country-year with `arrange()` use:

```
fert_long <- dplyr::arrange(fert_long, country, year)
```

To arrange a variable in descending order, place it in the `desc()` function from *dplyr*, e.g. `arrange(fert_long, country, desc(year))`.

7.1.5 Subsetting data

Sometimes you may want to use only a subset of a data frame. For example, the density plot in the following figure shows us that the *fert_long* data has a few very extreme values (see the chapter's Appendix for the source code to create this figure).



We can use the `subset()` function to examine these outliers, for example, countries that have fertilizer consumption greater than 1000 kilograms per hectare.

```
# Create outlier data frame
fert_outliers <- subset(x = fert_long,
                        fert_cons > 1000)

fert_outliers
```

```
## # A tibble: 46 x 4
##    iso2c country             year  fert_cons
##    <chr> <chr>               <chr>     <dbl>
##  1 BH    Bahrain             2010      1721.
##  2 BH    Bahrain             2011      1178.
##  3 BH    Bahrain             2012      1553.
##  4 BH    Bahrain             2013      1606.
##  5 BH    Bahrain             2014      1319.
##  6 BH    Bahrain             2015      1319.
##  7 BH    Bahrain             2016      1319.
```

```
##  8 HK    Hong Kong SAR, China 2012      1307.
##  9 HK    Hong Kong SAR, China 2014      1974.
## 10 HK    Hong Kong SAR, China 2015      2334.
## # ... with 36 more rows
```

If we want to drop these outliers from our data set, we can use `subset()` again:

```
fert_long_sub <- subset(x = fert_long,
                        fert_cons <= 1000)
```

In this example, non-country units like "Arab World" are included. We might also want to drop these units with `subset()`. For example:

```
fert_long_sub <- subset(x = fert_long_sub,
                        country != "Arab World")
```

We can also use `subset()` to remove observations with missing values (`NA`) for **fert_cons**.

```
# Remove observations of fert_cons
# with missing values
fert_long_sub <- subset(x = fert_long_sub,
                        !is.na(fert_cons))

# Summarize fert_cons
summary(fert_long_sub$fert_cons)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0    26.2   103.0   129.7   170.9   900.0
```

Let's step back. I've introduced a number of new logical operators and a new function in the subsetting examples. The first example included the greater than sign (`>`). The second example included the less than or equal to operator: `<=`. The third example included the not equal operator: `!=`. In R, exclamation points (`!`) generally denote 'not'. We used this again in the final example in combination with the `is.na` function. This function indicates if an element is missing, so `!is.na` means "not missing". See Table 7.4 for a list of R's logical operators. You can use these operators and functions when subsetting data and throughout R.

TABLE 7.4: R's Logical Operators

Operator	Meaning
`<`	less than
`>`	greater than
`==`	equal to
`<=`	less than or equal to
`>=`	greater than or equal to
`!=`	not equal to
`a | b`	a or b
`a & b`	a and b
`isTRUE(a)`	determine if a is TRUE
`is.na`	missing
`!is.na`	not missing
`duplicated`	duplicated observation
`!duplicated`	not a duplicated observation

7.1.6 Recoding string/numeric variables

You may want to recode your variables. In particular, when you merge data sets you need to have **identical** identification values that R can use to match each observation. If in one data set observations for the Republic of Korea are referred to as "Korea, Rep." and in another they are labeled "South Korea", R will not know to merge them. We need to recode values in the variables that we want to match our data sets on. For example, in *fert_long_sub* the southern Korean country is labeled "Korea, Rep.". To recode it to "South Korea", type:

```
# Recode country == "Korea, Rep." to "South Korea"
fert_long_sub$country[fert_long_sub$country ==
                          "Korea, Rep."] <- "South Korea"
```

This code assigns "South Korea" to all values of the **country** variable that equal "Korea, Rep.".[2] You can use a similar technique to recode numeric variables as well. The only difference is that you omit the quotation marks. We will look at how to code factor variables later.

[2]The *countrycode* package (Arel-Bundock, 2018) is very helpful for creating standardized country identification variables.

7.1.7 Creating new variables from old

As part of your data cleanup process (or later during statistical analysis), you may want to create new variables based on existing variables. For example, we could create a new variable that is the natural logarithm of **fert_cons**. To do this, we run the variable through the `log()` function and assign a new variable that we'll call **fert_cons_log**.

```
fert_long_sub$fert_cons_log <- log(fert_long_sub$fert_cons)

summary(fert_long_sub$fert_cons_log)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   -5.12    3.27    4.63    4.15    5.14    6.80
```

Imagine that when we summarized the new log transformed variable that we had a minimum (and mean) value of `-Inf`. This would indicate that by logging the variable we have created observations with the value negative infinity. R calculates the natural logarithm of zero as negative infinity.[3] We probably don't want negative infinity values. There are a few ways to deal with this. We could drop all observations of **fert_cons** with the value zero before log transforming it. Another common solution is recoding zeros as some small nonnegative number like 0.001. For example:

```
# Recode zeros in Fertilizer Consumption
fert_long_sub$fert_cons[fert_long_sub$fert_cons ==
                                0] <- 0.001

# Natural log transform Fertilizer Consumption
fert_long_sub$fert_cons_log <- log(fert_long_sub$fert_cons)
```

Note that this example is included to demonstrate R syntax rather than to prescribe a certain transformation of skewed data with zeros. The choice of which transformation to make should ultimately be made based on the data, model, and context. See Hyndman (2010) for more information on various alternatives including Box-Cox (Box and Cox, 1964) and inverse hyperbolic sine transformations (Burbidge and Robb, 1988).

Creating factor variables

We can create factor variables from numeric or string variables. For example, we may want to turn the continuous numeric **fert_cons** variable into an

[3] R denotes positive infinity with `Inf`.

TABLE 7.5: Example Factor Levels

Number	Label	Value of **FertilizerConsumption**
1	low	< 18
2	medium low	≥ 18 and < 81
3	medium high	≥ 81 and < 158
4	high	≥ 158

ordered categorical (i.e. factor) variable. Imagine that we want to create a factor variable called **fert_cons_group** with four levels called 'low', 'medium low', 'medium high', and 'high'. To do this, let's first create a new numeric variable based on the values listed in Table 7.5. Now let's use a procedure that is similar to the variable recoding we did earlier:[4]

```
# Create numeric factor levels variable

# Attach fert_long_sub data frame
attach(fert_long_sub)

# Created new fert_cons_group variable based on # fert_cons
fert_long_sub$fert_cons_group[fert_cons < 18] <- 1
fert_long_sub$fert_cons_group[fert_cons >= 18 &
                                  fert_cons < 81] <- 2
fert_long_sub$fert_cons_group[fert_cons >= 81 &
                                  fert_cons < 158] <- 3
fert_long_sub$fert_cons_group[fert_cons >= 158] <- 4
fert_long_sub$fert_cons_group[is.na(fert_cons)] <- NA

# Detach data frame
detach(fert_long_sub)

summary(fert_long_sub$fert_cons_group)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    2.00    3.00    2.66    4.00    4.00
```

You'll notice that we don't have a factor variable yet; our new variable is numeric. We can use the `factor()` function to convert *fert_cons_group* into a factor variable with the labels we want.

[4] In this code, I attached the data frame *fert_long_sub* so that it is easier to read.

```
# Create vector of factor level labels
fc_labels <- c("low", "medium low", "medium high", "high")

# Convert fert_cons_group to a factor
fert_long_sub$fert_cons_group <-
                   factor(fert_long_sub$fert_cons_group,
                          labels = fc_labels)

summary(fert_long_sub$fert_cons_group)
```

```
##         low  medium low medium high        high
##         281         310         392         398
```

We first created a character vector with the factor-level labels and then applied using `factor`'s `labels` argument. Using `summary()` with a factor variable gives us its level labels as well as the number of observations per level.

The `cut()` function provides a less code-intensive way of creating factors from numeric ones and labeling factor levels. For example:

```
# Create a factor variable with the cut function
fert_factor <- cut(fert_long_sub$fert_cons,
                   breaks = c(-0.01, 17.99, 80.99,
                              157.99, 999.99),
                   labels = fc_labels)

summary(fert_factor)
```

```
##         low  medium low medium high        high
##         281         310         392         398
```

The `labels` argument lets us specify the factor levels' names. The `breaks` argument lets us specify what values separate the factor levels. Note that we set the first break as `-0.01`, not because any country had negative fertilizer consumption, but because the intervals created by `break()` exclude the left value and include the right value.[5] If we had used `0`, then all of the observations where a country used effectively no fertilizer would be excluded from the "low" category.

[5] In mathematical notation, the "low" level includes all values in the interval $(-0.01, 17.99]$.

7.1.8 Changing variable types

Sometimes a variable will have the wrong type. For example, a numeric variable may be incorrectly made a character string when a data set is imported from Excel. You can change variable types with a number of functions. We already saw how to convert a numeric variable to a factor variable with the `factor()` function. Unsurprisingly, to convert a variable to a character, use `character()` and `numeric()` to convert it to a numeric type variable. We can place `as.` before these functions (e.g. `as.factor()`) as a way of coercing a change in type.

Warning: Though these functions have straightforward names, a word of caution is necessary. Always try to understand why a variable is not of the type you would expect. Often variables have unexpected types because they are coded (or miscoded) in a way that you didn't anticipate. Changing the variable types, especially when using `as.`, can introduce new errors. Make sure that the conversion made the changes you expected.

7.2 Merging Data Sets

In the previous section, we learned crucial skills for cleaning up data sets. When your data sets are (a) in the same format and (b) have variables with identically matching ID values, you can merge your data sets. In this section, we'll look at two different ways to merge data sets: binding and the `merge()` function. We'll also look at ways to address a common issue when merging data: duplicated observations and columns.

7.2.1 Binding

As we saw in Chapter 3, if your data sets are in the same order—rows in all of the data sets represent the same observation of the same subject—then you can use the `cbind()` function to bind columns from the data sets together. This situation is unusual when merging real-world data. If your data sets are not in exactly the same order you will create a data set with nonsensical rows that combine data from multiple observations. Therefore, you should avoid using `cbind()` for merging most real-world data.

If you have data sets with the exact same columns and variable types and you just want to attach one under the other, you can use the `rbind()` function. It

binds the rows in one object to the rows in another.[6] It has the same syntax as `cbind()`. Again, you should be cautious when using this function, though it is more difficult to accidentally create a nonsensical data set with `rbind()`. R will give you an error if it cannot match your objects' columns.

7.2.2 Merging data frames

Generally, the `merge()` function is the safest and most effective way to merge two data sets. Imagine that we want to merge our *fert_long_sub* data frame with two other data frames we created in Chapter 6: *fin_regulator* and *disprop_data*. The simplest way to do this is to use the merge function twice, i.e.:

```
# Merge fin_regulator and disprop_data
merged_data_1 <- merge(x = fin_regulator, y = disprop_data,
                       by = "iso2c", all = TRUE)

# Merge combined data set with and fert_long_sub
merged_data_1 <- merge(x = merged_data_1, y = fert_long_sub,
                         by = "iso2c", all = TRUE)

names(merged_data_1)
```

```
##  [1] "iso2c"              "idn"
##  [3] "country.x"          "year.x"
##  [5] "reg_4state"         "country.y"
##  [7] "year.y"             "disproportionality"
##  [9] "country"            "year"
## [11] "fert_cons"          "fert_cons_log"
## [13] "fert_cons_group"
```

Let's go through this code. The `x` and `y` arguments specify which data frames we want to merge. The `by` argument specifies what variable(s) in the two frames identify the observations so that we can match them. In this example, we are merging by countries' ISO country two-letter codes.[7] We set the argument `all = TRUE` so that we keep all of the observations from both of the data frames. If the argument is set to `FALSE`, only observations that are common to both data frames will be included in the merged data frame. The others will not be included.

[6] Some programming languages and statistical programs refer to this type of action as "appending" one data set to another.

[7] Please see this chapter's Appendix for details on how I created an ISO country two-letter code variable in the *fin_regulator* data frame.

You might have noticed that this isn't actually the merge that we want to accomplish with these data frames. Remember that observations are not identified in this time-series cross-section data by one country name or other country code variable. Instead, they are identified by both country and year variables. To merge data frames based on the overlap of two variables (e.g. match Afghanistan-2010 in one data frame with Afghanistan-2010 in the other), we need to add the `union()` function to `merge`'s `by` argument. Here is a full example:

```
# Merge fin_regulator and disprop_data
merged_data_2 <- merge(fin_regulator, disprop_data,
                       union("iso2c", "year"),
                       all = TRUE)

# Merge combined data frame with fert_long_sub
merged_data_2 <- merge(merged_data_2, fert_long_sub,
                       union("iso2c", "year"),
                       all = TRUE)

names(merged_data_2)
```

```
##  [1] "iso2c"              "year"
##  [3] "idn"                "country.x"
##  [5] "reg_4state"         "country.y"
##  [7] "disproportionality" "country"
##  [9] "fert_cons"          "fert_cons_log"
## [11] "fert_cons_group"
```

After merging data frames, it is always a good idea to look at the result and make sure it is what you expected. Some post-merging cleanup may be required to get the data frame ready for statistical analysis.

Bigger data

Before discussing post-merge cleanup, it is important to highlight ways to handle large data sets. The `merge()` function and many of the other data frame manipulation functions covered so far in this chapter may not perform well with very large data sets. If you are using very large data sets, it might be worth investing time learning how to use packages like *dbplyr* (Wickham and Ruiz, 2019) and *data.table* packages (Dowle and Srinivasan, 2019). They have many capabilities for working efficiently with large data sets. Likely, if you have very large data, you will need to learn SQL (Structured Query

Language) or another special purpose data handling language.[8] Once you know how these languages work, you can incorporate them into your R workflow with R packages like *dbplyr*.

Duplicate values

Duplicate observations are one thing to look out for after (and before) merging. You can use the `duplicated()` function to check for duplicates. Use the function in conjunction with subscripts to remove duplicate observations. For example, let's create a new object called *data_duplicates* from the iso2c-years that are duplicated in *merged_data_2*. Remember that **iso2c** and **year** are in the first and second columns of the data frame.

```
# Created a data frame of duplicated country-years
data_duplicates <- merged_data_2[
                      duplicated(merged_data_2[, 1:2]), ]

# Show the number of rows in data_duplicates
nrow(data_duplicates)
```

```
## [1] 6
```

In this data frame, there are duplicated iso2c-year observations. We know this because `nrow` tells us that the data frame with the duplicated values has rows, i.e. observations.

To create a data set without duplicated observations (if there are duplicates), add an exclamation point (`!`) before `duplicated`, i.e. not duplicated, in the above code.

```
# Created a data frame of unique country-years
data_not_duplicates <- merged_data_2[
                          !duplicated(merged_data_2[, 1:2]), ]
```

Note that if you do have duplicated values in your data set and you run a similar procedure on it, it will drop duplicated values that have a lower order in the data frame. To keep the lowest ordered value and drop duplicates higher in the data set, use `duplicated`'s `fromLast` argument like this: `fromLast = TRUE`.

Warning: Look over your data set and the source code that created the data set to try to understand why duplicates occurred. There may be a fundamental

[8] w3schools has an online SQL tutorial at: `http://www.w3schools.com/sql/default.asp`.

problem in the way you are handling your data that resulted in the duplicated observations.

7.2.3 Duplicate columns

Another common post-merge cleanup issue is duplicate columns, i.e. variables. These are variables from the two data frames with the same name that were not included in `merge`'s `by` argument. For example, in our previous merged data examples, there are three country name variables: **country.x**, **country.y**, and **country** to signify which data frame they are from.[9]

You should decide what to do with these variables on a case-by-case basis. But if you decide to drop one of the variables and rename the other, you can use subscripts (as we saw in Chapter 3). The *dplyr* package has a useful function called `select()` that can also remove variables from data frames. To remove variables, write a minus sign (`-`) and then the variable name without quotes. For example, imagine that we want to keep **country.x** and drop the other variables.[10] Let's also remove the **idn** variable:

```
# Remove country.y, country, X, and idn
final_cleaned <- dplyr::select(data_not_duplicates, -country.y,
                               -country, -idn)

# Rename country.x = country
final_cleaned <- dplyr::rename(final_cleaned,
                               country = country.x)
```

```
names(final_cleaned)
```

```
## [1] "iso2c"               "year"
## [3] "country"             "reg_4state"
## [5] "disproportionality" "fert_cons"
## [7] "fert_cons_log"       "fert_cons_group"
```

Alternatively, you can select specific variables to keep with the `select` function by writing the variables' names without a minus sign. **Note**: If you are merging many data sets, it can sometimes be good to clean up duplicate columns between each `merge()` call.

[9] The former two were created in the first merge between *fin_regulator* and *disprop_data*. When the second merge was completed, there were no variables named **country** in the MergeData2 data frame, so **country** did not need to be renamed in the new merged data set.

[10] This version of the country variable is the most complete.

Chapter summary

This chapter has provided you with many tools for cleaning up your data to get it ready for statistical analysis. Before moving on to the next chapter to learn how to incorporate statistical analysis as part of a reproducible workflow with knitr/R Markdown, it's important to reiterate that the function we've covered in this chapter should usually be embedded in the types of data creation files we saw in Chapter 6. These files can then be tied together with a makefile into a process that should be able to relatively easily take very raw data and clean it up for use in your analyses. Embedding these functions in data creation source code files, rather than just typing the functions into your R console or manually changing data in Excel, will make your research much more reproducible. It will also make it easier to backtrack and find mistakes that you may have made while transforming the data. Including new or updated data when it becomes available will also be much easier if you use a series of segmented data creation source code files that are tied together with a makefile.

Appendix

R code for turning *fert_cons_data* into year-wide-format:

```
library(WDI)
library(tidyr)
library(dplyr)

# Gather fertilizer consumption data from WDI
fert_cons_data <- WDI(indicator = "AG.CON.FERT.ZS")

# Reshape fert_cons_data to year wide-format
fert_wide <- tidyr::pivot_wider(fert_cons_data,
                                names_from = year,
                                values_from = AG.CON.FERT.ZS)

# Order fert_wide by country
fert_wide <- arrange(fert_wide, country)
```

R code for creating iso2c country codes with the *countrycode* package:

```
library(countrycode)

fin_regulator$iso2c <- countrycode(fin_regulator$country,
                                   origin = "country.name",
                                   destination = "iso2c")
```

R code for creating the chapter's density plot:

```
library(ggplot2)

# Set plot theme to "minimal"
theme_set(theme_minimal())

# Create density plot
ggplot(data = fert_long, aes(fert_cons)) +
      geom_density() +
```

```
      xlab("Fertilizer Consumption") + ylab("Density") +
      theme_bw()
```

Part III

Analysis and Results

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

8

Statistical Modeling and knitr/R Markdown

When you have your data cleaned and organized, you will begin to examine it with statistical analyses. In this book we don't look at how to do statistical analysis in R (a subject that would and does take up many other books). Instead, we focus on how to make your analyses really reproducible. You do this by dynamically connecting your data gathering and analysis source code to your presentation documents. When you dynamically connect your data gathering makefiles and analysis source code to your markup document, you will be able to completely rerun your data gathering and analysis and present the results whenever you compile the presentation documents. This makes it very clear how you found the results that you are advertising. It also automatically keeps the presentation of your results, including tables and figures, up-to-date with any changes you make to your data and analyses source code files.

You can dynamically tie your data gathering, statistical analyses, and presentation documents together with knitr/R Markdown. In Chapter 3 you learned basic *knitr/rmarkdown* package syntax. For the rest of the chapter, I'll refer to it as "*knitr* syntax", but it applies to R Markdown as well when it is not specific to LaTeX. In this chapter we will begin to learn knitr syntax in more detail, particularly code chunk options for including dynamic code in your presentation documents. This includes code that is run in the background, i.e. not shown in the presentation document, as well as displaying the code and output in your presentation document both as separate blocks and inline with the text. We will also learn how to dynamically include code from languages other than R. We examine how to use knitr with modular source code files. Finally, we will look at how to create reproducible random analyses and how to work with computationally intensive code chunks.

The goal of this and the next two chapters, which cover dynamically presenting results in tables and figures, is to show you how to tie data gathering and analyses into your presentation documents so closely that every time the documents are compiled they actually reproduce your analysis and present the results. Please see the next part of this book, Part IV, for details on how to create the LaTeX and Markdown documents that can include *knitr* code chunks.

Reminder: Before discussing the details of how to incorporate your analysis

into your source code, it's important to reiterate something we discussed in Chapter 2. The syntax and capabilities of R packages and R itself can change with new versions. Also, as we have seen for file path names, syntax can change depending on what operating system you are using. So it's important to have your R session info available (see Section 2.2.1 for details) to make your research more reproducible and future-proof. If someone reproducing your research has this information, they will be able to download your files and use the exact version of the software that you used. For example, CRAN maintains an archive of previous R package versions that can be downloaded.[1] Previous versions of R itself can also be downloaded through CRAN.[2]

8.1 Incorporating Analyses into the Markup

For a relatively short piece of code that you don't need to run in multiple presentation documents, it may be simplest to type the code directly into chunks written in your *knitr* markup document. In this section we will learn how to set *knitr* options for handling these code chunks. For a list of many of the chunk options covered here, see Table 3.1.

8.1.1 Full code chunks

By default, *knitr* code chunks are run by R, and the code and any text output (including warnings and error messages) are inserted into the text of your presentation documents in blocks. The blocks are positioned in the final presentation document text at the points where the code chunk was written in the knittable markup. Figures are inserted as well. Let's look at the main options for determining how code chunks are handled by *knitr*.

`include`

Use `include=FALSE` if you don't want to include anything in the text of your presentation document, but you still want to evaluate a code chunk. It is `TRUE` by default.

[1] See: `http://cran.r-project.org/src/contrib/Archive/`.
[2] See: `http://cran.r-project.org/src/base/`.

`eval`

The `eval` option determines whether or not the code in a chunk will be run. Set the `eval` option to `FALSE` if you would like to include code in the presentation document text without actually running the code. By default it is set to `TRUE`, i.e. the code is run. You can alternatively use a numerical vector with `eval`. The numbers in the vector tell *knitr* which expressions in the chunk to evaluate. For example, if you only want to evaluate the first two expressions, set `eval=1:2`.

`echo`

If you would like to hide a chunk's code from the presentation document, you can set `echo=FALSE`. Note that if you also have `eval=TRUE`, then the chunk will still be evaluated and the output will be included in your presentation document. Clearly, if `echo=TRUE`, then source code will be included in the presentation document. As with `eval`, you can alternatively use a numerical vector in `echo`. The numbers in the vector indicate which expressions to echo in your final document.

`results`

We will look at the `results` option in more detail in the next two chapters (see especially Section 9.1). However, let's briefly discuss the option value `hide`. Setting `results='hide'` is almost the opposite of `echo=FALSE`. Instead of showing the results of the code chunk and hiding the code, `results='hide'` shows the code, but not the results. Warnings, errors, and messages will still be printed.

`warning`, `message`, `error`

If you don't want to include the warnings, messages, and error messages that R outputs in the text of your presentation documents, just set the `warning`, `message`, and `error` options to `FALSE`. They are set to `TRUE` by default.

`cache`

If you want to run a code chunk once and save the output for when you knit the document again, rather than running the code chunk every time, set the option `cache=TRUE`. When you do this the first time the document is knitted, the chunk will be run and the output stored in a sub-directory of the working directory called *cache*. When the document is subsequently knitted, the chunk will only be run if the code in the chunk changes or its options change. This is

very handy if you have a code chunk that is computationally intensive to run. The `cache` option is set to `FALSE` by default. Later in this chapter (Section 8.4), we will see how to use the `cache.vars` function to cache only certain variables created by a code chunk.

`dependson`

Cached chunks are only rerun when their code changes. Sometimes one chunk will depend on the results from a prior chunk. In these cases, it is good to rerun the chunk if the prior chunk one is also rerun. The `dependson` option allows you to do this automatically. You can specify either a vector of the labels for the chunks depended on or their numbers in order from the start of the document. For example, `dependson=c(2, 3)` specifies that if the second or third chunks are rerun, then the current chunk will also be rerun.

`cache.extra`

Sometimes to ensure reproducibility, it may be useful to rerun a chunk when some other condition changes, such as when a new version of R is installed or a dependent file changes. You can feed a list of conditions to `cache.extra` to do this. For instance:

```
cache.extra=list(file.info(data.csv)$mtime, R.version)
```

Here we set two conditions under which the chunk will be rerun. The first specifies that the chunk should be rerun whenever the *data.csv* file is modified. The `file.info` function extracts information about the file and `mtime` gives the last time that the file was modified. If this differs from when the chunk was last run, then it will be run again. This is very useful for keeping your cached chunks and the files they rely on in sync.

The second condition enabled by `R.version` reruns the chunk whenever the R version or even the operating system changes. If you only want to rerun the chunk when the version of R is different, then use `R.version.string`.

`size`

If you do want to print part or all of your code chunk into a LaTeX document, you may also want to resize the text. To do this, use the `size` option. By default, it is set to `size='normalsize'`. You can use any of the LaTeX font sizes listed in Chapter 11.

8.1.2 Showing code and results inline

Sometimes you may want to have R code or output show up inline with the rest of your presentation document's text. For example, you may want to include a small chunk of stylized code in your text when you discuss how you did an analysis. Or you may want to dynamically report the mean of some variable in your text so that the text will change when you change the data. The *knitr* syntax for including inline code is different for the LaTeX and Markdown languages. We'll cover both in turn.

LaTeX

Inline static code

There are a number of ways to include a code snippet inline with your text in LaTeX. You can use the LaTeX function `\texttt` to have text show up in the `typewriter` font commonly used in LaTeX-produced documents to indicate that some text is code (I use typewriter font for this purpose in this book, as you have probably noticed). For example, using `\texttt{2 + 2}` will give you `2 + 2` in your text. Note that in LaTeX curly brackets (`{}`) work exactly like parentheses in R, i.e. they enclose a function's arguments.

However, the `\texttt` function isn't always ideal, because your LaTeX compiler will still try to run the code inside of the function as if it were LaTeX markup. This can be problematic if you include characters like the backslash `\` or curly brackets `{}`. They have special meanings for LaTeX. The hard way to solve this problem is to use escape characters (see Chapter 4). The backslash is an escape character in LaTeX.

Probably the better option is to use the `\verb` function. It is equivalent to the `eval=FALSE` option for full *knitr* code chunks. To use the `\verb` function, pick some character you will not use in the inline code. For example, you could use the vertical bar (`|`). This will be the `\verb` delimiter. Imagine that we want to actually include `\texttt` in the text. We would type:

```
\verb|\texttt|
```

The LaTeX compiler will ignore almost anything from the first vertical bar up until the second bar following `\verb`. All of the text in-between the delimiter characters is put in typewriter font.[3]

[3] For more details, see the LaTeX Wikibooks page: `https://en.wikibooks.org/wiki/LaTeX/Paragraph_Formatting#Verbatim_text` (accessed 21 September 2019). Also, for help troubleshooting, see the UK List of Frequently Asked Questions: `https://texfaq.org/FAQ-verbwithin` (accessed 21 September 2019).

Inline dynamic code

If you want to dynamically show the results of some R code in your *knitr* LaTeX-produced text you can use `\Sexpr`. This is a pseudo-LaTeX function; it looks like LaTeX, but it is actually *knitr* syntax.[4] Its structure is more like a LaTeX function's structure than *knitr*'s in that you enclose your R code in curly brackets (`{}`) rather than the `<<>>= . . . @` syntax you use for block code chunks.

For example, imagine that you wanted to include the mean of a vector of river lengths, 591, in the text of your document. The *rivers* numeric vector, loaded by default in R, has the lengths of 141 major rivers recorded in miles. You can use the `mean()` function to find the mean and the `round()` function to round the result to the nearest whole number:

```
round(mean(rivers), digits = 0)
```

```
## [1] 591
```

To have just the output show up inline with the text of your document, you would type something like:

```
The mean length of 141 major rivers in North America is
\Sexpr{round(mean(rivers), digits = 0)} miles.
```

R code included inline with `Sexpr` is evaluated using current R options. So if you want all of the output from `Sexpr` to be rounded to the same number of digits, for example, it might be a good idea to set this in a code chunk with R's `options()` function.

Markdown

Inline static code

To include static code inline in an R Markdown (and regular Markdown) document, enclose the code in single backticks (`` ` . . . ` ``). For example:

```
This is example R code: `MeanRiver <- mean(rivers)`.
```

produces:[5]

[4] The function directly descends from *Sweave*.

[5] The exact look of the text depends on the Cascading Style Sheets (CSS) style file you are using. The example here was created with RStudio's default style file.

This is example R code: `MeanRiver <- mean(rivers)`.

Inline dynamic code

Including dynamic code in the body of your R Markdown text is similar to including static code. The only difference is that you put the letter `r` after the first single backtick.

8.1.3 Dynamically including non-R code in code chunks

You are not limited to dynamically including just R code in your presentation documents. *knitr* can run code from a variety of other languages including: Python, Ruby, Bash, Julia, and Stan. All you have to do to dynamically include code from one of these languages is use the `engine` code chunk option to tell *knitr* which language you are using. For example, to dynamically include a simple line of Python code in an R Markdown document type:

````
```{r engine='python'}
print "Reproducible Research"
```
````

In the final HTML file you will get output that looks like Figure 8.1.[6]

```
print "Reproducible Research"
```

```
## Reproducible Research
```

FIGURE 8.1: Output from Python Engine in HTML Markdown

Many of the programming language values `engine` can take are listed in Table 8.1.

8.2 Dynamically Including Modular Analysis Files

There are a number of reasons why you might want to have your R source code located in separate files from your markup documents even if you compile them together with *knitr*.

[6] Again, this was created using RStudio's default CSS style file.

TABLE 8.1: A Selection of *knitr* `engine` Values

Value	Programming Language
`awk`	Awk
`bash`	Bash shell
`gawk`	Gawk
`haskell`	Haskell
`julia`	Julia
`python`	Python
`R`	R (default)
`ruby`	Ruby
`sas`	SAS
`sh`	Bourne shell
`stan`	Stan probablistic programming language

First, it can be unwieldy to edit both your markup and long R source code chunks in the same document, even with RStudio's handy *knitr* code folding and chunk management options. There are just too many things going on in one document.

Second, you may want to use the same code in multiple documents, for example an article and slide show presentation. It is nice to not have to copy and paste the same code into multiple places. Instead, it is easier to have multiple documents link to the same source code file. When you make changes to this source code file, the changes will automatically be made across all of your presentation documents. You don't need to make the same changes multiple times.

Third, other researchers trying to replicate your work might only be interested in specific parts of your analysis. If you have the analysis broken into separate and clearly labeled modular files that are explicitly tied together in the markup file with *knitr*, it is easy for them to find the specific bits of code that they are interested in.

8.2.1 Source from a local file

Usually, in the early stages of your research, you may want to run code stored in analysis files located on your computer. Doing this is simple. The *knitr* syntax is the same as for block code chunks. The only change is that instead of writing all of your code in the chunk, you save it to its own file and use the

`source()` function to access it.[7] For example, in an R Markdown file we could run the R code in a file called *main-analysis.R* from our *example-project* like this:

```
```{r, include=FALSE}
# Run main analysis
source("/example-project/analysis/main-analysis.R")
```
```

Notice that we set the option `include=FALSE`. This will run the analysis and produce objects created by the analysis code that can be used by other code chunks, but the output will not show up in the presentation document's text.

Sourcing a makefile in a code chunk

In Chapter 6 we created a GNU Makefile to organize our data gathering. You can run makefiles every time you compile your presentation document. This can keep your data, analyses, figures, and tables up-to-date. One way to do this is to run the GNU makefile in an R code chunk with the `system()` function. Perhaps a better way to run makefiles from *knitr* presentation documents is to include the functions in a code chunk using the Bash engine. For example, a Sweave-style code chunk for running the makefiles in our example project would look like this:

```
<<r engine='bash', include=FALSE>>=
# Change working directory to /example-project/analysis/Data
cd /example-project/analysis/Data/

# Run makefile
make cleanMerge all

# Change to working directory to /example-project/analysis/
cd /example-project/analysis/
@
```

Please see Chapter 6 for details on the `make` command arguments used here.

You can also use R's `source()` function to run an R make-like data gathering file. Unlike GNU Make, this will rerun all of the data gathering files, even if they have not been updated. This may become very time consuming depending on the size of your data sets and how they are manipulated.

One final note on including makefiles in your knitr presentation document

[7] We used the `source()` function in Chapter 6 in our make-like data gathering file.

source code: it is important to place the code chunk with the makefile before code chunks containing statistical analyses that depend on the data file it creates. Placing the makefile first will keep the others up-to-date.

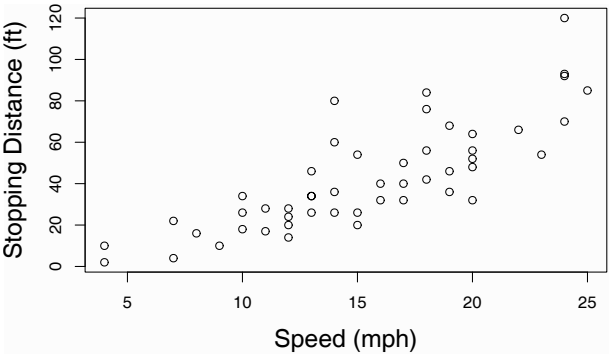
8.2.2 Source from a URL

If you are using GitHub or another service that uses secure URLs to host your analysis source code files, you need to use the `source_url()` function in the *devtools* package.[8] For GitHub-based source code, we find the file's URL the same way we did in Chapter 5. Remember to use the URL for the *raw* version of the file. I have a short script hosted on GitHub for creating a scatterplot from data in R's *cars* data set. The script's shortened URL is `http://bit.ly/1D5p1w6`.[9] To run this code and create the scatterplot using `source_url()`, type:

```
library(devtools)

# Run the source code to create the scatter plot
source_url("http://bit.ly/1D5p1w6")
```

```
## SHA-1 hash of file is ff75a88b90decfcaefc9903bbc283e1fc4cd2339
```



You can also use the *devtools* function `source_gist()` in a similar way to source GitHub Gists. Gists are a handy way to share code over the internet. For more details, see: `https://gist.github.com/`.

[8] You can also make the replication code accessible for download and either instruct others to change the working directory to the replication file or have them change the directory information as necessary. You will need to do this with GNU makefiles like those included with this book.

[9] The original URL is at `https://raw.githubusercontent.com/christophergandrud/Rep-Res-Examples/master/Graphs/SimpleScatter.R`. This is very long, so I shortened it using bitly. You may notice that the shortened URL is not secure. However, it does link to the original secure URL.

Similar to what we saw in Chapter 5 if you would like to use a particular version of a file stored on GitHub, include that version's URL in the `source_url()` call. This can be useful for replicating particular results. Linking to a particular version of a source code file will enable replication even if you later make changes to the file. To access the URL for a particular version of a file, first click on the file on GitHub's website, then click the `History` button. This will take you to a page listing all of the file's versions. Click on the `Browse Code` button next to the version of the file that you want to use. Finally, click on the `Raw` button to be taken to the text-only version of the file. Copy this page's URL and use it in `source_url()`.

8.3 Reproducibly Random: `set.seed()`

If you include simulations in your analysis it is often a good idea to specify the random number generator state you used. This will allow others to exactly replicate your 'randomly'—really pseudo-randomly—generated simulation results. Use the `set.seed()` function in your source code files or code chunks to do this. For example, use the following code to set the random number generator state[10] and randomly draw 1,000 numbers from a standard normal distribution with a mean of 0 and a standard deviation of 2.

```
# Set seed as 125
set.seed(125)

# Draw 1000 numbers
draw_1 <- rnorm(1000, mean = 0, sd = 2)

summary(draw_1)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -7.211  -1.407  -0.104  -0.122   1.316   5.677
```

The `rnorm()` function draws the 1,000 simulations. The `mean` argument allows us to set the normal distribution's mean and `sd` sets its standard deviation. Just to show you that we will draw the same numbers if we use the same seed, let's run the code again:

[10] See the `Random` help file for detailed information on R's random number generation capabilities by typing `?Random` into your console.

```
# Set seed as 125
set.seed(125)

# Draw 1000 numbers
draw_2 <- rnorm(1000, mean = 0, sd = 2)

summary(draw_2)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -7.211  -1.407  -0.104  -0.122   1.316   5.677
```

8.4 Computationally Intensive Analyses

Sometimes you may want to include computationally intensive analyses that take a long time to run as part of a knitr document. This can make writing the document frustrating because it will take a long time to knit it each time you make changes. There are at least two solutions to this problem: the `cache` chunk option and makefiles. We discussed makefiles in Chapter 6, so let's look at how to work with the `cache` option.

When you set `cache=TRUE` for the code chunk that contains the analysis, the code chunk will only be run when the chunk's contents change[11] or the chunk options change. This is a very easy solution to the problem. It does have a major drawback: other chunks can't access objects created by the chunk or use functions from packages loaded in it. Solve these problems by (a) having packages loaded in a separate chunk and (b) save objects created by the cached chunk to a separate RData file that can be loaded in later chunks (see Section 3.1.3 for information on saving to RData files).[12]

Imagine that in a cached code chunk we create an object called *Sample*. Then in a later code chunk we want to use the `hist()` function to create a histogram of the sample. In the cached code chunk, we save *Sample* to a file called *sample.RData*.

[11] Note that the chunk will not be run if only the contents of a file that the chunk sources are changed. Use the `dependson` option in cases where it is important to rerun a chunk when a prior chunk changes.

[12] It's true that when *knitr* caches a code chunk it saves the chunk's objects to an `.RData` file. However, it is difficult to load this file directly because the file name changes every time the cached chunk is rerun.

```
<<Sample, cache=TRUE>>=
Sample <- (n = 1000, mean = 5, sd = 2)

save(Sample, file = "sample.RData")
@
```

The latter code chunk for creating the histogram would go something like this:[13]

```
<<Histogram>>=
load(file = "sample.RData")

hist(Sample)
@
```

`cache.vars`

If the code chunk you want to cache creates many objects, but you only want to save a few of them, you can use *knitr*'s `cache.vars` chunk option. Simply give it a character vector of the objects' names that you want to save.

Chapter summary

In this chapter we covered in more detail key *knitr* syntax for including code chunks in our presentation documents. This and other tools we learned in this chapter are important for tying our statistical analyses directly to its advertising, i.e. our presentation documents. In the next two chapters, we will learn how to take the output from our statistical analysis and, using *knitr*, present the results with dynamically created tables and figures.

[13] For reference, *Sample* was created by using the `rnorm()` function to take a random sample of size 1,000 from a normal distribution with a mean of five and standard deviation of two.

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

9

Showing Results with Tables

Graphs and other visual methods, discussed in the next chapter, can often be more effective ways to present descriptive and inferential statistics than tables.[1] Nonetheless, tables of parameter estimates, descriptive statistics, and so on can sometimes be important tools for describing your data and presenting research findings. See Ehrenberg (1977) and Gelman (2011) for information on creating tables for effective communication.

Learning how to dynamically connect statistical results with tables in your presentation documents aids reproducibility and can ultimately save you a lot of time. Manually typing results into tables by hand is tedious, not very reproducible, and can introduce errors.[2] It's especially tedious to retype tables to reflect changes you made to your data and models. Fortunately, you don't actually need to create tables by hand. There are many ways to have R do the work for you.

The goal of this chapter is for you to learn how to dynamically create tables for your presentation documents written in LaTeX and Markdown. We will first learn the simple knitr/R Markdown syntax we need to dynamically include tables created from R objects. Then we will learn how to actually create the tables. There are a number of ways to turn R objects into tables that can be dynamically included in LaTeX or Markdown/HTML markup. In this chapter we mostly focus on three tools for creating tables: the `kable()` function from *knitr*, the *xtable* package, and the *texreg* package (Leifeld, 2017). `kable()` can create tables from data frames for both LaTeX and Markdown/HTML documents. *xtable* does the same, but is much more customizable. *texreg* produces publication-quality tables from objects containing statistical model results, or model objects. It allows you to combine results from multiple models into one table. Unfortunately *texreg* is less flexible with objects of classes it does not support.[3]

[1] This is especially true of the small-print, high-density coefficient estimate tables that are sometimes descriptively called 'train schedule' tables.

[2] For example, in a replication of Reinhart and Rogoff's (2010) much cited study of economic growth and public debt, Herndon et al. (2014) found a number of apparent transcription errors. Analysis results in the original spreadsheets appear to not have been entered into the paper's tables accurately.

[3] These are not the only packages available in R for creating presentation document tables from R objects. I personally really like the *stargazer* package (Hlavac, 2018). It has a similar

Warning: Automating table creation removes the possibility of adding errors to the presentation of your analyses by incorrectly copying output, a big potential problem in hand-created tables. However, it is not error-free. You could easily create inaccurate tables with coding errors. So, as always, it is important to 'eyeball' the output. Does it make sense? If you select a couple values in the R output, do they match what is in the presentation document's table? If not, you need to go back to the code and see where things have gone wrong. With that caveat, let's start making tables.

9.1 Basic *knitr* Syntax for Tables

The most important *knitr*/*rmarkdown* chunk option for showing tables is `results`. The `results` option can have one of four values:

- `'hide'`,
- `'asis'`,
- `'markup'`,
- `'hold'`.

The value `hide` clearly hides the results of your code chunk from your presentation document. `hold` collects all of the output and prints it at the end of the chunk. To include tables created from R objects in your LaTeX or Markdown output you should set `results='asis'` or `results='markup'`. `asis` is the simplest option as it writes the raw markup form of the table into the presentation document, not as a highlighted code chunk, but as markup. It is then compiled as table markup with the rest of the document. `markup` uses an output hook to mark up the results in a predefined way. In this chapter, we will work with examples using the `asis` option.

9.2 Table Basics

Before getting into the details of how to create tables from R objects, it is useful to first learn how generic tables are created in LaTeX and Markdown/HTML. If you are not familiar with basic LaTeX or Markdown syntax, you might want

syntax to *texreg* and is particularly good for showing results from multiple models estimated using different model types in one table.

to skip ahead to Chapters 11 and 12, respectively, before coming back to learn about making tables in these languages.

9.2.1 Tables in LaTeX

Tables in LaTeX are usually embedded in two environments: the `table` and `tabular` environments. What is a LaTeX environment in general?

A LaTeX environment is a part of the markup where special commands are executed. A simple environment is the `center` environment.[4] Everything placed in a center environment is, unsurprisingly, centered. Typing:

```
\begin{center}
    This is a center environment.
\end{center}
```

creates the following text in the PDF output:

This is a center environment.

LaTeX environments all follow the same general syntax:

```
\begin{ENVIRONMENT_NAME}
    ...
    ...
\end{ENVIRONMENT_NAME}
```

You do not have to indent the contents of an environment. Indentations neither affect how the document is compiled nor show up in the final PDF.[5] It is conventional to indent them, however, because it makes the markup easier to read.

In this chapter we will learn about two types of environments you need for tables in LaTeX. The `tabular` environment allows you to format the content of a table. The `table` environment allows you to format a table's location in the text and its caption.

[4] For a comprehensive list of LaTeX environments, see `https://latex.wikia.org/wiki/List_of_LaTeX_environments`.

[5] An aside: the `tabbing` environment is a useful way to create tabbed text in LaTeX. We don't cover this here though.

The tabular environment

The tabular environment allows you to create tables in LaTeX. Let's work through the basic syntax for a simple table.[6]

To begin a simple tabular environment type \begin{tabular}{TABLE_SPEC}. The TABLE_SPEC argument allows you to specify the number of columns in a table and the alignment of text in each column. For example, to create a table with three columns, the first of which is left-justified and the latter two center-justified we type:

```
\begin{tabular}{l c c}
```

The l argument creates a left-justified column, c creates a centered one. If we wanted a right-justified column we would use r.[7] Finally, we can add a horizontal line between columns by adding a vertical bar | between the column arguments.[8] For example, to place a vertical line between the first and second columns in our example table, we would type:

```
\begin{tabular}{l | c c}
```

Now let's enter content into our table. We saw earlier how CSV files delimit individual columns with commas. In LaTeX's tabular environment, columns are delimited with ampersands (&).[9] In CSV tables, new lines are delimited by starting a new line. In LaTeX tables you use two backslashes (\\).[10] Here is a simple example of the first two lines of a table:

```
\begin{tabular}{l | c c}
    Observation & Variable1 & Variable2 \\
    Subject1 & a & b \\
```

[6] For a comprehensive overview, see the LaTeX Wiki page on tables: https://en.wikibooks.org/wiki/LaTeX/Tables.

[7] You can also specify a column's width by using m{WIDTH} instead. Be sure to load the *array* package in the preamble for this to work. Using m will create a column of a specified width that is vertically justified in the middle. For example, m{3cm} would create a column with a width of 3 centimeters. Text in the column would automatically be wrapped onto multiple lines if need be. You can replace the m with either p or b. p vertically aligns the text at the top, b aligns it at the bottom.

[8] If you add two vertical bars (||), you will get two lines.

[9] If you want to include an ampersand in the text of your LaTeX document, you need to escape it like this: \&.

[10] You can use two backslashes outside of the tabular environment as well to force a new line. Also, to increase the space between the line, you can add a vertical width argument to the double backslashes. For example, \[3cm] will give you a 3-centimeter gap between the current line and the next one.

It is common to demarcate the row with a table's column names, the first row, with horizontal lines. A horizontal line also often visually demarcates a table's end. You can add horizontal lines in the `tabular` environment with the `\hline` command.

```
\begin{tabular}{l | c c}
    \hline
    Observation & Variable1 & Variable2 \\
    \hline \hline
    Subject1 & a & b \\
    \hline
```

Finally, we close the `tabular` environment with `\end{tabular}`. The full code (with a few extra rows added) is:

```
\begin{tabular}{l | c c}
    \hline
    Observation & Variable1 & Variable2 \\
    \hline \hline
    Subject1 & a & b \\
    Subject2 & c & d \\
    Subject3 & e & f \\
    Subject4 & g & h \\
    \hline
\end{tabular}
```

This produces the following table:

Observation	Variable1	Variable2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	h

The `table` float environment

You might notice that the table we created so far lacks a title and is bunched very closely to the surrounding text. In LaTeX we can create a `table` float environment to solve this problem. Float environments allow us to separate a table from the text, specify its location, and give it a caption.[11] To begin a `table` float environment, use `\begin{table}[POSITION_SPEC]`. The argument allows us to determine the location of the table. It can be set to `h` for here,

[11] We will see in the next chapter how to use `figure` floats as well.

TABLE 9.1: Example Simple LaTeX Table

Observation	Variable1	Variable2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	h

i.e. where the table is written in the text. It can also be `t` to place it on the top of a page or `b` for the bottom of the page. To set a title for the table, use the `\caption` command. LaTeX automatically determines the table's number, so you only need to enter the text. You can also declare a cross-reference key for the table with the `\label` command.[12] A `table` environment is closed with `\end{table}`. Let's see a full example.

```
\begin{table}[t]
    \caption{Example Simple LaTeX Table}
    \label{ExLaTeXTable}
    \begin{center}
        \begin{tabular}{l | c c}
            \hline
            Observation & Variable1 & Variable2 \\
            \hline \hline
            Subject1 & a & b \\
            Subject2 & c & d \\
            Subject3 & e & f \\
            Subject4 & g & h \\
            \hline
        \end{tabular}
    \end{center}
\end{table}
```

Notice that the `tabular` environment is further nested in the `center` environment. This centers the table, while leaving the table's title left-justified. The final result is Table 9.1. One final tip: to have the caption placed at the bottom rather than the top of the table in the final document, simply put the `caption` command after the `tabular` environment is closed.

You can see how typing out a table in LaTeX gets very tedious very fast. For all but the simplest tables, it is best to try to have R do the table-making work for you.

[12] This command works throughout LaTeX. To reference the table type in the text of your document `\ref{KEY}`, where `KEY` is what you set with the `\label` command. Use `\pageref` to reference the page number.

9.2.2 Tables in Markdown/HTML

Now we will briefly look at the syntax for creating simple Markdown and HTML tables before turning to learn how to have R create these tables for us.

Markdown tables

Markdown table syntax, as with all Markdown syntax, is generally much simpler than LaTeX's tabular syntax. The markup is much more human readable. Nonetheless, larger tables can still be tedious to create.

You do not need to declare any new environments to start creating a Markdown table. Just start typing in the content. Columns are delimited in Markdown tables with a vertical bar (`|`). Rows are started with a new line. To indicate the head of the table, usually the row(s) containing the column names, separate it from the body of the table with a row of dashes (e.g. `-----`). Here is an example based on the table we created in the previous section:

```
Observation | Variable1  | Variable2
----------- | ---------- | ---------
Subject1    | a          | b
```

Note that it is not necessary to line up the vertical bars. You just need to have the same number of them on each row.

You can specify each column's text justification using colons on the dashed row. For example, this code will create the left-center-center justified formatted table we made earlier:

```
Observation | Variable1 | Variable2
:---------- | :-------: | :-------:
Subject1    | a         | b
Subject2    | c         | d
Subject3    | e         | f
Subject4    | g         | c
```

To create a left-justified column, use a colon on only the left side of the dashes.

The ultimate look of a Markdown table is highly dependent on the CSS style file you are using (see Chapter 12 for how to change your CSS style file). The default RStudio CSS style as of late 2019 formats our table to look like this:

Observation	Variable1	Variable2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	c

Using a different CSS style file,[13] we can get something like this:

OBSERVATION	VARIABLE1	VARIABLE2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	c

In basic Markdown, you can add a caption with the heading syntax (see Section 12.1.3). In this example the three hashes (`###`) create the header:

```
### Example Simple Markdown Table
Observation | Variable1 | Variable2
:---------- | :-------: | :-------:
Subject1    | a         | b
```

producing something like this:

[13] The table was created using the Upstanding Citizen style from the program Marked.

Example Simple Markdown Table

OBSERVATION	VARIABLE1	VARIABLE2
Subject1	a	b
Subject2	c	d
Subject3	e	f
Subject4	g	c

HTML tables

The `texreg()` function that we will learn in the next section doesn't create tables formatted with Markdown syntax. It can create tables with HTML syntax. This is useful for us because virtually any HTML markup can be incorporated into a Markdown document. In fact, Markdown table syntax is only a stepping stone for more easily producing tables with HTML syntax. So it is useful to also understand the basic syntax for HTML tables.

HTML uses element "tags" to begin and end tables. The main element we use to create tables is, well, the `tables` element. This is very similar to LaTeX's `tabular` environment. An HTML element generally begins with a start tag and ends with an end tag. This is similar to LaTeX's `\begin{}` and `\end{}` commands. Begin tags are encapsulated in a greater than and less than sign and include the element tag name (`<TAG>`). End tags are similar, but include a forward slash like this `</TAG>`. The content of the element goes between the start and end tags. For example:

```
<table>
    . . .
    . . .
</table>
```

As in LaTeX, you are not required to tab the content of a table element;

however, it does make the markup document easier to read and, as the number of tags proliferates, easier to write.

You can specify element attributes inside of start tags.[14] For example, to add a border to the table, use: `<table border="1">`.[15]

Table rows are put inside of `tr` (table rows) element tags. Individual cells are delimited with `td` (standard cell) tags. Here is what the first row of our example table looks like in basic HTML:

```
<table>
  <tr>
    <td>Observation</td> <td>Variable1</td> <td><Variable2/td>
  </tr>
```

We can further delimit a table's header row(s) from its body with the `thead` and `tbody` tags. Finally, before making a full table it's useful to mention that table captions can be included with `caption` tags. Let's put this all together:

```
<table>
  <thead>
    <tr>
      <td>Observation</td> <td>Variable1</td> <td>Variable2</td>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Subject1</td> <td>a</td> <td>b</td>
    </tr>
    <tr>
      <td>Subject2</td> <td>c</td> <td>d</td>
    </tr>
    <tr>
      <td>Subject3</td> <td>e</td> <td>e</td>
    </tr>
    <tr>
      <td>Subject4</td> <td>f</td> <td>f</td>
    </tr>
  </tbody>
</table>
```

[14] These work like arguments in R in that they change how the element is evaluated.

[15] Whether or not a border appears is determined by whether or not the style sheet you are using includes borders.

As with Markdown tables, the ultimate appearance of the table is highly dependent on the style files you use.

9.3 Creating Tables from Supported Class R Objects

Just as the `write.csv()` function turns an R data frame into a CSV formatted text file, there are a number of methods in R to take an object, e.g. a matrix, data frame, the output from a statistical analysis, and so on, and turn them into LaTeX and HTML tables. `kable()`, *xtable*, and *texreg* each work most easily with specific object classes that their designers explicitly supported.

9.3.1 `kable` for Markdown and LaTeX

`kable()` easily converts matrices and data frames into tables for Markdown, HTML, and LaTeX among others. Let's create a simple data frame:

```
library(knitr)

kable_ex <- data.frame(
            Observation = c("Subject1", "Subject2",
                            "Subject3", "Subject4"),
            Variable1 = c("a", "c", "e", "g"),
            Variable2 = c("b", "d", "f", "c")
)
```

Then place this data frame into a `kable()` call:

```
kable(kable_ex, caption = "Example kable Table")
```

Beyond setting the table's caption with `caption`, there are a few other alterations that can be made with `kable` arguments. You can specify new column and row names by passing character vectors to `col.names` and `row.names`, respectively. These are very useful, as it can be difficult, or at least irritating, for your readers to try to decode the names you give to your data frame rows and columns in R. Another useful argument is `digits`. This will round numbers in the table to a specified number of digits after the decimal place. To effectively convey your results, you should *at least* only include digits that are significant in that they meaningfully vary in the data (Ehrenberg, 1977, 281).

You can also change the markup language that the table is created in using the `format` argument. For example, to create a LaTeX formatted table, use `format = 'latex'`. In general, you do not need to specify the format if you are using *knitr* or *rmarkdown* to include the table in a presentation document. This will be done automatically.

9.3.2 *xtable* for LaTeX and HTML

While `kable()` allows you to quickly create simple tables, it can only do so from matrices and data frames. It also has limited customizability. The *xtable* package can create more customizable tables from a wider variety of R objects, including statistical model objects.

Different R statistical model estimation commands can produce model objects of different classes. For example, the `lm()` (linear model) function creates model summaries of the `lm` class. Let's create a simple linear regression using the *swiss* data frame and `lm()`. This data frame is included with R by default. The simple linear regression model we are going to make has the *swiss* variable **Examination** as the dependent variable and **Education** as the only independent variable.[16]

```
# Fit simple linear regression model
M1 <- lm(Examination ~ Education, data = swiss)

# Return class
class(M1)
```

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

By using the `class` function, we can see that *M1* is of the `lm` class. *M1* contains items estimated by the linear regression model[17] such as the coefficient estimates and their standard errors. To get a summary of a model object's contents, use the `summary()` function like this:

```
summary(M1)
```

```
##
```

[16] For a description of these variables, type `?swiss` into the console.

[17] If you are unfamiliar with the syntax of R statistical estimation models, the previous code might be confusing. In general 'response' (Y) variables are written first and are separated from the 'explanatory' (X) variables by a tilde (`\sim`). Crawley (2005, 107) notes that you can read $Y \sim X$ as 'Y is modeled as a function of X'. In later examples we will see that individual explanatory variables are generally separated by plus signs (`+`), indicating that they are included in the model, not that they are added. For more information, see Crawley (2005, Ch. 7).

```
## Call:
## lm(formula = Examination ~ Education, data = swiss)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -10.932  -4.763  -0.184   3.891  12.498
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  10.1275     1.2859    7.88  5.2e-10 ***
## Education     0.5795     0.0885    6.55  4.8e-08 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.77 on 45 degrees of freedom
## Multiple R-squared:  0.488,  Adjusted R-squared:  0.476
## F-statistic: 42.9 on 1 and 45 DF,  p-value: 4.81e-08
```

To find a full list of object classes that *xtable* supports, type `methods(xtable)` into the R Console after you have loaded the package.

xtable for LaTeX

Let's look at how to create LaTeX tables with *xtable* by creating a table summarizing the estimates from the *M1* model object.

```
<<results=asis, echo=FALSE>>=
library(xtable)

# Create LaTeX table from M1 and show the output markup
xtable(M1,
       caption = "Linear Regression, DV: Exam Score",
       label = "BasicXtableSummary",
       digits = 1)
@
```

When included in an R Sweave-style LaTeX document, this code will create a table exactly like Table 9.2.

Let's go through this code, working from the outside in. First you'll notice that we've set two *knitr* code chunk options. As we discussed earlier, `results='asis'` allows us to include the LaTeX formatted table created by *xtable*. The next option `echo=FALSE` hides the code from being shown in our final document. The `xtable()` function creates the summary table of our *M1*

model object. Not only does it produce both complete `tabular` and `table` environments, but also through the `caption` and `label` arguments it automatically adds in the table's title and cross-reference label, respectively. Finally, notice that I added the `digits = 1` argument. As in `kable()`, this specifies that I want numbers in the table to be rounded to one decimal digit.

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	10.1	1.3	7.9	0.0
Education	0.6	0.1	6.5	0.0

TABLE 9.2: Linear Regression, DV: Exam Score

The caption is printed below the table by default.

xtable for Markdown/HTML

We can use *xtable* and the `print.xtable()` function[18] to also create tables for Markdown and HTML documents. The *xtable* function produces, unsurprisingly, `xtable()` class objects. We can run these through the `print()` function and add arguments to customize how the table is formatted. By default, `print.xtable()`'s `type` argument is set to `"latex"`. To create an HTML table that can be inserted into Markdown and HTML documents, set the `type` argument from `"latex"` to `"html"`. For example, to create an HTML version of the table summarizing *M1* and include it in an R Markdown document, type:

````
```{r results='asis', echo=FALSE}
library(xtable)

# Create an xtable object from M1
m1_table <- xtable(M1,
                caption = "Linear Regression, DV: Exam Score",
                label = "BasicXtableSummary",
                digits = 1)

# Create HTML summary table of m1_table
print.xtable(m1_table, type = "html", caption.placement = "top")
```
````

If you intend to include multiple tables in your R Markdown document, you will want to set all of the tables to be printed in HTML. You can place `options("xtable.type" = "html")` in a code chunk near the beginning of

[18] Note: you can abbreviate `print.xtable()` as `print()`.

your document.[19] This makes it so that you don't need to include `type = "html"` every time you use `print`.

Notice in the previous code example that we also added the `caption.placement = "top"` argument. This will move the caption from the bottom of the table, as it is in Table 9.2, to the top. See the *xtable* package documentation[20] for the full list of `print.xtable()` options.

9.3.3 *texreg* for LaTeX and HTML

`kable()` and *xtable* are limited when it comes to creating tables from statistical model objects. `kable` only works with matrices and data frames. *xtable* is easiest when working with only one model object at a time. Furthermore, by default these tools do not create output tables that present estimates from multiple statistical models in the style used by many prominent academic journals. The *texreg* package is very useful for creating these types of tables. It also supports more model object types than *xtable*.

texreg for LaTeX

Imagine we want to show the estimates from a number of nested regression models in LaTeX as the next table. For example, to estimate nested regression models from the remaining variables in the *swiss* data set, we type:

```
# Estimate nested regression models
M2 <- lm(Examination ~ Education + Agriculture, data = swiss)

M3 <- lm(Examination ~ Education + Agriculture + Catholic,
            data = swiss)

M4 <- lm(Examination ~ Education + Agriculture + Catholic +
            Infant.Mortality, data = swiss)

M5 <- lm(Examination ~ Education + Agriculture + Catholic +
            Infant.Mortality + Fertility, data = swiss)
```

We can now include these model objects in one LaTeX table with *texreg*. Remember to include `results='asis'` in the code chunk head.

[19] Of course, you will probably want to use the `include=FALSE` *knitr* option with this code chunk.

[20] `https://cran.r-project.org/web/packages/xtable/xtable.pdf`

TABLE 9.3: Nested Estimates Table with *texreg*

	Model 1	Model 2	Model 3	Model 4	Model 5
(Intercept)	10.13***	19.72***	18.54***	18.66**	24.57**
	(1.29)	(3.20)	(2.64)	(5.84)	(8.24)
Education	0.58***	0.36**	0.42***	0.42***	0.33*
	(0.09)	(0.10)	(0.09)	(0.09)	(0.13)
Agriculture		−0.14**	−0.07	−0.07	−0.08
		(0.04)	(0.04)	(0.04)	(0.04)
Catholic			−0.08***	−0.08***	−0.07**
			(0.02)	(0.02)	(0.02)
Infant Mortality				−0.01	0.10
				(0.23)	(0.25)
Fertility					−0.10
					(0.09)
R^2	0.49	0.59	0.73	0.73	0.73
Adj. R^2	0.48	0.57	0.71	0.70	0.70
Num. obs.	47	47	47	47	47
RMSE	5.77	5.25	4.30	4.36	4.35

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

```
library(texreg)

# Create custom coefficient names
cust_coef <- c('(Intercept)', 'Education', 'Agriculture',
               'Catholic', 'Infant Mortality', 'Fertility')

# Create nested regression model table
texreg(list(M1, M2, M3, M4, M5),
    caption = 'Nested Estimates Table with \\emph{texreg}',
    caption.above = TRUE,
    label = 'Basic_texregTable',
    custom.coef.names = cust_coef)
```

Notice that we placed the model objects in a list when we called `texreg()`. `texreg()` automatically created the `table` and `tabular` environments and by default centers the table.[21] We added a caption and reference label with the `caption` and `label` arguments, respectively. By default, the caption is placed below the table, so we used `caption.above = TRUE` to place it on top. Finally, we created custom coefficient names with `custom.coef.names` that are a bit

[21] Use the `center = FALSE` argument to override centering. If you would like to only create the `tabular` environment, use the argument `table = FALSE`. Creating your own `table` environment can be useful in situations where you want more customizability.

tidier than the variable names in our R data set. Your readers will appreciate easily discernible coefficient names.

In the LaTeX caption, you'll notice `\\emph{texreg}`. In LaTeX the `emph` command italicizes text (we'll see this again in Chapter 11). We added an additional escape character `\` so that R would not try to interpret the `e` and instead feed it to LaTeX. By default, `texreg()` uses `stars = c(0.001, 0.01, 0.05)` to determine at what p-values to display statistical significance stars. This is the same as the `lm` model summary default showing three sets of statistical significance stars. You can define the significance levels by assigning a different numeric vector to the `stars` argument.

There are many other changes you can make to tables created with *texreg*. You can change the column and coefficient names, determine what type of standard errors to show, and so on. For the full list of arguments, see the help file by typing `?texreg` into your R Console.

texreg for HTML

You can also use the *texreg* package to create tables in Markdown/HTML documents. Instead of the `texreg` function, use `htmlreg`. The syntax is largely similar, though arguments relating to LaTeX are not available, while others relating the HTML are. Here is a simple example:

```
htmlreg(list(M1, M2, M3, M4, M5),
        caption = 'Nested Estimates Table in HTML Document',
        caption.above = TRUE, custom.coef.names = cust_coef)
```

Notice that we did not include the `label` argument as this is not available in HTML. The resulting table looks like this:

Nested Estimates Table in HTML Document

	Model 1	Model 2	Model 3	Model 4	Model 5
(Intercept)	10.13***	19.72***	18.54***	18.66**	24.57**
	(1.29)	(3.20)	(2.64)	(5.84)	(8.24)
Education	0.58***	0.36**	0.42***	0.42***	0.33*
	(0.09)	(0.10)	(0.09)	(0.09)	(0.13)
Agriculture		-0.14**	-0.07	-0.07	-0.08
		(0.04)	(0.04)	(0.04)	(0.04)
Catholic			-0.08***	-0.08***	-0.07**
			(0.02)	(0.02)	(0.02)
Infant Mortality				-0.01	0.10
				(0.23)	(0.25)
Fertility					-0.10
					(0.09)
R^2	0.49	0.59	0.73	0.73	0.73
Adj. R^2	0.48	0.57	0.71	0.70	0.70
Num. obs.	47	47	47	47	47

***p < 0.001,** p < 0.01, p < 0.05

9.3.4 Fitting large tables in LaTeX

Sometimes you may have large tables that are difficult to fit onto a page in LaTeX. There are a number of ways to adjust tables so that they fit on the page.

LaTeX landscape tables

If your LaTeX table is very wide, e.g. because it shows results from many estimation models, you can use LaTeX's `lscape` package to create `landscape` formatting environments. Rather than orienting the text of a page so that it is in profile (a long page), a `landscape` environment turns it 90 degrees so that it has a landscape orientation (a wide page).

To use the *lscape* package, first place `\usepackage{lscape}` in your LaTeX document's preamble. Then begin a `landscape` environment with `\begin{landscape}` where you would like it located in the text. Then place the `table` environment information and *knitr* code for creating the table. Finally, close the `landscape` environment with `\end{landscape}`.

LaTeX scalebox for tables

In addition, the `scalebox` command from the *graphics* package could be useful for fitting large tables onto a PDF page. This command expands or shrinks the text in the table. `texreg` actually has a `scalebox` argument. If you use `scalebox = 0.5`, it will halve the size of the table; `scalebox = 2` doubles it.

More generally, to rescale a table use:

```
\scalebox{HORIZONTAL_SCALE}[VERTICAL_SCALE]{TABLE}
```

`HORIZONTAL_SCALE` is how much to scale the table horizontally. `VERTICAL_SCALE` is how much to scale vertically and `TABLE` is the table or R code chunk to create the table.

9.3.5 *xtable* with non-supported class objects

The `kable`, *texreg*, and *xtable* packages are very convenient for model objects they know how to handle. With supported class objects, the functions in these packages know where to look for the vectors containing the things—coefficient names, standard errors, and so on—that they need to create tables. With unsupported classes, however, they don't know where to look for these things. Luckily, there is a work-around. You tell `xtable()` where to find elements you want to include in your table. `xtable()` can handle matrix and data frame class objects. The rows of these objects become the table rows and the columns become the table columns. So, to create tables with non-supported class objects you need to:

1. find and extract the information from the unsupported class object that you want in the table,
2. convert this information into a matrix or data frame where the rows and columns of the object correspond to the rows and columns of the table that you want to create,
3. use *xtable* with this object to create the table.

Imagine that you want to create a results table showing the covariate names, coefficient means, and quantiles for marginal posterior distributions estimated from an linear regression using the *brms* package (Bürkner, 2020) and data from the *swiss* data frame. Let's fit the model:

```
library(brms)

# Fit model
```

```
linear_brms <- brm(Examination ~ Education,
                   data = swiss,
                   family = gaussian(link = "identity"),
                   refresh = 0)

# Find linear_brms's class
class(linear_brms)
```

```
## [1] "brmsfit"
```

Note: I included `refresh = 0` to suppress output about the model fitting process.

Using the `class()` function, we see that the model output object in *linear_brms* is of the `brmsfit` class. This class is not supported by *xtable*. If you try to create a table summarizing the estimates in *linear_brms_table*, you will return an error telling you the object's class is not supported.

With unsupported class objects, you have to create the summary yourself and extract the elements that you want from it manually. A good knowledge of vectors, matrices, and component selection is very handy for this (see Chapter 3).

First, create a summary of your output object *linear_brms*:

```
linear_brms_summary <- summary(linear_brms)
```

```
## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

This creates a new object of the class `brmssummary`. We're still not there yet as this object contains not just the covariate names and so on, but also information we don't want to include in the results table, like the estimation formula. The second step is to extract a matrix from inside *linear_brms_summary* called *summary* with the component selector (`$`). Remember that to find the components of an object, use the `names()` function.

```
names(linear_brms_summary)
```

```
##  [1] "formula"   "data.name" "group"     "nobs"
##  [5] "ngrps"     "autocor"   "prior"     "algorithm"
##  [9] "chains"    "iter"      "warmup"    "thin"
## [13] "sampler"   "fixed"     "spec_pars" "cor_pars"
```

The *fixed* matrix is where the things we want in our table are located. I find it

easier to work with data frames, so let's also convert the matrix into a data frame.

```
linear_brms_summary_df <- data.frame(linear_brms_summary$fixed)
```

Here is what the model summary data frame looks like:

```
linear_brms_summary_df
```

```
##           Estimate Est.Error l.95..CI u.95..CI  Rhat
## Intercept  10.1173   1.29142   7.6324   12.641 1.002
## Education   0.5802   0.08866   0.4035    0.756 1.000
##           Bulk_ESS Tail_ESS
## Intercept     3687     2960
## Education     3584     2840
```

Now we have a data frame object *xtable* can handle. After a little cleaning up (see the chapter's Appendix for more details) you can use *xtable* as before to create Table 9.4.

TABLE 9.4: Coefficient Estimates Predicting Examination Scores in Swiss Cantons (1888) Found Using Bayesian Linear Regression

	2.5%	50%	97.5%
Intercept	7.63	10.12	12.64
Education	0.40	0.58	0.76

It may take some hunting to find what you want, but a similar process can be used to create tables from objects of virtually any class.[22] Hunting for what you want can be easier if you look inside of objects by clicking on them in RStudio's *Environment* tab.

9.3.6 Creating variable description documents with *xtable*

You can use *xtable* to create a table describing variables in your data set and insert these into Markdown documents created with the concatenate and print (`cat`) command (see Section 4.4). This is useful because our data so far has been stored in plain-text files. Unlike binary Stata or SAS data files, plain-text data files do not include variable descriptions.

Imagine that we want to create a Markdown file with a table describing the variables from the *swiss* data frame. First we will create two vectors: one for the variable names and the other for the variable descriptions.

[22] This process can also be useful for creating graphics as we will see in Chapter 10.

```
# Create variable vector from column names
Variable <- names(swiss)

# Create variable description vector
Description <- c("common standardized fertility measures",
        "% of males involved in agriculture as occupation",
        "% draftees receiving highest mark on army examination",
        "% education beyond primary school for draftees",
        "% catholic",
        "% live births who live less-than 1 year"
)
```

In the first line we use the `names()` function to create a vector of the *swiss* data frame's column names. Then we create a vector of descriptions with the combine function (`c()`). Now we can combine these vectors into a matrix and use it to create an HTML table.

```
# Combine Variable and Description variables into a matrix
descriptions_bound <- cbind(Variable, Description)

# Create an xtable object from descriptions_bound
descriptions_table <- xtable(descriptions_bound)

# Format table in HTML
descript_table <- print.xtable(descriptions_table, type = "html")
```

Finally, we can use `cat()` to create our Markdown variable description file.

```
# Create variable description file
cat("# Swiss Data Variable Descriptions \n",
    "### Source: Mosteller and Tukey, (1977) \n",
    descript_table,
    file = "swiss-variable-descriptions.md"
)
```

The first part of the `cat()` function here is the title of the document. As we will see in Chapter 12, hashes (`#`) create headers. `\n` creates a new line in the Markdown document. The next line is information on the *swiss* data frame's source. We then include the HTML table in the *descript_table* object and save it to a file called *swiss-variable-descriptions.md*.

It is convenient to include the creation of this table in your data gathering makefiles and have it saved into the same directory as your data. This way it will be easy to update as you update your data and easy to find. If you

are storing your data on GitHub, it will automatically render the variable description Markdown file and make it easy for others to read. See this book's makefile example for more information: `https://bit.ly/2UtvOys`.[23]

Chapter summary

In this chapter, we have learned how to take the results from our statistical analyses and other information from our data and dynamically present it in LaTeX and Markdown documents with knitr/R Markdown. In the next chapter, we will do the same thing with figures.

Appendix

Source code for cleaning *linear_brms_summary_df* and using it to create a LaTeX table:

```
library(dplyr)
library(xtable)

# Change posterior summary variable names
linear_brms_summary_df <- rename(linear_brms_summary_df,
    `2.5%` = `l.95..CI`)
linear_brms_summary_df <- rename(linear_brms_summary_df,
    `50%` = Estimate)
linear_brms_summary_df <- rename(linear_brms_summary_df,
    `97.5%` = `u.95..CI`)

# Reorder variables and remove the Est. Error
linear_brms_summary_df <- linear_brms_summary_df[,
    c("2.5%", "50%", "97.5%")]

# Create table
xtable(linear_brms_summary_df,
    caption = "Coefficient Estimates Predicting
    Examination Scores in Swiss Cantons (1888)
    Found Using Bayesian Linear Regression",
    label = "CoefEstTable")
```

[23] The long URL is: `https://github.com/christophergandrud/rep-res-book-v3-examples/tree/master/data`.

```
# Create table
xtable(linear_brms_summary_df,
    caption = "Coefficient Estimates Predicting
    Examination Scores in Swiss Cantons (1888)
    Found Using Bayesian Normal Linear Regression")
```

Note that the new variable names are in quotation marks, in contrast to the example from Chapter 7. The quotation marks allow us to specify a name that begins with a number and has special characters like the percent sign.

10

Showing Results with Figures

One of the main reasons that many people use R is to take advantage of its comprehensive and powerful set of data visualization tools. Visually displaying information with graphics is often a much more effective way of presenting both descriptive statistics and analysis results than the tables we covered in the last chapter.[1]

Nonetheless, dynamically incorporating figures with knitr/R Markdown has many of the same benefits as dynamically including tables, especially the ability to have data set or analysis changes automatically cascade into your presentation documents. The basic process for including figures in knitted presentation documents is also very similar to including tables, though there are some important extra considerations we need to make to properly size the figures and be able to include interactive visualizations in our presentation documents.

In this chapter we will first learn how to include non-knitted graphics in LaTeX and Markdown documents before turning to dynamically knit R graphics into presentation documents. In the remainder of the chapter, we will look at how to actually create graphics with R including some of the fundamentals of R's default graphics package, as well as the *ggplot2* (Wickham et al., 2019a) and *googleVis* (Gesmann and de Castillo, 2019) packages. In each case we will focus on how to include the figures created by these packages in knitted presentation documents.

[1] There are, of course, a number of exceptions to this rule of thumb. van Belle (2008, Ch. 9) argues that a few numbers should be listed in a sentence, many numbers shown in tables, and relationships between numbers are best shown with graphs. Similarly, Tufte (2001) argues that tables tend to outperform graphics for displaying 20 or fewer numbers. Graphics often outperform tables for showing larger data sets and relationships within the data.

10.1 Including Non-knitted Graphics

Understanding how *knitr*/*rmarkdown* dynamically include figures is easier if you understand how figures are normally included in LaTeX and Markdown. Unlike a word processing program like Microsoft Word, in LaTeX, Markdown, HTML, and other markup languages you don't copy and paste figures into your document. Instead, you link to an image file outside of your markup document. Typically these image files are in formats such as *PDF*, *PNG*, and *JPEG*.[2]

While you lose the flexibility of drag and drop, there are advantages to this method of including graphics. The first is that whenever the image files are changed, the changes are updated in the final presentation document when it is compiled, no recopying and pasting. The second advantage is that the images are sized and placed with the markup code rather than pointing and clicking. This is tedious at first, but saves considerable time and frustration when a document becomes larger. It also makes it easy to consistently format multiple images in a document.

If the image files are in the same directory as the markup document, we don't need to specify the image's full file path, only its name. If they are in another directory, we need to include additional file path information. Remember to use relative paths when possible. In this section we will learn how to include graphics files in documents created with LaTeX and Markdown.

10.1.1 Including graphics in LaTeX

The main way to include graphics (graphs, photos, and so on) in LaTeX documents is to use the `includegraphics` function to link to image files. To have the full range of features for `includegraphics`, make sure to load the *graphicx* package in your document's preamble. Imagine that we wanted to

[2] PDF: Portable Document Format, PNG: Portable Network Graphic, JPEG: Joint Photographic Experts Group.
A quick note about file formats: By default, *knitr* creates PDF-formatted figure files when knitting R LaTeX documents. These figures, generally built with vector graphics, allow you to zoom in on them by any amount without them becoming pixelated. This means that your images will be crisp in PDF presentation documents. For Markdown documents, *knitr* creates PNG images. PNG images are usually relatively high quality and can be rendered directly on websites, unlike PDFs. JPEG formatted files usually take up less disk space than PDF and PNG files. However, their quality is also worse and can often look very pixelated. For more information, Wikipedia has a comprehensive comparison of graphics file formats at: `https://en.wikipedia.org/wiki/Comparison_of_graphics_file_formats`.

include an image of butterflies stored in a file called *HeliconiusMimicry.png* in a LaTeX-produced document.[3] We type:

```
\includegraphics[scale=0.8]{HeliconiusMimicry.png}
```

In the square brackets, you'll notice `scale=0.8`. This formats the image to be included at 80 percent of its actual size. You can use other options such as `height` to specify the height, `width` to specify the width, and `angle` to specify the angle at which to rotate the image. You can add more than one option if they are separated by commas. Rather than hard coding the width in exact centimeters, you can determine its width as a proportion of the text width using `\textwidth`.[4] For example, to set our image at 80 percent of the text width we can type:

```
\includegraphics[scale=0.8\textwidth]{HeliconiusMimicry.png}
```

`figure` float environment

Most often you will want to include LaTeX figures in a `figure` float environment. The *figure* environment works almost exactly the same way as the `table` environment we saw in the last chapter. It allows you to separate the figure from the text, add a caption, and label the figure. We begin the environment with `\begin{figure}[POSITION_SPEC]`. `POSITION_SPEC` can have the same values as we saw earlier with tables in Chapter 9. We can then include a `caption` and `label` function. The environment is closed with `\end{figure}`. For example, to create Figure 10.1 exactly as is, I used the following code:[5]

```
\begin{figure}[ht]
        \begin{center}
            \includegraphics{HeliconiusMimicry.png}
        \end{center}
    \caption{An Example Figure in LaTeX}
    {\scriptsize{Image source: \cite{meyer2006}}}
    \label{ExampleLaTeXFigure}
\end{figure}
```

Notice that after the call to end the `center` environment we include

[3] The image used here is from Meyer (2006).

[4] Note there are a number of other ways to set the size of a figure relative to a page element. See the LaTeX Wiki Book for more details: `https://en.wikibooks.org/wiki/LaTeX/Page_Layout`.

[5] For simplicity, this code does not include the full image's actual file path.

`{\scriptsize{Source: \cite{meyer2006}}}`. This includes a note in the figure environment giving the image's source. The note moves with the figure and is separate from the text. The `scriptsize` function transforms the text to smaller than normal size font. See Chapter 11 for more details on LaTeX font sizes. The function `\cite{meyer2006}` inserts a citation from the bibliography for Meyer (2006). We will also discuss bibliographies in more detail in Chapter 11.

FIGURE 10.1: An Example Figure in LaTeX
Image source: Meyer (2006)

10.1.2 Including graphics in Markdown/HTML

Markdown has a similar function as LaTeX's `includegraphics`. It goes like this: `![ALT_TEXT](FILE_PATH)`. This syntax may seem strange now, but it will hopefully make more sense when we cover Markdown hyperlinks in Chapter 12. This is what it is intended to imitate. `ALT_TEXT` refers to HTML's `alt` (alternative text) attribute. This should be a very short description of the image that will appear if it fails to load in a web browser. `FILE_PATH` specifies the image's file path.[6] Here is an example using the image we worked with before.

```
![ButterflyImage](HeliconiusMimicry.png)
```

[6] You can also include a title in quotation marks after the file path. This specifies the HTML `title` attribute. However, this attribute does not create a title for the image in the way that `caption` does for LaTeX float figures. Instead, it creates a tooltip, a small box that appears when you place your cursor over the image. Specifying descriptive alt text is very useful for screen readers that help visually impaired people access web content.

Note that the file path can be a URL. You may, for example, store an image on GitHub and use its raw URL to link to it in the Markdown document.[7]

Markdown does not easily include ways to resize or reposition an image. If you want to resize or reposition your image, it is often most straightforward to use HTML markup. Probably the simplest way to include images with HTML is by using the `img` (image) element tag. To create the equivalent of what we just did in Markdown with HTML, type:

```
<img src="HeliconiusMimicry.png" alt="ButterflyImage"></img>
```

The `src` (script) attribute specifies the file path. To change the width and height of the image, use the `width` and `height` attributes. For example:

```
<img src="HeliconiusMimicry.png" alt="ButterflyImage"
     width="100px" height="100px">
</img>
```

creates an image that is 100 pixels (`px`) wide by 100 pixels high.[8] It is also possible to specify the alignment of figures in Markdown with a custom CSS style file. I don't cover how to do that here.

10.1.3 Non-knitted graphics with *knitr*/*rmarkdown*

Now that we've seen how LaTeX, Markdown, and HTML include non-dynamically generated graphics, it raises a question: how do we include these graphics in a document that we intend to use *R Markdown* to compile to more than one of these formats? *knitr* includes the `include_graphics()` function just for this purpose.

For example, in a code chunk place:

````
```{r fig.cap="An Example Figure"}
knitr::include_graphics('HeliconiusMimicry.png')
```
````

Now the figure will be included regardless of which markup language we compile to. Notice the code chunk option `fig.cap`. In the next section, we discuss this type of *knitr* options in detail.

[7] Use the URL for the raw version of the file for images stored on GitHub.

[8] A pixel is the smallest discrete part of images displayed on a screen. See the "pixel" Wikipedia page for more details: `https://en.wikipedia.org/wiki/Pixel`.

10.2 Basic *knitr*/*rmarkdown* Figure Options

In addition to including precompiled images with the `include_graphics()` function, *knitr*, and by extension *rmarkdown*, allows us to combine a figure's creation by R with its inclusion in a presentation document. They are tied together and update together. We use *knitr* chunk options to specify how the figure will look in the presentation document and where it will be saved. We can also use them to specify captions. Let's learn some of the more important chunk options for figures.

10.2.1 Chunk options

`fig.path`

When you use *knitr* to create and include figures in your presentation documents, it (1) runs the code you give it to create the figure, (2) automatically saves it into a particular directory,[9] and (3) includes the necessary LaTeX or Markdown code to include the figure in the final presentation document. By default, *knitr* saves images into a folder (it creates) called *figure* located in the working directory.[10] You can tell *knitr* where to save the images with the `fig.path` option. Simply use the file path naming conventions suitable for your system and include the new path in quotation marks.

Note if you use *rmarkdown* to compile to HTML, by default the graphic will not be saved in a separate file, but instead converted to a format that is embedded directly in the HTML markup document.

`out.height`

To set the height that a figure will be in the final presentation document, use the `out.height` option. In R LaTeX documents, you can set the width using centimeters, inches, or as a proportion of a page element. In R Markdown documents, you use pixels to set the height. For example, to set a figure's height in an R Markdown document to 200 pixels, use `out.height='200px'`.

[9] If a code chunk creates more than one figure, *knitr* automatically saves each into its own file in the same directory.

[10] File names are based on the code chunk label where they were created.

`out.width`

Similarly, we can set the width of a *knitr* created figure using the `out.width` option. The same rules apply as with `out.width`. For example, to have a figure shown up at 80 percent of the text width in an R LaTeX document, use: `out.width='0.8\\textwidth'`. Notice that there are two backslashes before `textwidth`. As we saw earlier, the LaTeX function only has one. However, all *knitr* code chunk options must be written as they would be in R. We need to escape the backslash with the backslash escape character, i.e. use two backslashes.

`fig.align`

You can set a knitted figure's alignment using `fig.align`. The option can be set to `left`, `center`, or `right`. To center a figure, add `fig.align='center'`.

`fig.cap`

If your document compiles to LaTeX, you can use the `fig.cap` option to set the figure's caption as we did in the example above.

Other figure chunk options

The previous options are probably the most commonly used ways of adjusting figures with *knitr*. However, *knitr* has many other chunk options to help you adjust your figures so that they are incorporated into your presentation documents the way that you want. For example, the option `fig.lb` allows you to set the label.[11] As we will see below, you can use the `dev` option to choose the figure's output file format, e.g. PDF, PNG, JPEG. Please see the official *knitr* code chunk options webpage for more information on figure chunk options: `https://yihui.name/knitr/options/#chunk_options`.

10.2.2 Global options

If you want all of your figures to share the same options—e.g. same height and alignment—you can set global figure options at the beginning of your document with `opts_chunk$set`. Imagine that we are making an R LaTeX Sweave-style document and want all of our figures to be center aligned and 80 percent of the text width. We type:

[11] In this chapter we will set this option in the markup rather than the code chunk. I prefer doing this because *knitr* options need to be on the same line and so they can sometimes result in very long lists of options that are difficult to read.

```
opts_chunk$set.(fig.align = "center",
                out.width = "0.8\\textwidth")
```

You can also set some global figure options, such as `fig_height` and `fig_width` in your *rmarkdown* YAML header.

10.3 Knitting R's Default Graphics

R's *graphics* package, loaded by default, includes functions to create numerous plot types. These include `hist()` for histograms, `pairs()` for scatterplot matrices, `boxplot()` for creating boxplots, and the versatile `plot()` for creating x-y plots, including scatterplots and bar charts depending on the data's type.

There are many useful resources for learning how to fully utilize R's default graphics capabilities. These include Paul Murrell's (2011) comprehensive *R Graphics* book. The Cookbook for R[12] and Quick-R[13] websites are also helpful. Winston Chang (2012), the maintainer of the Cookbook for R, also has a full book devoted to creating R graphics. Kieran Healy (2018) is a strong intoduction to data visualisation in general with R examples.

In this section we are going to see how to include R's default graphics in our LaTeX and Markdown presentation documents. We will also see an example of how to source the creation of a graph from a segmented analysis file. Most of R's default graphics capabilities create static graphics. They are not animations or interactive. The discussion in this section is exclusively about using static graphics with *knitr/rmarkdown*. Later in the chapter, we will discuss how to knit interactive graphics.

Let's look at an example we first saw at the end of Chapter 8. Remember that we accessed an R source code file stored on GitHub to create a simple scatterplot of cars' speed and stopping distances using R's *cars* data set, which is loaded by default. We haven't yet seen the code in the R source file that created the plot.

In the *cars* data frame, the variable **speed** contains the stopping speed, and **dist** contains the stopping distances. Here is the code to create the plot:

```
# Create simple scatterplot of cars' speed and stopping distance
plot(x = cars$speed, y = cars$dist,
```

[12] http://www.cookbook-r.com/Graphs/
[13] http://www.statmethods.net/advgraphs/

```
     xlab = "Speed (mph)",
     ylab = "Stopping Distance (ft)",
     cex.lab = 1.5)
```

We select the variables from *cars* to plot on the x- and y-axes of our graph with the component selector (`$`). Then we use the `xlab` and `ylab` arguments to specify the x- and y-axes labels. We could have added a title for the plot using the `main` argument. We didn't do this because we will give the plot a title in the LaTeX `figure` environment. The `cex.lab` argument increased the labels' font size. The argument specifically determines how to scale the labels relative to the default size: 1.5 means 50 percent larger than the default.

Now let's see how to create this plot with *knitr* and include it in a LaTeX `figure` environment.

```
\begin{figure}[ht]
```{r echo=FALSE, fig.align='center', out.width='8cm'}
    plot(x = cars$speed, y = cars$dist,
         xlab = "Speed (mph)",
         ylab = "Stopping Distance (ft)",
         cex.lab = 1.5)
```
    \caption{Example Simple Scatterplot Using \texttt{plot}}
    \label{BasicFigureExample}
\end{figure}
```

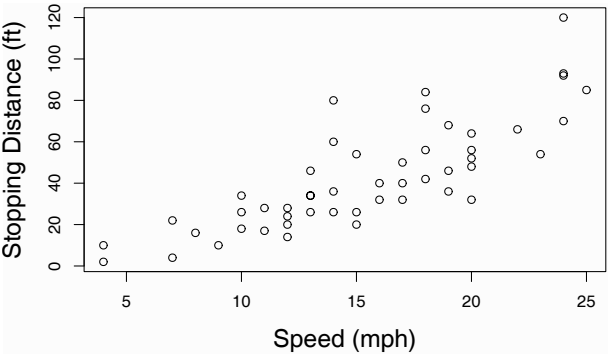


FIGURE 10.2: Example Simple Scatter Plot Using `plot`

This code produces Figure 10.2.[14] If you are familiar with R graphics, you

[14]Note that I did not specify the center environment. This is because it is specified in a *knitr* global chunk option.

will notice that we did not need to tell *knitr* to save the file in a particular format. Instead, behind the scenes it automatically saves the plot as a PDF file in a folder called *figure* that is a child of the current working directory. You can choose the figure file's format with the `dev` (graphical device) chunk option. For example, to save the figure in a PNG formatted file, add the chunk option `dev='PNG'`. You can choose any graphical device format supported by R. For a full list of R's graphical devices, type `?Devices` into your console. One reason you might want to change the format is to reduce your presentation document's file size. Using a bitmap format like PNG will create smaller files than PDFs, though lower-quality images.

We could, of course, link to the original R source code file stored on GitHub with the `source_url()` function. Let's look at an example of this with a different source code file. Remember in Chapter 6 we used a makefile to gather data from three different sources on the internet. The CSV is called *main-data.csv* and is stored on GitHub at: `http://bit.ly/V0ldsf`.[15] We can download this data into R and make the following scatterplot matrix (Figure 10.2) with this code:

```
# Download data
main_data <- rio::import("http://bit.ly/V0ldsf",
                         format = "csv" )

# Subset main_data so that it only includes the year 2003
data_sub <- subset(main_data, year == 2003)

# Remove iso2c, country, year variables
# Keep reg_4state, disproportionality, FertilizerConsumption
data_sub <- data_sub[, c("reg_4state", "disproportionality",
                         "FertilizerConsumption")]

# Create a scatterplot matrix
pairs(x = data_sub)
```

This is a lot of code, but you should be familiar with most of it. You will notice that after downloading the data we cleaned it up in preparation for plotting with the `pairs()` function by removing data from all years other than 2003 and all of the country-year identifying variables. Finally, we created the scatterplot matrix with `pairs()`.

To dynamically include the plot in our final document, we don't need to include all of this code in a code chunk in our markup document. A file containing the

[15] The full version of the URL is: `https://raw.githubusercontent.com/christophergandrud/rep-res-book-v3-examples/master/data/main-data.csv`

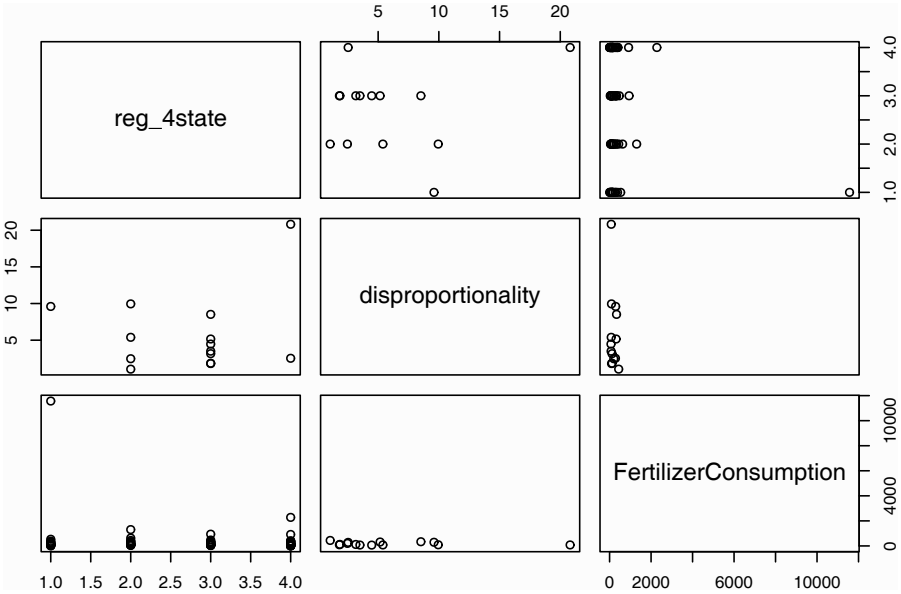


FIGURE 10.3: Example of a Scatterplot Matrix

code is available on GitHub.[16] So we only need to use `source_url()` to link to it. I've shortened the raw source code file's URL to: `http://bit.ly/TE0gTc`. Let's look at the syntax for knitting this into an R Markdown file:

```
```{r echo=FALSE, fig.cap='Example of a Scatterplot Matrix'}
# Create scatterplot matrix from main-data.csv
devtools::source_url("http://bit.ly/TE0gTc")
```
```

Because we have linked all the way back to the original data set *main-data*, any time it is updated by the makefile, the update will automatically cascade all the way through to our final presentation document the next time we knit it.

[16] See: `https://raw.githubusercontent.com/christophergandrud/Rep-Res-Examples/master/Graphs/ScatterPlotMatrix.R`.

10.4 Including *ggplot2* Graphics

The *ggplot2* package[17] (Wickham et al., 2019a) is probably one of the most popular packages for making graphics with R. It greatly expands the aesthetic and substantive tools R has for displaying quantitative information. Figures created with *ggplot2* are (generally) static,[18] so they are included in knitted documents the same way as most of R's default graphics.

There are a number of very good resources for learning how to use *ggplot2*. These include Hadley Wickham's *ggplot2* book (2009) and article (2010). The official *ggplot2* website[19] has up-to-date information. I've also found the Cookbook for R website helpful.[20]

Given that there is already extensive good documentation on *ggplot2*, we are not going to learn the full details of how to use the package here. Instead, let's look at some examples of how to manipulate a data frame and a regression results object so that they can be graphed with *ggplot2*. First we will create a multi-line time series plot. Then we will create a caterpillar plot of regression results. Along with giving you a general sense of how *ggplot2* works, the examples illuminate how *ggplot2* can be made part of a fully reproducible research workflow.[21]

Sometimes we may want to show how multiple variables change together overtime. For example, imagine we have data on inflation in the United States along with inflation forecasts made by the US Federal Reserve two quarters beforehand. The data is stored on GitHub at: `https://raw.githubusercontent.com/christophergandrud/Rep-Res-Examples/master/Graphs/InflationData.csv`.[22] I've loaded the data into R and put it into an object called *inflation_data*. It looks like this:

```
names(inflation_data)
```

```
## [1] "Quarter"             "ActualInflation"
## [3] "EstimatedInflation"
```

[17] "GG" stands for grammar of graphics and "2" indicates that it is the second major version of the package.

[18] It is possible to combine a series of figures created with *ggplot2* into an animation. For a nice example of an animation using *ggplot2*, see Jerzy Wieczorek's animation of 2012 US presidential campaigning: `http://bit.ly/UUVKka`.

[19] `http://docs.ggplot2.org/current/`

[20] `http://wiki.stdout.org/rcookbook/Graphs/`

[21] Note that everything we do here with *ggplot2* can also be done with R's default graphics, though the appearance will be different.

[22] This data is from Gandrud and Grafström (2015). The example here partially recreates Figure 1 from that paper.

We want to create a plot with **Quarter** as the x-axis, inflation as the y-axis, and two lines. One line will represent **ActualInflation** and the other **EstimatedInflation**. To do this, we need to reshape our data so that the inflation variables are in long format like this:

Quarter	Variable	Value
1969.1	ActualInflation	
1969.1	EstimatedInflation	
1969.2	ActualInflation	
1969.2	EstimatedInflation	
...		

We can use the `pivot_longer` function from *tidyr* that we first saw in Chapter 7 to reshape the data. The variable identifying the observations in this case is `Quarter`. The **ActualInflation** and **EstimatedInflation** variables (in columns two and three) are the variables that we want to pivot. So let's pivot the data:

```
library(tidyr)

# Pivot inflation_data
inflation_long <- pivot_longer(inflation_data, cols = 2:3,
                               names_to = "variable")

inflation_long
```

```
## # A tibble: 304 x 3
##    Quarter variable           value
##      <dbl> <chr>              <dbl>
##  1   1969. ActualInflation    NA
##  2   1969. EstimatedInflation  4.55
##  3   1969. ActualInflation     3.5
##  4   1969. EstimatedInflation  4.80
##  5   1969. ActualInflation     3.5
##  6   1969. EstimatedInflation  5.28
##  7   1969. ActualInflation     3.3
##  8   1969. EstimatedInflation  5.15
##  9   1970. ActualInflation     3.7
## 10   1970. EstimatedInflation  5.54
## # ... with 294 more rows
```

Now we have a data set we can use to create our line graph with *ggplot2*.

Let's cover a few basic *ggplot2* ideas that will help us understand the following code better. First, plots are composed of layers including the coordinate system, points, labels, and so on. Each layer has aesthetics, including the

variables plotted on the x- and y-axes, label sizes, colors, and shapes. Aesthetic elements are defined by the `aes()` argument. Finally, the main layer types are called geometrics, including lines, points, bars, and text. Functions that set geometrics usually begin with `geom`. For example, the geometric to create lines is `geom_line()`.

```
library(ggplot2)

# Create plot
line_plot <- ggplot(data = inflation_long,
                    aes(x = Quarter, y = value,
                      color = variable, linetype = variable)) +
                    geom_line() +
                    scale_color_discrete(name = "",
                           labels = c("Actual", "Estimated")) +
                    scale_linetype(name = "",
                           labels = c("Actual", "Estimated")) +
                    xlab("Quarter") + ylab("Inflation") +
                    theme_bw(base_size = 15)
```

You can see we set the x- and y-axes using the **Quarter** and **value** variables. We told *ggplot* that elements in the geometric layer should have lines with different colors and line types (dashed, dotted, and so on) based on the value of **variable** that they represent. `geom_line` specifies that we want to add a line geometric layer.[23] `scale_color_discrete()` and `scale_linetype()` are used here to hide the plot's legend title with `name = ""` and customize the legend's labels with `labels = . . .`. You can also use them to determine the specific colors and line types you would like to use. `xlab()` and `ylab()` set the axes' labels. You can add a title with `ggtitle`. Finally, I added `theme_bw()` so that the plot would use a simple black-and-white theme. We added the argument `base_size = 15` to increase the plot's font size.

All of the code required to create this graph is on GitHub at: `https://bit.ly/2FaMkOJ`.[24] To knit the graph into a LaTeX document manually specifying the figure environment, type:

```
\begin{figure}[ht]
\caption{Example Multi-line Time Series Plot Created with
  \emph{ggplot2}} \label{ggplot2Line}
    \begin{center}
```

[23] Remember from Chapter 3 that functions must be followed by parentheses. These layers are functions so they need to be followed by parentheses.

[24] The full URL is: `https://raw.githubusercontent.com/christophergandrud/Rep-Res-Examples/master/Graphs/InflationLineGraph.R`.

```
```{r echo=FALSE, out.width='10cm'out.height='8cm'}
    # Create plot
    devtools::source_url("https://bit.ly/2FaMkOJ")
```
    \end{center}
\end{figure}
```

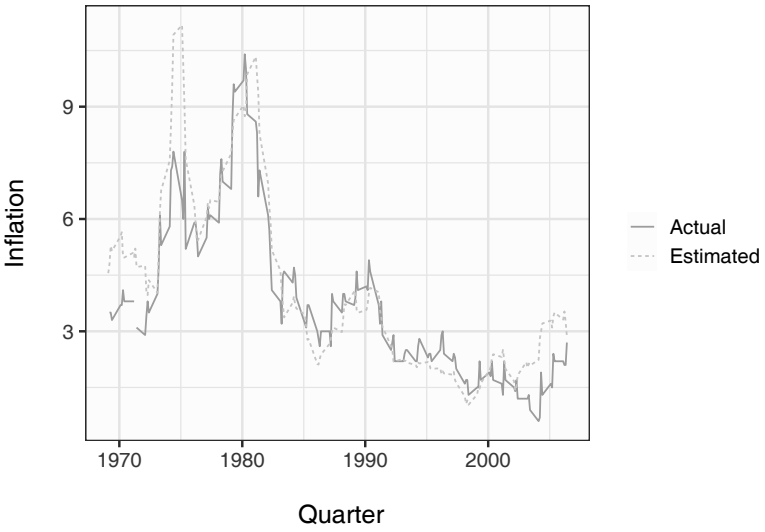


FIGURE 10.4: *ggplot2* Time Series Line Plot

The syntax for including this and other *ggplot2* figures in an R Markdown document is the same as we saw for default R graphics.

10.4.1 Showing regression results with caterpillar plots

Many packages that estimate statistical models from data in R have built-in plotting capabilities. For example, the *survival* package (Therneau, 2019) has the `plot.survfit()` function for plotting survival curves created using event history analysis. These plots can be knitted into presentation documents like the plots we have seen already.

However, sometimes either a package doesn't have built-in functions for plotting model results the way you want to and/or you want to use *ggplot2* to improve the aesthetic quality of the plots they do create by default. In either case, you can almost always create the plot that you want by first breaking into the model results object, extracting what you want, then plotting it with *ggplot2*.

The process is very similar to what we did in Chapter 9 to create custom tables.

To illustrate how this can work, let's create a caterpillar plot, like the following figure, showing the mean coefficient estimates and the uncertainty surrounding them from a Bayesian normal linear regression model using the *swiss* data frame. Here is our model:

```
# Fit model
linear_brms_2 <- brm(Examination ~ Education +
                     Agriculture + Catholic + Infant.Mortality,
                     data = swiss,
                     family = gaussian(link = "identity"),
                     refresh = 0)
```

Remember from Chapter 9 that we can create an object summarizing our estimation results like this:

```
# Create summary object
linear_brms_2_sum <- summary(linear_brms_2)

# Create summary data frame
linear_brms_2_sum_df <- data.frame(linear_brms_2_sum$fixed)

# Show data frame
linear_brms_2_sum_df
```

```
##                  Estimate Est.Error l.95..CI u.95..CI
## Intercept        18.738782   5.93976   7.1530 30.40377
## Education         0.423219   0.09460   0.2391  0.61269
## Agriculture      -0.067560   0.04277  -0.1505  0.01501
## Catholic         -0.079991   0.01797  -0.1152 -0.04502
## Infant.Mortality -0.008992   0.23439  -0.4793  0.44832
##                   Rhat Bulk_ESS Tail_ESS
## Intercept        1.001     2976     2905
## Education        1.001     3106     2906
## Agriculture      1.001     2720     2574
## Catholic         1.002     3730     2940
## Infant.Mortality 1.000     4033     2958
```

We want to use *ggplot2* to create credibility intervals for each variable with **l.95..CI** as the minimum value and **u.95..CI** as the maximum value. These are the lower and upper bounds of the middle 95 percent of the estimates' marginal posterior distributions, i.e. the 95 percent credibility intervals.[25] We

[25] The procedures used here are also generally applicable for graphing frequentist confidence

will also create a point at the **mean** of each estimate. To do this, we will use *ggplot2*'s `geom_pointrange` function.

First we need to do a little tidying up.

```
# Convert row.names to column
linear_brms_2_sum_df$Variable <- row.names(linear_brms_2_sum_df)

# Keep only coefficient estimates
## This allows for a more interpretable scale
linear_brms_2_sum_df <- subset(linear_brms_2_sum_df,
                               Variable != "Intercept")
```

The first line of executable code creates a proper variable out of the data frame's row.names attribute. In this case, `row.names` contains the names of the variables included in the regression. The second executable line removes the *Intercept* estimates. This allows the variable's coefficient estimates to be plotted on a scale that enables easier interpretation.

Now we can create our caterpillar plot (Figure 10.5).

```
library(dplyr)

# Rename variables to make them easier for ggplot2 to work with
linear_brms_2_sum_df <- rename(linear_brms_2_sum_df,
                               lower = `l.95..CI`)
linear_brms_2_sum_df <- rename(linear_brms_2_sum_df,
                               upper = `u.95..CI`)

# Make caterpillar plot
ggplot(data = linear_brms_2_sum_df,
       aes(x = reorder(Variable, lower),
           y = Estimate,
           ymin = lower, ymax = upper)) +
                    geom_pointrange(size = 1.4) +
                    geom_hline(yintercept = 0,
                               linetype = "dotted") +
                    xlab("Variable") +
                    ylab("Coefficient Estimate") +
                    coord_flip() + theme_bw(base_size = 20)
```

There are some new pieces of code in here, so let's take a look. First, the

intervals once you have calculated the confidence intervals. One useful function for doing this is `confint`.

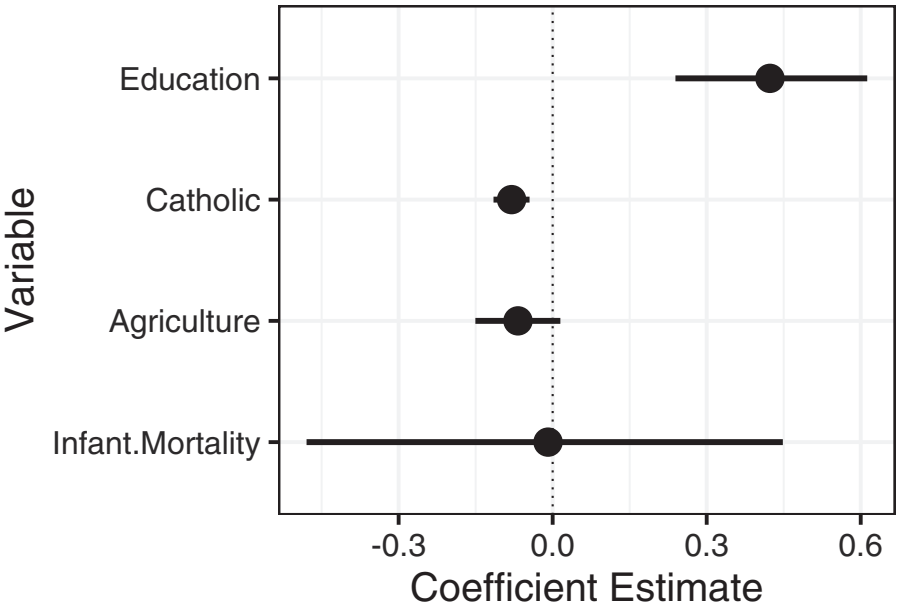


FIGURE 10.5: Example Caterpiller Plot

data frame is reordered from the highest to lowest value of **l.95..CI** using the `reorder()` function. We renamed this column "lower" so that it would be easier to work with in `ggplot()`. Reordering by these values makes the plot easier to read. The middle point of the point range is set with `y` and the lower and upper bounds with `ymin` and `ymax`. The `geom_hline()` function used here creates a dotted horizontal line at 0, i.e. no effect. `coord_flip()` flips the plot's coordinates so that the variable names are on the y-axis. We can include this plot in a knitted document the same way as before.

Note that we create this example to help you understand the power of *ggplot2* to create new graphics from complex objects. This particular task—creating caterpillar plots from *brms* objects—has been packaged into the `stanplot()` function that comes with *brms*.

10.5 JavaScript Graphs with *googleVis*

Markus Gesmann and Diego de Castillo's (2019) *googleVis* package allows us to use Google's Visualization API from within R to create interactive tables, plots, and maps with Google Chart Tools. Because the visualizations are

written in JavaScript, they can be included in HTML presentation documents created by R Markdown. Unfortunately, they cannot be directly[26] included in LaTeX-produced PDFs. The *animation* package (Xie, 2018) does have some limited features for including interactive visualizations in PDFs (as well as HTML documents) and is worth investigating if you want to do this. The *gganimate* package allows you to annimate *ggplot2* graphics as GIFs. However, these cannot be included in PDFs.

10.5.1 Basic googleVis figures

Let's briefly look at how to make one type of figure with *googleVis*: a choropleth map. This is created with the `gvisGeoChart()` function. We will use this example to illustrate how to incorporate *googleVis* figures into R Markdown.[27]

Imagine that we want to map global fertilizer consumption in 2011 using the World Bank data we gathered in Chapter 6. Remember that the data was highly right skewed, so we will actually map the natural logarithm of the **fert_cons** variable.[28] Assuming that we have already loaded the *main-data* data set, here is the code:

```
# Load googleVis
library(googleVis)

# Subset main_data so that it only includes 2011
data_sub <- subset(main_data, year == 2011)

# Keep values of fert_cons greater-than 0.1 data_sub <-
subset(data_sub, fert_cons > 0.1)

# Find the natural logarithm of fert_cons
## Round the results to one decimal digit.
data_sub$fert_cons_log <- round(log(data_sub$fert_cons),
                                digits = 1)

# Make a map of Fertilizer Consumption
fc_map <- gvisGeoChart(data = data_sub,
                       locationvar = "iso2c",
                       colorvar = "LogConsumption",
```

[26] The example in this chapter is from a screenshot.

[27] For demonstrations of the full range of plotting functions available, visit the *googleVis* website: `http://code.google.com/p/google-motion-charts-with-r/wiki/GadgetExamples#googleVis_Examples`.

[28] You'll notice in the code below that we remove all values of **fert_cons** less than 0.1. This is so that we can calculate integer values with the natural logarithm.

```
                        options = list(
                             colors = "['#ECE7F2', '#A6BDDB',
                                        '#2B8CBE']",
                             width = "780px", height = "500px")
)
```

The `locationvar` argument specifies the variable with information on each observation's location. Google Chart Tools can use ISO two-letter country codes to determine each country's location. `colorvar` specifies the variable with the values to map for each country. We can determine other options by creating a list-type object with arguments specifying characteristics such as the map's width, height, and colors. The colors here are written using hexadecimal values. This is a commonly used format for specifying colors on websites.[29]

To view the figure on your computer, use *googleVis*'s `plot()` function. For example, to view our map, type:

```
plot(fc_map)
```

Note that you need to be connected to the internet to view figures created by *googleVis*; otherwise, your image will not be able to access the required JavaScript files from the Google Visualization API.

10.5.2 Including *googleVis* in knitted documents

Typing `print(fc_map, tag = "chart")` in a knittable document would print the entire JavaScript code needed to create the map. Much like we saw with tables produced with *xtable* and *texreg* in Chapter 9, we need to change the code chunk `results` option to include the map as a map rather than as JavaScript markup. To have the visualization show up in your HTML output, rather than the code block, set the code chunk option to `results='asis'`.[30] For example, the full code needed to create and print *fc_map* is available at: `https://bit.ly/2CfXWOs`.[31] To knit the map into an R Markdown document, type:

[29] You can also use hexadecimal values in *ggplot2*. The Color Brewer 2 website (`http://colorbrewer2.org/`) is very helpful for picking hexadecimal color palettes, among others.

[30] You can use `results='asis'` to include almost any type of JavaScript graphics. For an example using the D3 JavaScript library and *knitr* see this page by Yihui Xie: `http://yihui.name/knitr/demo/javascript/`.

[31] The full URL is: `https://raw.githubusercontent.com/christophergandrud/Rep-Res-Examples/master/Graphs/GoogleVisMap.R`.

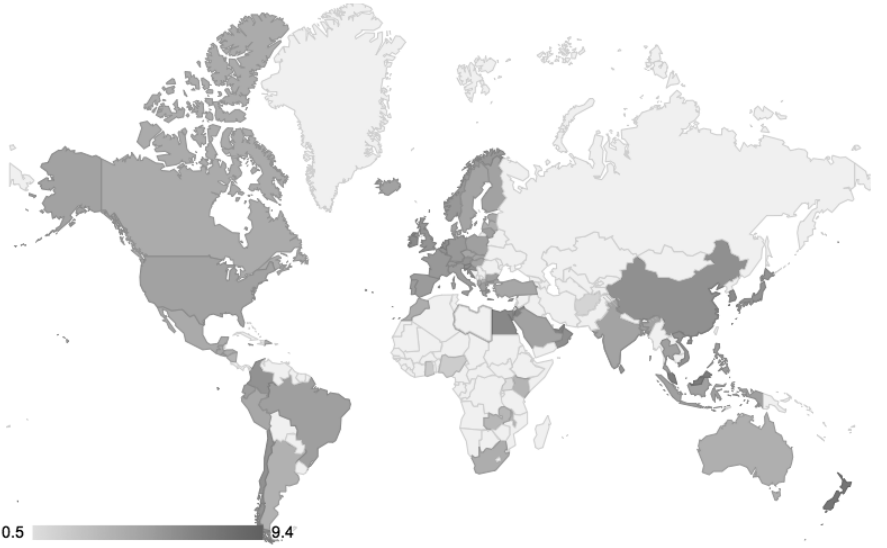


FIGURE 10.6: Screenshot of a *googleVis* Geo Chart

````
```{r}, echo=FALSE, message=FALSE, results='asis'}
 # Create and print geo map
devtools::source_url("https://bit.ly/2CfXWOs")
```
````

10.5.3 JavaScript Graphs with *htmlwidgets*-based packages

The number of tools for creating JavaScript graphs from R that can be knitted into HTML files is growing rapidly. The *htmlwidgets* (Vaidyanathan et al., 2019) framework is especially making the development of these tools easier. There are tools built on *htmlwidgets* for creating maps, network graphs, time series graphs, and interactive tables, among others. Though the syntax of each of these tools differs, they can all easily be included into R Markdown documents. Often you run their core functions in a code chunk, without needing to use an additional call to `print` or `plot`.

Chapter summary

In this chapter we have learned how to take results from our statistical analyses and other information from our data and dynamically present them in figures. In the next chapters, we will learn the details of how to create the LaTeX and

Markdown presentation documents we use to present the tables we created in Chapter 9 and the figures we created in this chapter.

Part IV

Presentation Documents

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

11

Presenting with LaTeX

We have already begun to see how LaTeX works for presenting research results. This chapter gives you a more detailed and comprehensive introduction to basic LaTeX document structures and commands. It is not a complete introduction to all that LaTeX is capable of, but we will cover enough that you will be able to create an entire well-formatted article and slideshow with LaTeX that you can use to dynamically present your results.

For basic LaTeX documents, such as short articles or simple presentations, it may often be quicker and simpler to write the markup using an R Markdown document and compile it to PDF with the *rmarkdown* package. Markdown syntax is much simpler than normal LaTeX. However, there are at least two reasons why it is useful to become familiar with LaTeX syntax. First, understanding LaTeX syntax will help you debug issues you might encounter when using *rmarkdown* with LaTeX that would otherwise be mysterious if you were only familiar with Markdown. Second, R Markdown has limited capabilities for creating more complex documents such as books and documents with highly customizable formatting needs.[1] Using *knitr* LaTeX or including LaTeX syntax directly in R Markdown documents can be useful in these situations.

In this chapter we will learn about basic LaTeX document structures and syntax as well as how to dynamically create LaTeX bibliographies with BibTeX, R, and *knitr*. Finally, we will look at how to create PDF beamer slideshows.

Note: This chapter and the following chapter are unusual for this book in that they do not refer to knitr and R Markdown interchangeably. Remember you can almost always include LaTeX syntax in an R Markdown document, though typically this will only impact the document when it is compiled to PDF.

[1] The *bookdown* (Xie, 2020a) R package greatly improved the ability to create book-like documents with R Markdown. The third edition of this book is made with *bookdown*.

11.1 The Basics

In this section we look at how to create a LaTeX article including what editor programs to use, the basic structure of a LaTeX document, including preamble and body, LaTeX syntax for creating headings, paragraphs, lines, text formatting, math, lists, footnotes, and cross-references. I will assume that you already have a fully functioning TeX distribution installed on your computer. See Section 1.5.1 for information on how to install TeX.

11.1.1 Getting started with LaTeX editors

RStudio is a fully functional LaTeX editor in addition to being an integrated development environment for R. If you want to create a new LaTeX document, you can click `File` in the menu bar, then `New File` and `R Sweave`.

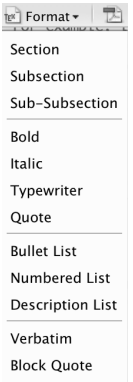


FIGURE 11.1: RStudio TeX Format Options

Remember from Chapter 3 that R Sweave files are basically LaTeX files that can include *knitr* code chunks. You can use RStudio to knit and compile a document with the click of one button: `Compile PDF`. You can use this button to compile R Sweave files like regular LaTeX files in RStudio even if they do not have code chunks. If you use another program to compile them, you might need to change the file extension from *.Rnw* to *.tex*. You can also insert many of the items we will cover in this section into your documents with RStudio's LaTeX `TeX Format` button. See the figure above.

There are many other LaTeX editors[2] and many text editors that can be

[2]Wikipedia has collated a table that comprehensively compares many of these editors: `https://en.wikipedia.org/wiki/List_of_text_editors`.

modified to compile LaTeX documents. For example, alongside writing this book in RStudio, I typed much of the LaTeX markup in the Atom[3] text editor because it was easier to work with a large number of files simultaneously. However, RStudio has by far the best integration with *knitr*.

11.1.2 Basic LaTeX command syntax

As you probably noticed in Part III's examples, LaTeX commands start with a backslash (`\`). For example, to create a section heading you use the `\section` command. The arguments for LaTeX commands are written inside of curly braces (`{}`) like this:

```
\section{My Section Name}
```

Probably one of the biggest sources of errors that occur when compiling a LaTeX document to PDF are caused by curly brackets that aren't closed, i.e. an open bracket (`{`) is not matched with a subsequent closed bracket (`}`). Watch out for this and use an editor (like RStudio) that highlights brackets' matching pairs. As we will see, unlike in R with parentheses, if your LaTeX command does not have an argument, you do not need to include the curly brackets at all.

There are a number of places to find comprehensive lists of LaTeX commands. The Netherlands TeX users group has compiled one: `http://www.ntg.nl/doc/biemesderfer/ltxcrib.pdf`.

11.1.3 The LaTeX preamble and body

All LaTeX documents require a preamble. The preamble goes at the very beginning of the document. The preamble usually starts with the `documentclass` command. This specifies what type of presentation document you are creating, e.g. an article, a book, a slideshow,[4] and so on. LaTeX refers to these as classes. Classes specify a document's formatting. You can add options to `documentclass` to change the format of the entire document. For example, if we wanted to create an article class document with two columns, we would type:

```
\documentclass[twocolumn]{article}
```

[3] `https://atom.io/`

[4] "Slideshow" is not a valid class. One slideshow class that we discuss later is called "beamer".

In the preamble you can also specify other style options and load extra packages. The command to load a package in LaTeX is `\usepackage`. For example, if you include `\usepackage{url}` in the preamble of your document, you will be able to specify URL links in the body with the command `\url{SOMEURL}`.

The preamble is often followed by the body of your document. It is specified with the `body` environment. See Chapter 9 for more details about LaTeX environments. You tell LaTeX where the body of your document starts by typing `\begin{document}`. The very last line of your document is usually `\end{document}`, indicating that your document has ended. When you open a new R Sweave file in RStudio, it creates an article class document with a very simple preamble and body like this:

```
\documentclass{article}

\begin{document}
\SweaveOpts{concordance=TRUE}




\end{document}
```

This is all you need to get a very basic article class document working.[5] If you want the document to be of another class, change `article` to something else, a `book` for example.

Let's begin to modify the markup. First we will include in the preamble the (`hyperref`) for clickable hyperlinks and `natbib` for bibliography formatting. We will discuss `natbib` in more detail below. Note that in general, and unlike in R, almost all of the LaTeX packages you will use are installed on your computer when you installed the TeX distribution.

Next, it's often a good idea to include *knitr* code chunks that specify features of the document as a whole. These can include global chunk options as well as loading data and packages used throughout the document.

Then you likely want to specify title information just after the `document` environment begins. Use the `title` command to add a title, the `author` command to add author information, and `date` to specify the date.[6] Then include the `maketitle` command. This will place your title and author information in the body of the document. If you are writing an article you may also want to

[5] `\SweaveOpts{concordance=TRUE}` maps the line numbers in your `.Rnw` file to the `.tex` file it generates. This especially helps with debugging. See: `https://support.rstudio.com/hc/en-us/articles/200532247-Weaving-Rnw-Files` (accessed 3 October 2019).

[6] In some document classes the current data will automatically be included if you don't specify the date.

follow `maketitle` with an abstract. Unsurprisingly, you can use the `abstract` environment to include this.

Here is a full LaTeX article class document with all of these changes added:

```
%%%%%%%%%%%%%%%% Article Preamble %%%%%%%%%%%%%%%%
\documentclass{article}

%% Load LaTeX packages
\usepackage{url}
\usepackage{hyperref}
\usepackage[authoryear]{natbib}

%% Set knitr global options and gather data
<<Global, include=FALSE>>=
#### Set chunk options ####
knitr::opts_chunk$set(fig.align='center')

#### Load and cite R packages ####
# Create list of packages
packages_used <- c("knitr", "ggplot2", "knitr")

# Load PackagesUsed and create .bib BibTeX file.
# Load packages
lapply(packages_used, library,
        character.only = TRUE)

# Create package BibTeX file
knitr::write_bib(packages_used, file = "packages.bib")

#### Gather Democracy data from Pemstein et al. (2010)
#### # For simplicity, store the URL in an object called 'url'.
url <- "http://www.unified-democracy-scores.org/files/20140312/
z/uds_summary.csv.gz"

# Create a temporary file called 'temp' to put the zip file into.
temp <- tempfile()

# Create a temporary file called 'temp' to put the zip file into.
temp <- tempfile()

# Download the compressed file into the temporary file.
download.file(url, temp)

# Decompress the file and convert it into a data frame
```

```
# class object called 'data'.
uds_data <- read.csv(gzfile(temp, "uds_summary.csv"))

# Delete the temporary file.
unlink(temp)
@

%% Start document body
\begin{document}
\SweaveOpts{concordance=TRUE}

%%%%%%%%%%%%% Create title %%%%%%%%%%%%%%%%%
\title{An Example knitr LaTeX Article}
\author{Christopher Gandrud \\
Zalando SE\thanks{Email: \href{mailto:christopher.gandrud@zalando
.de}{christopher.gandrud@zalando.de}}}
\date{August 2019}
\maketitle

%%%%%%%%%%%%% Abstract %%%%%%%%%%%%%%%%%%%%%
\begin{abstract}
  Here is an example of a knittable article class LaTeX
  document.
\end{abstract}

%%%%%%%%%%%% Article Main Text %%%%%%%%%%%%%
\section{The Graph}

I gathered data from \cite{Pemstein2010} on countries' democracy
level. They call their democracy measure the Unified Democracy
Score (UDS). Figure \ref{DemPlot} shows the mean UDS scores
over time for all of the countries in their sample.

\begin{figure}
  \caption{Mean UDS Scores}
  \label{DemPlot}
<<echo=FALSE, message=FALSE, warning=FALSE, out.width='7cm',
out.height='7cm'>>=
# Graph UDS scores
    ggplot(uds_data, aes(x = year, y = mean)) +
        geom_point(alpha = I(0.1)) +
        stat_smooth(size = 2) +
        ylab("Democracy Score") + xlab("") +
        theme_bw()
```

```
@
\end{figure}

%%%%%%%%%%% Reproducing the Document %%%%%
\section*{Appendix: Reproducing the Document}

This document was created using the R version
\Sexpr{paste0(version$major, ".",
version$minor)} and the R package \emph{knitr}
\citep{R-knitr}.

It also relied on the R packages \emph{ggplot2} and
\citep{R-ggplot2}. The document can be completely
reproduced from source files available on GitHub
at: \url{https://github.com/christophergandrud/
rep-res-book-v3-examples}.

%%%%%%%%% Bibliography %%%%%%%%%%%%%%%%%%%%%
\bibliographystyle{apa}
\bibliography{main.bib, packages.bib}
\end{document}
```

The *knitr* code chunk syntax should be familiar to you from previous chapters, so let's unpack the LaTeX syntax from just after the first code chunk, including the "Create title" and "Abstract" parts. New syntax shown in later parts of this example is discussed in the remainder of this section and the next section on bibliographies.

First, remember that the percent sign (%) is LaTeX's comment character. Using it to comment your markup can make it easier to read. Second, as we saw in Chapter 9, double backslashes (\\), like those after the author's name, force a new line in LaTeX. We will discuss the `emph` command in a moment. Third, using the `thanks` command allows us to create a footnote for author contact information[7] that is not numbered like the other footnotes (see below). Finally, you'll notice `\href{mailto: . . . .org}}`. This creates an email address in the final document that will open the reader's default email program when clicked.

You may have noticed the following line:

```
\Sexpr{paste0(version$major, ".", version$minor)}
```

This code finds the current version of R being used and prints the version number into the presentation document.

[7] Frequently it also includes thank yous to people who have helped with the research.

11.1.4 Headings

Earlier in the chapter, we briefly saw how to create section-level headings with `section`. There are a number of other sub section-level headings including `subsection`, `subsubsection`, `paragraph`, and `subparagraph`. Headers are numbered automatically by LaTeX.[8] To have an unnumbered section, place an asterisk in it like this: `\section*{UNNUMBERED_SECTION}`. In book class documents, you can also use `chapter` to create new chapters and `part` for collections of chapters.

11.1.5 Paragraphs and spacing

In LaTeX, paragraphs are created by adding a blank line between lines. It will format all of the tabs for the beginning of paragraphs based on the document's class rules. As we discussed before, writing tabs in the markup version of your document does nothing in the compiled document. They are generally used just to make the markup easier for people to read.

Note that adding more blank lines between paragraphs will not add extra space between the paragraphs in the final document. To specify the space following paragraphs (or almost any line) use the `vspace` (vertical space) command. For example, to add three centimeters of vertical space on a page type: `\vspace{3cm}`.

Similarly, adding extra spaces between words in your LaTeX markup won't create extra spaces between words in the compiled document. To add horizontal space use the `hspace` command in the same way as `vspace`.

11.1.6 Horizontal lines

Use the `hrulefill` command to create horizontal lines in the text of your document. For example, `\hrulefill` creates:

Inside of a `tabular` environment, use the `hline` command rather than `hrulefill`.

[8] The `paragraph` level does not have numbers.

TABLE 11.1: LaTeX Font Size Commands

11.1.7 Text formatting

Let's briefly look at how to do some of the more common types of text formatting in LaTeX and how to create some commonly used diacritics and special characters.

Italics and Bold

To italicize a word in LaTeX, use the `emph` (emphasis) command. For bold, use `textbf`. You can nest commands inside of one another to combine their effect. For example, to ***italicize and bold*** a word, use: `\emph{textbf{italicize and bold}}`.

Font size

You can specify the base font size of an entire document with a `documentclass` option. For example, to create an article with 12-point font, use: `\documentclass[12pt]{article}`.

There are a number of commands to set the size of specific pieces of text relative to the base size. See the following table for the full list. Usually a slightly different syntax is used for these commands that goes like this: `{\SIZE_COMMAND . . . }`. For example, to use the tiny size in your text use: `{\tiny{tiny size}}`.

You can change the size of code chunks that *knitr* places in presentation documents using these commands. Just place the code chunk inside of `{\SIZE_COMMAND . . . }`. This is similar to using the `size` code chunk option.

Diacritics

You cannot directly enter letters with diacritics—e.g. accent mark—into LaTeX. For example, to create a letter c with a cedilla (ç) you need to type `\c{c}`. To create an 'a' with an acute accent (á), type: `\'{a}`. There are obviously many types of diacritics and commands to include them within LaTeX-produced documents. For a comprehensive discussion of the issue and a list of commands see the LaTeX Wikibook page on the topic: `https://en.wikibooks.org/wiki/LaTeX/Special_Characters`. If you regularly use non-English alphabets, you might also be interested in reading the LaTeX Wikibook page on internationalization: `https://en.wikibooks.org/wiki/LaTeX/Internationalization`.

Quotation marks

To specify double left quotation marks ("), use two back ticks (``` `` ```). For double right quotes (”), use two apostrophes (`''`). Single quotes follow the same format (`` `' ``).

11.1.8 Math

LaTeX is particularly popular among quantitative researchers and mathematicians because it is very good at rendering mathematical notation. A complete listing of every math command would take up quite a bit of space.[9] I am briefly going to discuss how to include math in a LaTeX document. This discussion includes a few math syntax examples.

To include math inline with your text, place the math syntax in between backslashes and parentheses, i.e. `\( . . . \)`. For example, `\( s^{2} = \frac{\sum(x - \bar{x})^2}{n - 1} \)` produces $s^2 = \frac{\sum(x-\bar{x})^2}{n-1}$ in our final document.[10] We can display math separately from the text by placing the math commands inside of backslashes and square brackets: `\[ . . . \]`.[11] For example,

```
\[
  s^{2} = \frac{\sum(x - \bar{x})^2}{n - 1}
\]
```

[9] See the Netherlands TeX user group list mentioned earlier for an extensive compilation of math commands.

[10] Instead of backslashes and parentheses, you can also use a pair of dollar signs (`$. . .$`).

[11] Equivalently, use two pairs of dollar signs (`$$...$$`) or the `display` environment. Though it will still work in most cases, the double dollar sign math syntax may cause errors. You can also number display equations using the `equation` environment.

gives us:

$$s^2 = \frac{\sum(x - \bar{x})^2}{n - 1}$$

11.1.9 Lists

To create bullet lists in LaTeX, use the `itemize` environment. Each list item is delimited with the `item` command. For example:

```
\begin{itemize}
    \item The first item.
    \item The second item.
    \item The third item.
\end{itemize}
```

gives us:

- The first item.
- The second item.
- The third item.

To create a numbered list, use the `enumerate` environment instead of `itemize`.

You can create sublists by nesting lists inside of lists like this:

```
\begin{itemize}
    \item The first item.
    \item The second item.
    \begin{itemize}
        \item A sublist item
    \end{itemize}
    \item The third item.
\end{itemize}
```

which gives us:

- The first item.
- The second item.

  - A sublist item
- The third item.

11.1.10 Footnotes

To create plain, non-bibliographic footnotes, place `\footnote{` where you would like the footnote number to appear in the text. Then type the footnote's text. Remember to close the footnote with a `}`. LaTeX does the rest, including formatting and numbering.

11.1.11 Cross-references

LaTeX will also automatically format cross-references. We were already partially introduced to cross-references in Chapters 9 and 10. At the place where you would like to reference, add a `label` such as `\label{ACrossRefLabel}`. It doesn't really matter what label name you choose, though make sure they are not duplicated in the document. Then place a `ref` command (e.g. `\ref{ACrossRefLabel}`) at the place in the text where you want the cross-reference to be.

If you place the `label` on the same line as a heading command, `ref` will place the heading number. If `label` is in a `table` or `figure` environment, you will get the table or figure number. You can also use `pageref` instead of `ref` to include the page number. Finally, loading the *hyperref* package makes cross-references (or footnote) clickable. Clicking on them will take you to the items they refer to.

11.2 Bibliographies with BibTeX

LaTeX can take advantage of very comprehensive bibliography-making capabilities. All major TeX distributions come with BibTeX. BibTeX is basically a tool for creating databases of citation information. In this section, we are going to see how to incorporate a BibTeX bibliography into your LaTeX documents. Then we will learn how to use R to automatically generate a bibliography of packages used to create a knitted document. For more information on BibTeX syntax, see the LaTeX Wikibook page on Bibliography management: `https://en.wikibooks.org/wiki/LaTeX/Bibliography_Management`.

11.2.1 The *.bib* file

BibTeX bibliographies are stored in plain-text files with the extension `.bib`. These files are databases of citations.[12] The syntax for each citation goes like this:

```
@DOCUMENT_TYPE{CITE_KEY,
    title = {TITLE},
    author = {AUTHOR},
    . . . = {. . .}
}
```

`DOCUMENT\_TYPE` specifies what type of document—article, book, webpage, and so on—the citation is for. This determines what items the citation can and needs to include. Then we have the `CITE_KEY`. This is the reference's label that you will use to include the citation in your presentation documents. We'll look more at this later in the section. Each citation must have a unique `CITE_KEY`. A common way to write these keys is to use the author's surname and the publication year, e.g. `donoho2009`. The cite key is followed by the other citation attributes such as `author`, `title`, and `year`. These attributes all follow the same syntax: `ATTRIBUTE = {. . .}`.

It's worth taking a moment to discuss the syntax for the BibTeX author attribute. First, multiple author names are separated by `and`. Second, BibTeX assumes that the last word for each author is their surname. If you would like multiple words to be taken as the "surname", then enclose these words in curly brackets. If we wanted to cite the World Bank as an author, we write `{World Bank}`; otherwise, it will be formatted "Bank, World" in the presentation document.

Here is a complete BibTeX entry for Donoho et al. (2009):

```
@article{donoho2009,
  author = {David L Donoho and Arian Maleki and Morteza
  Shahram and Inam Ur Rahman and Victoria Stodden},
  title = {Reproducible research in computational harmonic
    analysis},
  journal = {Computing in Science & Engineering},
  year = {2009},
  volume = {11},
  number = {1},
  pages = {8--18}
}
```

[12] The order of the citations does not matter.

Each item of the entry must end in a comma, except the last one.[13]

11.2.2 Including citations in LaTeX documents

When you want to include citations from a BibTeX file in your LaTeX document, you first use the `bibliography` command. For example, if the BibTeX file is called *ain.bib* and it is in the same directory as your markup document, then type: `\bibliography{ain.bib}`. You can use a bibliography stored in another directory; just include the appropriate file path information. Usually `bibliography` is placed right before `\end{document}` so that it appears at the end of the compiled presentation document.

You can also specify how you would like the references to be formatted using the `bibliographystyle` command. For example, this book uses the American Psychological Association (APA) style for references. To set this, I included `\bibliographystyle{apa}` directly before `bibliography`. The default style[14] is to number citations (e.g. `[1]`) rather than include author-year information[15] used by the APA. You will need to include the LaTeX package *natbib* in your preamble to be able to use author-year citation styles. This book includes `\usepackage[authoryear]{natbib}` in its preamble.

Place the `cite` command in your document's text where you want to place a reference. You include the `CITE_KEY` for the reference in this command, e.g. `\cite{donoho2009}`. You can include multiple citations in `cite`, just separate the `CITE_KEY`s with commas. You can add options such as the page numbers or other text to a citation using square brackets (`[]`). For example, if we wanted to cite the tenth page of Donoho et al. (2009), we type: `\cite[10]{donoho2009}`. The author-year style in-text citation that this produces looks like this: (Donoho et al., 2009, 10). You can add text at the beginning of a citation with another set of square brackets. Typing `\cite[see][10]{donoho2009}` gives us: (see Donoho et al., 2009, 10).

If you are using an author-year style, you can use a variety of *natbib* commands to change what information is included in the parentheses. Table 11.2 contains a selection of these commands and examples.

11.2.3 Generating a BibTeX file of R package citations

Researchers are pretty good about citing others' articles and data. However, citations of R packages used in analyses is very inconsistent. This is unfortunate

[13] This is very similar to how we create vectors in R, though in BibTeX you can actually have a comma after the last attribute.

[14] It is referred to in LaTeX as the plain style.

[15] This is sometimes referred to as the "Harvard" style.

TABLE 11.2: A Selection of *natbib* In-text Citation Style Commands

Command Example	Output
`\cite{donoho2009}`	Donoho et al. (2009)
`\citep{donoho2009}`	(Donoho et al., 2009)
`\citeauthor{donoho2009}`	Donoho et al.
`\citeyear{donoho2009}`	2009
`\citeyearpar{donoho2009}`	(2009)

not only because correct attribution is not being given to those who worked to create the packages, but also because it makes reproducibility harder. Not citing packages obscures important steps that were taken in the research process, primarily which package versions were used. Fortunately, there are R tools for quickly and dynamically generating package BibTeX files, including the versions of the packages you are using. They will automatically update the citations each time you compile your document to reflect any changes made to the packages.

You can automatically create citations for R packages using the `citation()` function inside of a code chunk. For example, if you want the citation information for the `knitr` package, you type:

```
citation("knitr")
```

```
##
## To cite the 'knitr' package in publications use:
##
##   Yihui Xie (2020). knitr: A General-Purpose
##   Package for Dynamic Report Generation in R. R
##   package version 1.27.
##
##   Yihui Xie (2015) Dynamic Documents with R and
##   knitr. 2nd edition. Chapman and Hall/CRC. ISBN
##   978-1498716963
##
##   Yihui Xie (2014) knitr: A Comprehensive Tool
##   for Reproducible Research in R. In Victoria
##   Stodden, Friedrich Leisch and Roger D. Peng,
##   editors, Implementing Reproducible
##   Computational Research. Chapman and Hall/CRC.
##   ISBN 978-1466561595
##
## To see these entries in BibTeX format, use
## 'print(<citation>, bibtex=TRUE)', 'toBibtex(.)',
```

```
## or set 'options(citation.bibtex.max=999)'.
```

This gives you both the plain citation as well as the BibTeX version. If you only want the BibTeX version of the citation, use the `toBibtex()` function.

```
toBibtex(citation("knitr"))
```

The *knitr* package creates BibTeX bibliographies for R packages with the `write_bib()` function. Let's make a BibTeX file called *packages.bib* containing citation information for the *knitr* package.

```
# Create package BibTeX file
knitr::write_bib("knitr", file = "packages.bib")
```

`write_bib` automatically assigns each entry a cite key using the format `R-PACKAGE_NAME`, e.g. `R-knitr`.

Warning: *knitr*'s `write_bib()` function currently does not have the ability to append package citations to an existing file, but instead writes them to a new file. If there is already a file with the same name, it will overwrite the file. So, be very careful using this function to avoid accidental deletions. It is a good idea to have `write_bib()` always write to a file specifically for automatically generated package citations. You can include more than one bibliography in LaTeX's `bibliography` command. All you need to do is separate them with a comma.

```
\bibliography{main.bib, packages.bib}
```

We can use these techniques to automatically create a BibTeX file with citation information for all of the packages used in a research project. Simply make a character vector of the names of packages that you would like to include in your bibliography. Then run this through `write_bib()`.

You can make sure you are citing all of the key packages used in a knitted document by (a) creating a vector of all of the packages and then (b) using this in the following code to both load the packages and write the bibliography:

```
# Package list
packages_used <- c("ggplot2", "knitr", "xtable")

# Load packages
lapply(packages_used, library, character.only = TRUE)
```

```
# Create package BibTeX file
knitr::write_bib(packages_used, file = "packages.bib")
```

In the first executable line, we create our list of packages to load and cite. The next function is `lapply()` (list apply). This applies the function `library()` to all of the items in *packages_used*. `character.only = TRUE` is a `library` argument that allows us to use character string versions of the package names as R sees them in the *packages_used* vector, rather than as objects (how we have used `library` up until now). If you include these functions in a code chunk at the beginning of your knitted document, then you can be sure that you will have a BibTeX file with all of your packages.

11.3 Presentations with LaTeX Beamer

You can make slideshow presentations with LaTeX. Creating a presentation with a markup language can take a bit more effort than using a WYSIWYG program like Microsoft PowerPoint or Apple's Keynote. However, combining LaTeX and *knitr* can make fully reproducible presentations that dynamically create and present results. I have found this particularly useful in my teaching. Dynamically produced presentations allow me to provide my students with fully replicable examples of how I created a figure on a slide. *knitr* also makes it easy to beautifully present code examples.

One of the most popular LaTeX tools for slideshows is the beamer class. When you compile a beamer class document, a PDF will be created where every page is a different slide. All major PDF viewer programs have some sort of "View Full Screen" option to view beamer PDFs as full screen slideshows. Usually you can navigate through the slides with the forward and back arrows on the keyboard.

In this section we will take a brief look at the basics of creating slideshows with beamer, highlighting special considerations that need to be made when working with beamer and *knitr*. In Chapter 12 we will see how to use the *rmarkdown* package to create beamer presentations with the much simpler Markdown syntax.

11.3.1 Beamer basics

knitr largely works the same way in LaTeX slideshows as it does in article or book class documents. Nonetheless, there are a few differences to look out for.

Confidence Intervals in R for Objects of Class `lm`

To calculate the confidence intervals for regression coefficients in R use the `confint` command.

```
# Find confidence intervals
CI <- confint(M1)
CI

##                2.5 % 97.5 %
## (Intercept)  24.091 47.379
## duration     -1.124  2.484
## condused    -10.652  9.263
## wheels        5.122 15.788
```


Graphically

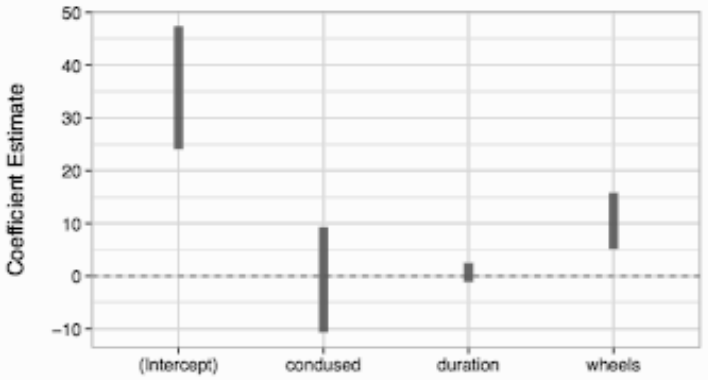


FIGURE 11.2: Knitted Beamer PDF Example

The Beamer preamble

You use `documentclass` to set a LaTeX document as a `beamer` slideshow. You can also include global style information in the preamble by using the commands `usetheme`, `usecolortheme`, `useinnertheme`, `useoutertheme`. For a fairly comprehensive compilation of beamer themes, see the Hartwork's Beamer theme matrix: `https://hartwork.org/beamer-theme-matrix/`.

Slide frames

After the preamble, you start your document as usual by beginning the `document` environment. Then you need to start creating slides. Individual beamer slides are created using the `frame` environments. Create a frame title using `frametitle`.

```
\frame{
    \frametitle{An example frame}


}
```

Note that you can also use the usual `\begin{frame} . . \end{frame}` syntax. Unlike in a WYSIWYG slide show program, you will not be able to tell if you have tried to put more information on one slide than it can handle until after you compile the document.[16]

Title frames

One important difference from a regular LaTeX article is that instead of using `maketitle` to place your title information, in beamer you place the `titlepage` inside of a frame by itself.

Sections and outlines

We can use section commands in much the same way as we do in other types of LaTeX documents. Section commands do not need to be placed inside of frames. After the title slide, many slideshows have a presentation outline. You can automatically create one from your section headings using the `tableofcontents` command. Like the `titlepage` command, `tableofcontents` can go on its own frame:

[16] One way to deal with frames that span multiple slides is to use the `allowframebreaks` command, `\begin{frame}[allowframebreaks]`.

```
%%% Title slide
\frame{
    \titlepage
}

%% Table of contents slide
\frame{
    \frametitle{Outline}
        \tableofcontents
}
```

Make list items appear

Lists work the same way in beamer as they do in other LaTeX document classes. They do have an added feature in that you can have each item appear as you progress through the slide show. After `\item`, place the number of the order in which the item should appear. Enclose the number in `< ->`. For example,

```
\begin{itemize}
    \item<1-> The first item.
    \item<2-> The second item.
    \item<2-> The third item.
\end{itemize}
```

In this example the first item will appear before the next two. These two will appear at the same time.

11.3.2 *knitr* with LaTeX slideshows

knitr code chunks have the same syntax in LaTeX slideshows as in other LaTeX documents. You do need to make one change to the `frame` options, however, to include highlighted *knitr* code chunks on your slides. You should add the `fragile` option to the `frame` command.[17] Here is an example:

```
\begin{frame}[fragile]
    \frametitle{An example fragile frame.}
```

[17] For a detailed discussion of why you need to use the `fragile` option with the `verbatim` environment that *knitr* uses to display highlighted text in LaTeX documents, see this blog post by Pieter Belmans: `https://pbelmans.ncag.info/blog/2011/02/20/why-latex-beamer-needs-fragile-when-using-verbatim/` (posted 20 February 2011).

```
\end{frame}
```

Chapter summary

In this chapter we have learned the nitty-gritty of how to create simple LaTeX documents, articles and slideshows, that we can embed our reproducible research in using *knitr*. In the next chapter we look at how to use R Markdown to expand the type of presentation documents we can create reproducibly.

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

12

Presenting in a Variety of Formats with R Markdown

While Markdown started as a simple way to write HTML documents for the web, R Markdown (and the programs it relies on in the background, particularly Pandoc) dramatically expands our ability to take advantage of simple markdown syntax for creating documents in many formats.

In this chapter we will learn about Markdown editors and the basic Markdown syntax for creating simple reproducible documents, including many of the things we covered for *knitr*/LaTeX documents such as headings and text formatting. Please refer back to previous chapters for syntax used to display code and code chunks (Chapter 8), tables (Chapter 9), and figures (Chapter 10) with R Markdown documents. In this chapter we will also briefly look at some more advanced features for including math with MathJax, footnotes and bibliographies with Pandoc, and customizing styles with CSS. Then we will learn how to create slideshows. We'll finish up the chapter by looking at options for publishing Markdown-created HTML documents, including locally on your computer and GitHub Pages.

12.1 The Basics

Markdown was created specifically to make it easy to write HTML (or XHTML[1]) using a syntax that is human readable and possibly publishable without compiling. For example, compare the Markdown table syntax in Chapter 9 to the HTML syntax for virtually the same table.[2] That being said, to make Markdown simple, it does not have as many capabilities as HTML. To get around this problem, you can still use HTML in Markdown, though note that Markdown syntax cannot be used between HTML element tags.

[1] Extensible HyperText Markup Language.

[2] For more information, see John Gruber's website: `https://daringfireball.net/projects/markdown/`.

Pandoc and R Markdown extend Markdown so that it can be used to create reproducible PDF and MS Word documents.

Note: If you are using *rmarkdown* to compile a document to PDF or Word, using raw HTML syntax will often not work as intended, if at all. As a rule, syntax specific to LaTeX or HTML that is included in an R Markdown document can only be properly compiled to a PDF or HTML document, respectively. Similarly, you are only able to include graphics that are of types supported by the output format. You are not able to include a JavaScript plot directly in a PDF. R Markdown has been continuously improving its ability to interoperate between the different formats. For example, the `kable()` function creates tables without having to worry too much about the output format. The *knitr* code chunck option `fig.ext` (figure extension) allows you to more dynamically set the output format of a dynamically created figure so that it will be compilable to multiple formats.

12.1.1 Getting started with Markdown editors

RStudio functions as a very good editor for R Markdown documents and regular non-knittable Markdown documents as well. To create a new R Markdown document in RStudio, click `File` in the menu bar, then `New R Markdown`. You will then be able to select what output format you would like. RStudio has full syntax highlighting for code chunks and can compile *.Rmd* files into *.md*, then render them in *.html*, for example, with one click of the `Knit HTML` button. As we saw in Chapter 3, when you knit a Markdown document in RStudio, it will preview the HTML document for you. You can always view HTML documents by opening them with your web browser. You can do this directly from RStudio's **Preview HTML** window by clicking the `Open in Browser` button.

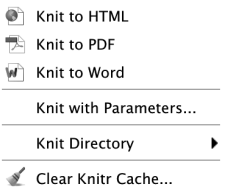


FIGURE 12.1: R Markdown Compile Dropdown Menu

If you click on the downward arrow next to `Knit HTML`, you will see the above dropdown menu. This allows you to also compile the document to PDF or MS Word, regardless of which format you originally chose when you created the document. As with HTML, you will be given a preview of the PDF or Word document when it is compiled.

Being plain-text, you can also use any other text editor to modify Markdown documents, though they will lack the level of integration with knitr/R Markdown that RStudio has.

12.1.2 Preamble and document structure

That was kind of a trick subsection title. Unlike LaTeX documents, plain Markdown documents do not have a preamble. R Markdown documents can have a header, basically another name for a preamble, but we will get to that later. There is also no need to start a body environment or anything like that. HTML head elements (HTML's preamble equivalent) are added automatically when you render Markdown documents into HTML. With Markdown, you can just start typing the content of your document.

Here is an example of an R Markdown document that creates the map we saw in Chapter 10. We'll go through all of the code below.

````
---
title: "Fertilizer Consumption"
author: "Christopher Gandrud"
date: "12/29/2018"
output: html_document
---

## Fertilizer Consumption (kilograms per hectare of arable land)
in 2003

Note: Data is from the [World Bank](https://data.worldbank.org/
indicator/AG.CON.FERT.ZS)

```{r CreategvisGeoMap, echo=FALSE, message=FALSE, results='asis'}
source("analysis/googlevis-map.R")
```
-----------

## R Session Info

```{r echo=FALSE}
sessionInfo()
```
````

12.1.3 Headings

Headings in Markdown are simple. Note that Markdown *headings* and R Markdown *headers* are not the same thing. The latter gives instructions for how to render the document, the former are section titles in the text. To create a line in the topmost heading style—maybe a title—just place one hash mark (`#`) at the beginning of the line. The second-tier heading gets two hashes (`##`) and so on. You can also put the hash mark(s) at the end of the heading, but this is not necessary. Here is an example of the three headings:

```
# A level one heading

## A level two heading

### A level three heading
```

There are six heading levels in Markdown. You can also create a level one heading by following a line of text with equal signs. Level two headings can be created by following a line of text with dashes:

```
A level one heading
===================

A level two heading
-------------------
```

12.1.4 Horizontal lines

If you would like to create horizontal lines that run the width of the page in Markdown, place three or more equal signs or dashes separated by text from above by one blank line:

```
Create a horizontal line.

=========
```

12.1.5 Paragraphs and new lines

Just like in LaTeX, new paragraphs are created by putting text on a new line separated from previous text with a blank line. For example:

```
This is the first paragraph.

This is the second paragraph.
```

Separating lines with a blank line places a blank line in the final document. End a line with two or more white spaces ( ) to create a new line that is not separated by a blank line.

12.1.6 Italics and bold

To *italicize* a word in Markdown, place it between two asterisks, e.g. `*italicize these words*`. To make words **bold**, place them between four asterisks, two on either side: `**make these words bold**`.

12.1.7 Links

To create hyperlinks in Markdown, use the `[LINK_TEXT](URL)` syntax.[3] `LINK_TEXT` is the text that you would like to show up as the hyperlink text. When you click on this text, it will take you to the linked site specified by `URL`. If you want to show only a URL as the text, type it in both the square brackets and parentheses. This is a little tedious, so in RStudio you can just type the URL and it will be hyperlinked. In regular Markdown, place the URL between less than and greater than signs (`<URL>`).

Special characters and font customization

Unlike LaTeX rendered with pdfLaTeX, Markdown can include almost any letters and characters included in your system. The main exceptions are characters used by Markdown syntax (e.g. `*`, `#`, `\` and so on). You will have to escape these (see below). Font sizes and typefaces cannot be set directly with Markdown syntax. You need to set these with HTML or CSS, which I don't cover here, though below we will look at how to use a custom CSS file.

12.1.8 Lists

To create itemized lists in Markdown, place the items after one dash:

[3] You can also include a `title` attribute after the URL, though this is generally not very useful.

```
- Item 1
- Another item
- Item 3
```

To create a numbered list, use numbers and periods rather than dashes.

```
1. Item 1
2. Another item
3. Item 3
```

Escape characters

Markdown, like LaTeX and R, uses a backslash (\) as an escape character. For example, if you want to have an asterisk in the text of your document (rather than start to italicize your text, e.g. `*some italicized text*`), type: `\*`. Two characters—ampersand (`&`) and the less-than sign (`<`)—have special meanings in HTML.[4] So, to have them printed literally in your text, you have to use the HTML code for the characters. Ampersands are created with `&amp`. Less than signs are created with `&lt`.

12.1.9 Math with MathJax

Markdown by itself can't format mathematical equations. We can create LaTeX-style equations in HTML documents by adding on the MathJax JavaScript engine. MathJax syntax is the same as LaTeX syntax (see Section 11.1.8), especially when used from RStudio or when rendered with *rmarkdown*. Markdown-HTML documents rendered in RStudio automatically link to the MathJax engine online.[5] If you want to use another program to render Markdown documents with MathJax equations, you may need to take extra steps to link to MathJax. For more details, see `https://www.mathjax.org/#gettingstarted`.

Because backslashes are Markdown escape characters, in many Markdown editors you will have to use two backslashes to create math environments with MathJax. For example, in LaTeX and RStudio's Markdown, you can create a display equation like this:

$$s^2 = \frac{\sum(x - \bar{x})^2}{n - 1}$$

[4] Ampersands declare the beginning of a special HTML character. Less-than signs begin HTML tags.

[5] You will not be able to render equations when you are not online.

by typing:[6]

```
$$s^{2} = \frac{\sum(x - \bar{x})^2}{n - 1}$$
```

But, in other Markdown programs, you may have to use:

```
\\[
    s^{2} = \frac{\sum(x - \bar{x})^2}{n - 1}
\\]
```

To make inline equations, use parentheses instead of square brackets as in LaTeX, e.g. `\( s^{2} = \frac{\sum(x - \bar{x})^2}{n - 1} \)`. You can also use single dollar signs, e.g. `$ s^{2} = \frac{\sum(x - \bar{x})^2}{n - 1} $`

12.2 Further Customizability with *rmarkdown*

Markdown is simple and easy to use. But being simple means that it lacks important functionality for presenting research results, such as footnotes and bibliographies, and custom formatting. In this section we will learn how to overcome these limitations with Pandoc via the *rmarkdown* package.

More on *rmarkdown* Headers

In Chapter 3 we first saw an R Markdown header written in YAML. Just as a refresher, here is the basic header we looked at:

```
---
title: "A Basic PDF Presentation Document"
author: "Christopher Gandrud"
date: "29 August 2019"
output: pdf_document:
    toc: true
---
```

This header provides instructions for what to do when the document is rendered, gives instructions to render the document as a PDF (via LaTeX), and inserts a title, author, date, and table of contents at the beginning.

[6] In RStudio you can also use dollar signs to delimit MathJax equations as in LaTeX.

We also have the option to include other formatting options, many of which we would include in a *knitr* LaTeX document's preamble. You include these at the top level, i.e. without being tabbed. R Markdown refers to these options as "metadata". For example, to change the font size to 11 point we could use:

```
---
title: "A Basic PDF Presentation Document"
author: "Christopher Gandrud"
date: "30 November 2019"
output: pdf_document:
    toc: true
fontsize: 11pt
---
```

We could double-space the PDF document with a similar top-level entry: `linestretch: 2`.[7] To find more options for PDF documents, type `?pdf_document` into your R console. Note that these options will only affect your PDF document, not a rendered HTML file.

Remember from Chapter 3 that we can specify rendering instructions for multiple output formats in the same header. Here is a longer header, building on what we just saw:

```
---
title: "An Example rmarkdown Article"
author: "Christopher Gandrud"
date: "15 January 2019"
output:
    pdf_document:
        latex_engine: xelatex
        number_sections: yes
        toc: yes
    html_document:
        toc: no
        theme: "flatly"
linestretch: 2
fontsize: 11pt
bibliography:
    - main.bib
    - packages.bib
---
```

Ok, let's go through this in detail. We have already seen the `title`, `author`, `date`, `linestretch`, and `fontsize` options. Notice that we used `latex_engine`

[7] 1 would be for single space and 1.5 would be for one and a half spacing.

to set the LaTeX engine to XeLaTeX, which is useful for documents that include non-standard English characters. We also specified with `number_sections` that the PDF document should have numbered section headings.

For the HTML version of the document we do not want a table of contents as we set `toc: no`. We specified a CSS theme called Flatly for our HTML document using `theme: "flatly"`. As of this writing, *rmarkdown* has a built-in ability to use a range of themes from Bootswatch (`https://bootswatch.com/`). Alternatively, you can link to a custom CSS file with the `css` option. Use `html_document` to see other options. Notice that we can use `no` and `yes` instead of `false` and `true`, respectively.

We linked to two BibTeX files with the `bibliography` option. Using Pandoc syntax, the references will apply to both the PDF and HTML documents.

If you want to also enable the creation of a Microsoft Word document, include `output: word_document` in the header.

Bibliographies with Pandoc

Pandoc via *rmarkdown* allows us to insert citations from normal BibTeX files (see Chapter 11) specified in the header with `bibliography`. The main difference is that Pandoc has a different syntax from LaTeX for making in-text citations. Basic Pandoc citations begin with `@` followed by the BibTeX citation key. Square brackets (`[]`) create parentheses around the citation. Here is an example:

```
This is a citation \[@donoho2009].
```

Pandoc uses *natbib* by default, so the citation `[@donoho2009]` will appear as (Donoho et al., 2009). To add text before and after the citation inside of the parentheses, use something like this: `[see @donoho2009, 10]`; which creates: (see Donoho et al. 2009, 10). If you do not want the parentheses around the entire citation (only the year) then omit the square brackets. To include only the year, and not the authors' surnames, add a minus sign, e.g. `[-@donoho2009]`. See the table above for more options.

Full bibliographic information for each item that is cited in the text will be produced at the end of the output document. I suggest placing a heading like `# References` at the very end of your document so that the bibliography will be differentiated from the document's text.

TABLE 12.1: A Selection of Pandoc In-Text Citations

Markup	Result
`[@donoho2009]`	(Donoho 2009)
`[-@donoho2009]`	(2009)
`[see @donoho2009]`	(see Donoho 2009)
`[see @donoho2009, 10-11]`	(see Donoho 2009, 10–11)
`[@donoho2009; @Box1973]`	(Donoho 2009; Box 1973)
`@donoho2009 [10-11]`	Donoho (2009, 10–11)

Footnotes with Pandoc

You can also include footnotes in documents rendered with *rmarkdown* by using Pandoc's footnote syntax. In the text where you would like a footnote to be located, use: `[^NOTE_KEY]`. Then at the end of your document, place `[^NOTE_KEY]: The footnote text.`[8] `NOTE_KEY`s generally follow the same rules as BibTeX citation keys, so no spaces. The footnotes will be numbered sequentially when rendered.

To sum up, here is an example of a document that can be rendered in HTML or PDF using R Markdown. It includes footnotes and a bibliography.

````
---
title: "Minimal rmarkdown Example"
output:
    pdf_document:
        toc: true
    html_document:
        toc: false
bibliography: main.bib
---

This is some text.[^FirstNote]

This is a *knitr* code chunk:

```{r}
plot(cars$speed, cars$dist)
```
````

[8] You can actually put this almost anywhere and it will be placed and numbered correctly in the output document, but I find it easier to organize the footnotes when they are placed at the end.

```
This is a citation [see @donoho2009, 10].

[^FirstNote]: This is a footnote.

# References
```

We have only covered a small proportion of Pandoc's capabilities that you can take advantage of with *rmarkdown*. For full range of Pandoc's abilities, see `https://pandoc.org/MANUAL.html`.

12.2.1 CSS style files and Markdown

You can customize the formatting of HTML documents created with Markdown files using custom CSS style sheets. CSS files allow you to specify the way a rendered Markdown file looks in a web browser including fonts, margins, background color, and so on. We don't have space to cover CSS syntax here. There are numerous online resources for learning CSS. One of the best ways may be to just copy a CSS style sheet into a new file and play around with it to see how things change. A really good resource for this is Google Chrome's Developer Tools. The Developer Tools allows you to edit your webpages, including their CSS, and see a live preview. It is a really nice way to experiment with CSS (and HTML and JavaScript).[9] There are also numerous pre-made style sheets available online.[10]

Rendering R Markdown files to HTML using custom CSS

The simplest way to use a custom CSS style sheet is to include the file path to the CSS file in an *rmarkdown* header. As mentioned earlier, *rmarkdown* has a number of built-in CSS file options that you can access with `style`. If you want to use another custom CSS file, use the `css` option. If our custom CSS file is called *custom_style.css* in the same directory as the R Markdown document, then a basic header would be:

```
---
output:
    html_document:
```

[9] For more information on how to access and use Developer Tools in Chrome see: `https://developers.google.com/chrome-developer-tools/`.

[10] One small note: when you create a new style sheet or copy an old one, make sure the final line is blank. Otherwise you may get an "incomplete final line" error when you render the document.

```
        css: custom_style.css
---
```

If you are using the *knitr* package to render an R Markdown document to HTML, you can also include a custom CSS file. First use `knit` to knit the document to a plain Markdown file. Then use the `markdownToHTML()` function from the *markdown* package (Allaire et al., 2019a) to render the plain Markdown document in HTML, including the `stylesheet` argument with the path to the CSS file.

12.3 Slideshows with Markdown, R Markdown, and HTML

Because R Markdown documents can be compiled into HTML files, it is possible to use them to create HTML5 slideshows.[11] There are a number of advantages to creating HTML presentations with Markdown:

- You can use the relatively simple Markdown syntax.
- HTML presentations are a nice native way to show content on the web.
- HTML presentations can incorporate virtually any content that can be included in a webpage. This includes interactive content, like motion charts created by *googleVis* (see Chapter 10).

Let's look at how to create HTML slideshows from Markdown documents using (a) the *rmarkdown* package and (b) RStudio's built-in slideshow files, called R Presentations. You can also use *rmarkdown* to create beamer presentations.

HTML5 frameworks

Before getting into the details of how to use R Markdown for presentations and R Presentations, let's briefly look more into what an HTML5 slideshow is and the frameworks that make it possible. HTML5 slideshows rely on a number of web technologies in addition to HTML5, including CSS, and JavaScript to create a website that behaves like a LaTeX beamer or PowerPoint presentation. They run in your web browser and you may need to be connected

[11] The slideshows created by the tools in this section use features introduced in the 5th version of HTML, i.e. HTML5. In this section I often refer to HTML5 as just HTML for simplicity.

to the internet for them to work properly, as key components are often located remotely. Most browsers have `Full Screen` mode you can use to view presentations.

There are a number of different HTML5 slideshow frameworks that let you create and style your slideshows. In all of the frameworks, you view the slideshow in your web browser and advance through slides with the forward arrow key on your keyboard. You can go back with the back arrow. Despite these similarities, the frameworks have different looks and capabilities.

12.3.1 HTML slideshows with *rmarkdown*

It is very easy to create an HTML presentation using *rmarkdown* and the IO Slides[12] or Slidy[13] HTML5 frameworks. The syntax for IO Slides and Slidy presentations with *rmarkdown* presentations is almost exactly the same as the syntax we have seen throughout this chapter. There are two main differences from the syntax we have seen so far. First, `ioslides_presentation` for IO Slides or `slidy_presentation` for Slidy presentations is the output type to set in the header. Second, two hashes (`##`) set a frame's header.[14] For example,

```
---
title: "Simple rmarkdown Presentation Example"
author: "Christopher Gandrud"
date: "26 December 2015"
output:
    ioslides_presentation:
        incremental: true
---

## Access the code

The code to create the following figure is available online.
```

This code creates a slide show that begins with the slide in the following figure. Bullet points will be brought in incrementally because we used `incremental: true` under `output: ioslides_presentation`. Bullets are created using Markdown list syntax.

Use three dashes (`---`) to delineate a new slide without a header. You can style the presentation further using the `css` option in the header to link to a custom CSS file.

[12] `https://code.google.com/p/io-2012-slides/`
[13] `https://www.w3.org/Talks/Tools/Slidy2/#(1)`
[14] You can create sections with one hash.

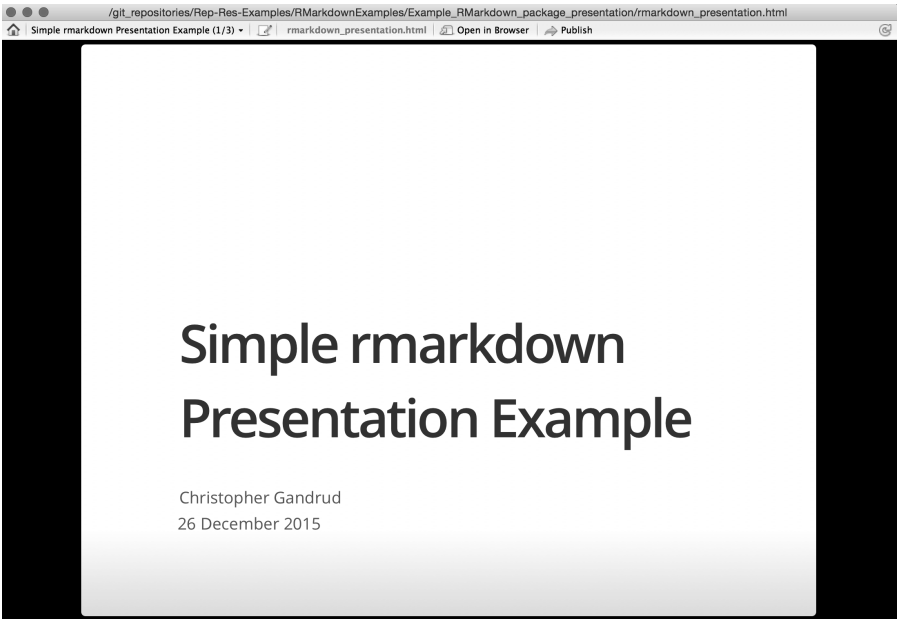


FIGURE 12.2: R Markdown/IO Slides Example Title Slide

You can create a new IO Slides or Slidy *rmarkdown* presentation in RStudio by selecting `File R Markdown...` then `Presentation` in the menu on the left of the window (see figure below). Finally, click `HTML (ioslides)` or `HTML (Slidy)`.

12.3.2 LaTeX Beamer slideshows with *rmarkdown*

As we saw in Chapter 11, creating a presentation with LaTeX beamer involves rather convoluted syntax. Luckily, we can use *rmarkdown* to create beamer presentations using much cleaner Markdown syntax.

An R Markdown beamer presentation uses the same syntax that we just saw with HTML presentations. The main difference is in the header where we use `output: beamer_presentation`. You create a new R Markdown beamer document in RStudio in a similar way as IO Slides or Slidy. The only difference is that we select `PDF (Beamer)`. As before, frame titles are delineated with two hashes (`##`). You can mark sections in much the same way with one hash. In the header you can switch the beamer theme, font theme, and color theme with `theme`, `colortheme`, and `fonttheme`, respectively. For example:

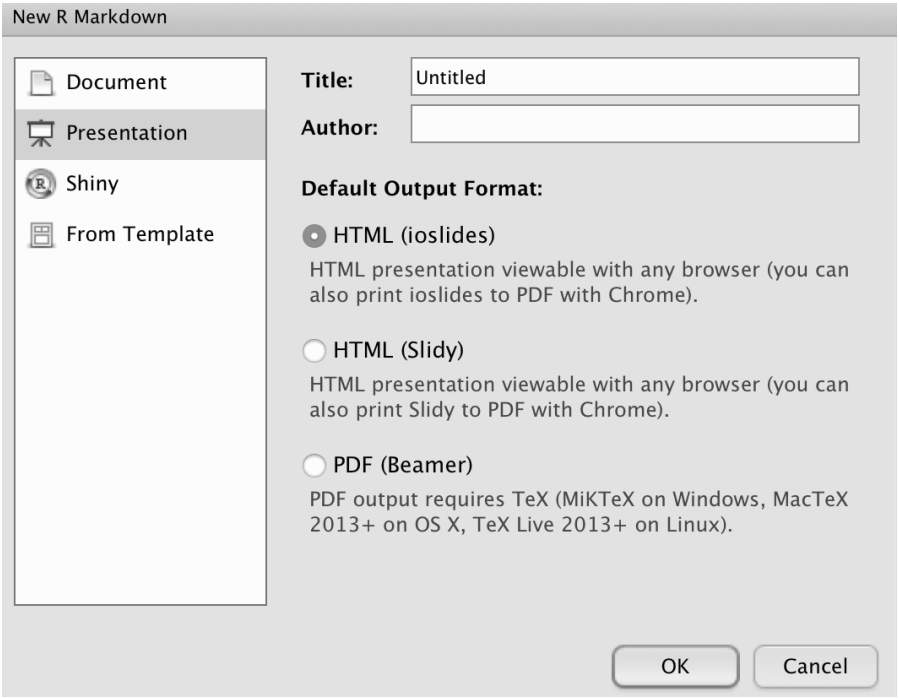


FIGURE 12.3: Create New R Markdown Presentation in RStudio

```
output:
    beamer\_presentation:
        incremental: true
        theme: "Bergen"
        colortheme: "crane"
        fonttheme: "structurebold"
```

Note that themes are placed in quotation marks. You can also include a custom template with the `template` option followed by the path to the custom template file.

12.3.3 Slideshows with Markdown and RStudio's R Presentations

Another easy, but less customizable way to create HTML slideshows is with RStudio's R Presentation documents. To get started, open RStudio and click `File`, `New`, then `R Presentation`. RStudio will then ask you to give the

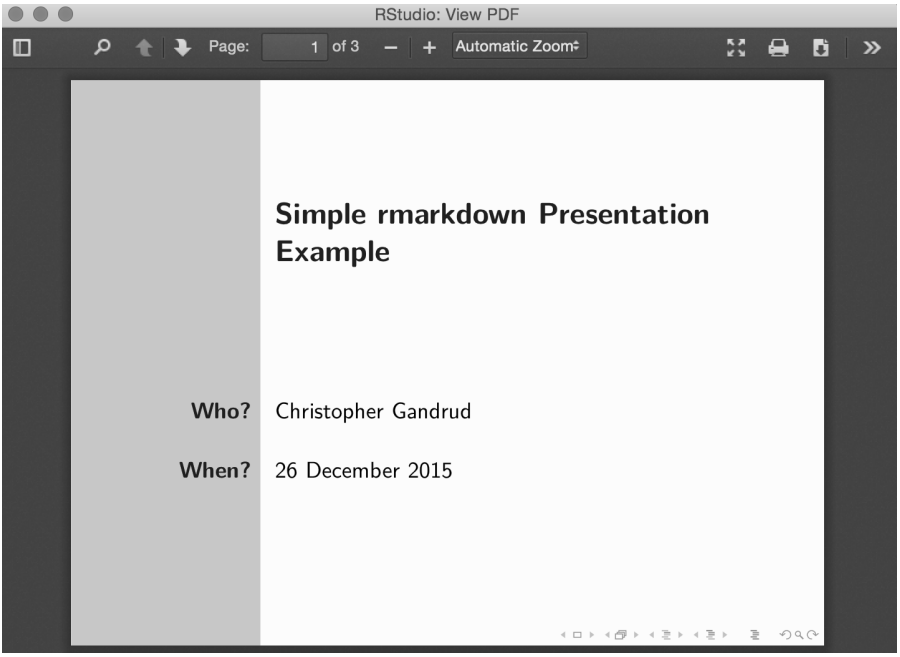


FIGURE 12.4: *rmarkdown*/Beamer Example Title Slide

presentation a name and save it in a particular file. The reason RStudio does this is because an R Presentation is not just one file. Instead, it includes:

- A *.Rpres* file, which is very similar to a *knitr* Markdown *.Rmd* file.
- A *.md* Markdown file created from the *.Rpres* file.
- *knitr* cache and figure folders, also created from the *.Rpres* file.

Editing and compiling the presentation

You change the presentation's content by editing the *.Rpres* file using the normal *knitr* Markdown syntax we've covered. The only difference is how you create new slides. Luckily, the syntax for this is very simple. Type the slide's title, then at least three equal signs (`===`). For example,

```
This is an Example .Rpres Slide Title
===
```

The very first slide is automatically the title slide and will be formatted differently from the rest.[15] Here is an example of a complete *.Rpres* file:

````
Example R Presentation
===

## Christopher Gandrud

## 1 July 2019

Access the Code
===

The code to create the following figure is available online.

To access it we type:

```{r, eval=FALSE}
# Access and run the code to create a caterpillar plot

devtools::source_url("http://bit.ly/VRKphr")
```

Caterpillar Plot
===

```{r, echo=FALSE, message=FALSE}
# Access and run the code to create a caterpillar plot

devtools::source_url("http://bit.ly/VRKphr")
```

Fertilizer Consumption Map (2003)
===

```{r CreategvisGeoMap, echo=FALSE, message=FALSE, results='asis'}
# Create geo map of global fertilizer consumption for 2003
devtools::source_url("http://bit.ly/VNnZxS")
```
````

This example includes four slides and three code chunks. The last code chunk uses the *googleVis* package to create the global map of fertilizer consumption we saw earlier. Because the slideshow we are creating is in HTML, the map

[15] As of this writing, it is a blue slide with white letters.

will be fully dynamic. Note that, like before, you will not be able to see the map in the RStudio preview, only in a web browser.

To compile the slideshow, either click the `Preview` button or save the *.Rpres* document. When you do this, you can view your updated slideshow in the *Presentation* pane. You can navigate through the slideshow using the arrow buttons at the bottom right of the *Presentation* pane. If you click the magnifying glass icon at the top of the *Presentation* pane, you will get a much larger view of the slideshow. You can also view the slideshow in your web browser by clicking on the `More` icon, then `View in Browser`.

Publishing slideshows

You can of course, view your slideshows locally. To share your presentation with others, you probably want to either publish the presentation to a standalone HTML file and host it (e.g. with a service like Netlify `https://www.netlify.com/`) or publish it directly to RPubs. For R Presentations, create a standalone HTML file by clicking the `More` button in the *Presentation* pane, then `Save as Webpage...`. Under the `More` button, you can also choose the option `Publish to RPubs...`.

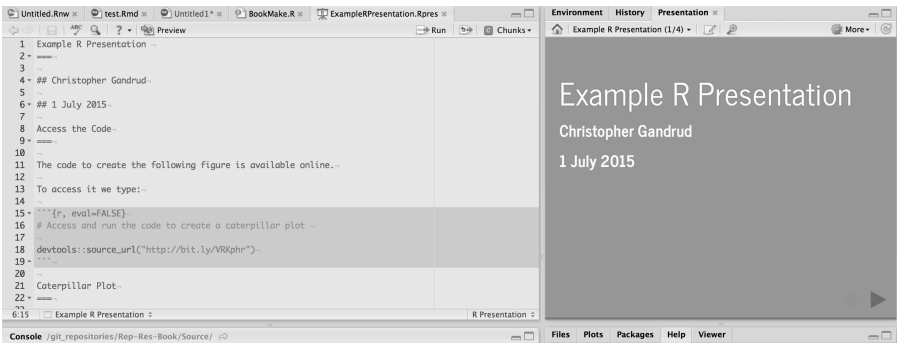


FIGURE 12.5: RStudio R Presentation Pane

12.4 Publishing HTML Documents Created with R Markdown

In Chapter 3 we saw how to publish other R Markdown documents compiled with RStudio to RPubs. The *knitr* function `knit2wp()` can be used to post a

knitted Markdown file to WordPress[16] sites, which are often used for blogging. In this section we will look at how to publish R Markdown documents using GitHub.

Standalone HTML files

You can open the HTML file rendered from any R Markdown document in your web browser. If the HTML file contains the full information for the page as it generally does when created by *rmarkdown*, e.g. the file does not depend on any auxiliary files, you can share this file via email or other means and anyone with a web browser can open it. We can of course, also send auxiliary files if need be, but this can get unwieldy.

GitHub Pages

GitHub also offers a free hosting service for webpages. These can be much more complex than a single HTML file. The simplest way to create one of these pages is to create a repository with a file called *README.Rmd*. You can `knit` this file and then create your GitHub Page with it. To do this, go to the `Settings`, then `GitHub Pages` on your repository's main GitHub website. Then click `Automatic Page Generator`. This places the contents of your *README.md* file in the page and provides you with formatting options. Click `Publish` and you will have a new website.

Clicking `Publish` creates a new orphan branch[17] called *gh-pages*. When these branches are pushed to GitHub, it will create a website based on a file called *index.html* that you include in the branch. This will be the website's main page.

If you want to create more customized and larger websites with GitHub Pages, you can manually create a GitHub Pages orphan branch and push it to GitHub. This is essentially what *slidify* did for us with its `publish` function. Imagine we have our working directory set as a repository containing an R Markdown file that we have rendered into an HTML file called *index.html*. Let's create a new orphan branch:

```
# Create orphan
gh-pages branch git checkout --orphan gh-pages
```

[16] `https://wordpress.com/`

[17] An orphan branch is a branch with a different root from other repository branches. Another way of thinking about this is that orphan branches have their own history.

Now `add` the files, `commit` the changes and `push` it to GitHub. Push it to the *gh-pages* branch like this

```
# Add files
git add .

# Commit changes
git commit -am "First gh-pages commit"

# Push branch to GitHub Pages
git push origin gh-pages
```

A new webpage will be created at: *USERNAME.github.io/REPO_NAME*. You can also add custom domain names. For details, see `https://help.github.com/en/articles/using-a-custom-domain-with-github-pages`.

12.4.1 Further information on R Markdown

We have covered many of the core capabilities of R Markdown for creating reproducible research documents. Please see RStudio's R Markdown documentation (`https://rmarkdown.rstudio.com/`) for even more information. Another tool to look into for interactive results presentation is the *shiny* package (Chang et al., 2019). It gives R the capability to create interactive web applications, not just the static websites that we have covered in this chapter. This package is well integrated with RStudio. For more information, please see `http://shiny.rstudio.com/`.

Chapter summary

In this chapter we learned a number of tools for dynamically presenting our reproducible research on the web, as well as how to create PDFs with the simple R Markdown syntax. Though LaTeX and PDFs will likely remain the main tools for presenting research in published journals and books for some time to come, choosing to also make your research available in online native formats can make it more accessible to general readers. It also allows you to take advantage of interactive tools for presenting your research. R Markdown also makes it easy to create documents in a variety of formats.

13

Conclusion

> *Well, we have completed our journey. The only thing left to do now is practice, practice, practice.* (Shotts Jr., 2012, 432)

In this book we learned a workflow for highly reproducible computational research and many of the tools needed to actually do it. Hopefully, if you haven't already, you will begin using and benefiting from these tools in your own work. Though we've covered enough material in this book to get you well on your way, there is still a lot more to learn. With most things computational (possibly most things in general), one of the best ways to continue learning is to practice and try new things. Inevitably you will hit walls, but there are almost always solutions that can be found with curiosity and patience. The R and reproducible research community is extremely helpful when it comes to finding and sharing solutions. I highly recommend getting involved in and eventually contributing to this community to get the most out of reproducible research.[1]

Before ending the book, I want to briefly address five issues we have not covered so far that are important for reproducible research: citing reproducible research, licensing this research, sharing your code with R packages, whether or not to make your research files public before publishing the results, and whether or not it is possible to completely future-proof your research.

[1] A good point of entry into the R reproducible research community is R-bloggers (`https://www.r-bloggers.com/`). The site aggregates many different blogs on R-related topics from both advanced and relatively new R users. I have found that beyond just consuming other peoples' insights, contributing to R-bloggers—having to clearly write down my steps—has sharpened my understanding of the reproducible research process and enabled me to get great feedback. Other really useful resources are the R Stack Overflow (`https://stackoverflow.com/questions/tagged/r`) and Cross Validated (`https://stats.stackexchange.com/questions/tagged/r`) sites.

13.1 Citing Reproducible Research

There are a number of well-established methods for citing presentation documents, especially published articles and books. However, as we discussed in the beginning, these documents are just the advertising for research findings rather than the actual research (Buckheit and Donoho, 1995; Donoho, 2010, 385). If other researchers are going to use the data and source code used to create the findings in their own work, they need a way of actually citing the particular data and source code they used. Citing data and source code presents unique problems. Data and source code can change and be updated over time in a way that published articles and books generally are not. As such we have a much less developed, or at least less commonly used set of standards for citing these types of materials.

One possibility is a standard for citing quantitative data sets laid out by Altman and King (2007; see also King, 2007). They argue that quantitative data set citations should:

- allow a reader to quickly understand the nature of the cited data set,
- unambiguously identify a particular version of the data set, and
- enable reliable location, retrieval, and verification of the data set.

The first issue can be solved by having a citation that includes the author, the date the data set was made public, and its title. However, these things do not unambiguously identify the data set, as it may be updated or changed and it does not enable its location and retrieval. To solve this problem, Altman and King (2007) suggest that these citations also include:

- a unique global identifier (UGI),
- a universal numeric fingerprint (UNF), and
- a bridge service.

A UGI uniquely identifies the data set. Examples include Document Object Identifiers (DOI) and the Handel System.[2] UGIs by themselves do not uniquely identify a particular version of a data set. This is where UNFs come in. They uniquely identify each version of a data set. Finally, a bridge service links the UGI and UNF to an actual document, usually posted online, so that it can be retrieved.

There are many ways to register DOIs and Handel UGIs. Most of these also include means for creating UNFs and a bridge service. Examples of services

[2] See: `http://www.handle.net/`.

that store your work and assign it DOIs are figshare[3] and Zenodo.[4] Zenodo can be integrated with GitHub so that it will store and create citations for a specific commit of a GitHub repository whenever you create a tag. For more information about integrating GitHub and Zenodo, see `https://guides.github.com/activities/citable-code/`. Please see Altman and King (2007) for details of other services.[5]

Though Altman and King (2007) are interested in data sets, their system could easily be applied to source code as well. UGIs could identify a source code file or collection of files. The UNF could identify a particular version and a bridge service would create a link to the actual files.

13.2 Licensing Your Reproducible Research

In the United States and many other countries, research, including computer code made available via the internet, is automatically given copyright protection. However, copyright protection works against the scientific goals of reproducible research, because work derived from the research falls under the original copyright protections (Stodden, 2009b, 36). To solve this problem, some authors have suggested placing code under an open source software license like the GNU General Public License (GPL) (Vandewalle et al., 2007). Stodden (2009b) argues that this type of license is not really adequate for making available the data, code, and other material needed to reproduce research findings in a way that enables scientific validation and knowledge growth. I don't want to explore the intricacies of these issues here. Nonetheless, they are important for computational researchers to think about, especially if their data and source code is publicly available. Two good places to go for more information are Stodden (2009b) and Creative Commons (2012).

13.3 Sharing Your Code in Packages

Developing R functions and putting them into packages is a good way to enable cumulative knowledge development. Many researchers spend a considerable

[3] `https://figshare.com/`

[4] `https://zenodo.org/`

[5] The Dataverse Project (`https://dataverse.org/`) offers a free service to host files that also uses the Handel System to assign UGIs, UNFs, and provides a bridge service. See Gandrud (2013a) for a comparison of Dataverse with GitHub and Dropbox for data storage.

amount of time writing code to solve problems that no one has addressed yet, or haven't addressed in a way that they believe is adequate. It is very useful if they make this code publicly accessible so that others can perhaps adopt and use it in their own work without having to duplicate the effort used to create the original functions. Abstracting your code into functions so that they can be applied to many problems and distributing them in easily installed packages makes it much easier for other researchers to adopt and use your code to help solve their research problems. The active community of researcher/package developers is one of the main reasons that R has become such a widely used and useful statistical language.

Many of the tools we have covered in this book provide a good basis to start making and distributing functions. We have discussed many of the R commands and concepts that are important for creating functions. We have also looked at Git and GitHub, which are very helpful for developing and distributing packages. Learning about Hadley Wickham's *devtools* package is probably the best next step for you to take to be able to develop and distribute functions in packages. He has an excellent introduction to *devtools* and R package development in general at `http://r-pkgs.had.co.nz/intro.html#introduction-to-devtools`.

RStudio Projects have excellent *devtools* integration and are certainly worth using. To begin creating a new package in RStudio, start a new project, preferably with Git version control (see Section 5.4.1). In the **New Project** window, select `Package`. Now you will have a new Project with all of the files and directories you need to get started making packages that will hopefully be directly useful for the computational research community.

13.4 Project Development: Public or Private?

Hopefully I have made a convincing case in this book that research results, especially in academia, should almost always be highly reproducible. The files used to create the results need to be publicly available for the research to be really reproducible.[6] During the development of a research project, however, should files be public or private?

On the one hand, openness encourages transparency and feedback. Other researchers may alert you to mistakes before a result is published. On the other hand, there are worries that you may be "scooped". Another researcher might see your files, take your idea, and publish it before you have a chance to. In

[6] There are obvious exceptions, such as when a study's participants' identities need to remain confidential.

general, this worry may be a bit overblown. Especially if you use a version control system that clearly dates all of your file versions, it would be very easy to make the case that someone has stolen your work. Hopefully this possibility would discourage any malfeasance. That being said, unlike the clear need to make research files available after publication, during research development there are good reasons for both making files public and keeping them private.

Researchers should probably make this decision on a case-by-case basis. In general, I choose to make my research repositories public to increase transparency and encourage feedback. The community of researchers in my field is relatively small and close knit. It would be hard for someone to take my work and pass it off as their own. This is especially true if many people already know that they are my ideas, because I have made by research files publicly available. Regardless, cloud storage systems like GitHub make it easy to choose whether or not to make your files public or private. You can easily keep a repository private while you create a piece of research and then make it public once the results are published.

13.5 Is it Possible to Completely Future-Proof Your Research?

In this book we've looked at a number of ways to help future-proof your research so that future researchers (and you) are able to actually reproduce it. These included storing your research in text files, clearly commenting on your code, and recording information about the software environment you used by, for example, recording your session info. Are these steps enough to completely ensure that your research will always be reproducible? The simple answer is probably no. Software changes, but it is difficult to foresee what these changes will be. Nonetheless, beyond what we have discussed so far there are other steps we can take to make our reproducible research as future-proof as possible.

One of the main obstacles to completely future-proofing your research is that no (or at least very few) pieces of software are complete. R packages are updated. R is updated. Your operating system is updated. These and other software programs discussed in this book may not only be updated, but also discontinued. Changes to the software you used to find your results may change the results someone gets reproducing your research. This problem becomes larger as you use more pieces of software in your research.

That being said, many of the software tools we have learned about in this book have future-proofing at their heart. TeX, the typesetting system that underlies

LaTeX, is probably the best example. TeX was created in 1978 and has since been maintained with future-proofing in mind (Knuth, 1990). Though changes and new versions continue to be made, we are still able to use TeX to recreate documents in their original intended form even if they were written over thirty years ago. We also saw that, though R and especially R packages change rapidly, the Comprehensive R Archive Network stores and makes accessible old versions (as the name suggests). Old versions can be downloaded by anyone wishing to reproduce a piece of research, provided the original researcher has recorded which versions they used. One approach is to use the *packrat* R package (Ushey et al., 2018) for managing the packages your project depends on. Some of the other technologies discussed in this book may be less reliable over time, so some caution should be taken if you intend to use them to create fully reproducible research.

In addition to documenting what software you used and using software that archives old versions, some have suggested another step to future-proof reproducible research: encapsulate it in a virtual machine that is available on a cloud storage system. See in particular Howe (2012). A virtual reproducible research machine would store a "snapshot [of] a researcher's entire working environment, including data, software, dependencies, notes, logs, scripts, and more". If the virtual machine is stored on a cloud server, then anyone wanting to reproduce the research could access the full computing environment used to create a piece of research (Howe, 2012, 36). As long as others could run the virtual machine and access the cloud storage system, you would not have to worry about changing software, because the exact versions of the software you used would be available in one place.

We don't have space to cover the specifics of how to create a virtual machine in this book. However, using a virtual machine is a tool that can be added to the workflow discussed in this book, rather than being a replacement for it. Carefully documenting your steps, clearly organizing your files, and dynamically tying together your data gathering, analysis, and presentation files helps you and others understand how you created a result after a research project's results have been published. Being able to understand your research will give it higher research impact, as others can more easily build on it. The steps covered in this book will still encourage you to have better work habits from the beginning of your research projects even if you will be using a virtual machine. The tools and workflow will also continue to facilitate collaboration and make it easier to dynamically update your research documents when you make changes.

Now, get started with reproducible research!

Bibliography

Allaire, J., Horner, J., Xie, Y., Marti, V., and Porte, N. (2019a). *markdown: Render Markdown with the C Library 'Sundown'*. R package version 1.1.

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2019b). *rmarkdown: Dynamic Documents for R*. R package version 2.0.

Altman, M. and King, G. (2007). A proposed standard for the scholarly citation of quantitative data. *D-Lib Magazine*, 13(3/4).

Arel-Bundock, V. (2018). *countrycode: Convert Country Names and Country Codes*. R package version 1.1.0.

Bååth, R. (2012). The state of naming conventions in R. *The R Journal*, 4(2):74–75.

Bache, S. M. and Wickham, H. (2014). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5.

Bacon, F. R. (1859). *Opera quaedam hactenus inedita. Vol. I. containing I.–Opus tertium. II.–Opus minus. III.–Compendium philosophiae.* Google eBook. Retrieved from `http://books.google.com/books?id=wMUKAAAAYAAJ`.

Ball, R. and Medeiros, N. (2011). Teaching integrity in empirical research: A protocol for documenting data management and analysis. *The Journal of Economic Education*, 43(2):182–189.

Barr, C. D. (2012). Establishing a culture of reproducibility and openness in medical research with an emphasis on the training years. *Chance*, 25(3):8–10.

Bowers, J. (2011). Six steps to a better relationship with your future self. *The Political Methodologist*, 18(2):2–8.

Box, G. E. and Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26:211–252.

Braude, S. (1979). *ESP and Psychokinesis. A Philosophical Examination*. Temple University Press, Philadelphia, PA.

Buckheit, J. B. and Donoho, D. L. (1995). Wavelab and reproducible research. In Antoniadis, A., editor, *Wavelets and Statistics*, pages 55–81. Springer, New York.

Burbidge, J. B. and Robb, L. (1988). Alternative transformations to handle extreme values of the dependent variable. *Journal of the American Statistical Association*, 83(401):123–127.

Bürkner, P.-C. (2020). *brms: Bayesian Regression Models using 'Stan'*. R package version 2.11.0.

Chang, W. (2012). *R Graphics Cookbook: Practical Recipes for Visualizing Data*. O'Reilly Media, Inc., Sebastopol, CA.

Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. (2019). *shiny: Web Application Framework for R*. R package version 1.4.0.

Crawley, M. J. (2005). *Statistics: An Introduction Using R*. John Wiley & Sons Ltd., Chichester.

Crawley, M. J. (2013). *The R Book*. John Wiley & Sons Ltd., Chichester, 2nd edition.

Creative Commons (2012). Data. `http://wiki.creativecommons.org/Data`.

Donoho, D. L. (2002). How to be a highly cited author in mathematical sciences. *in-cites*. `http://www.in-cites.com/scientists/DrDavidDonoho.html`.

Donoho, D. L. (2010). An invitation to reproducible computational research. *Biostatistics*, 11(3):385–388.

Donoho, D. L., Maleki, A., Shahram, M., Rahman, I. U., and Stodden, V. (2009). Reproducible research in computational harmonic analysis. *Computing in Science & Engineering*, 11(1):8–18.

Dowle, M. and Srinivasan, A. (2019). *data.table: Extension of 'data.frame'*. R package version 1.12.8.

Ehrenberg, A. S. C. (1977). Rudiments of numeracy. *Journal of the Royal Statistical Society. Series A General*, 140(3):277–297.

Fomel, S. and Claerbout, J. F. (2009). Reproducible Research. *Computing in Science & Engineering*, 11(1):5–7.

Frazier, M. (2008). Bash parameter expansion. *The Linux Journal*. Available at: `http://www.linuxjournal.com/content/bash-parameter-expansion`.

Galili, T., Rowlingson, B., Hejblum, B., Dason, Grothendieck, G., Daroczi, G., Andrew, H., James, Leeper, T., VitoshKa, Xie, Y., Friendly, M., Rohmeyer, K., Menne, D., Hunt, T., Hiramura, T., Boessenkool, B., Godfrey, J., Allard, T., Chen, C., Hill, J., and Park, C.-Y. (2018). *installr: Using R to Install Stuff (Such As: R, 'Rtools', 'RStudio', 'Git', and More!)*. R package version 0.20.0.

Gandrud, C. (2013a). Github: A tool for social data set development and verification in the cloud. *The Political Methodologist*, 20(2):2–7.

Gandrud, C. (2013b). The diffusion of financial supervisory governance ideas. *Review of International Political Economy*, 20(4):881–916.

Gandrud, C. and Grafström, C. (2015). Inflated expectations: How government partisanship shapes bureaucrats' inflation forecasts. *Political Science Research and Methods*. Available at: `http://dx.doi.org/10.1017/psrm.2014.34`.

Gelman, A. (2011). Tables as graphs: The Ramanujan principle. *Significance*, 8(4):183.

Gesmann, M. and de Castillo, D. (2019). *googleVis: R Interface to Google Charts*. R package version 0.6.4.

Healy, K. (2018). *Data Visualization: A Practical Introduction*. Princeton University Press.

Henry, L. and Wickham, H. (2019). *purrr: Functional Programming Tools*. R package version 0.3.3.

Herndon, T., Ash, M., and Pollin, R. (2014). Does high public debt consistently stifle economic growth? a critique of Reinhart and Rogoff. *Cambridge Journal of Economics*, 38(2):257–279.

Hlavac, M. (2018). *stargazer: Well-Formatted Regression and Summary Statistics Tables*. R package version 5.2.2.

hong Chan, C. and Leeper, T. J. (2018). *rio: A Swiss-Army Knife for Data I/O*. R package version 0.5.16.

Howe, B. (2012). Virtual appliances, cloud computing, and reproducible research. *Computing in Science & Engineering*, 14(4):36–41.

Hyndman, R. J. (2010). Transforming data with zeros. Available at: `http://robjhyndman.com/hyndsight/transformations/`. Accessed March 2015.

Kelly, C. D. (2006). Replicating empirical research in behavioral ecology: How and why it should be done but rarely ever is. *The Quarterly Review of Biology*, 81(3):221–236.

King, G. (1995). Replication, replication. *PS: Political Science and Politics*, 28(3):444–452.

King, G. (2007). An introduction to the dataverse network as an infrastructure for data sharing. *Sociological Methods & Research*, 36(2):173–199.

King, G., Keohane, R., and Verba, S. (1994). *Designing Social Inquiry*. Princeton University Press, Princeton.

Kitzes, J., Turek, D., and Deniz, F., editors (2018). *The Practice of Repro-*

ducible Research: Case Studies and Lessons from the Data-Intensive Sciences. University of California Press, Oakland, CA.

Kluyver, T., Angerer, P., Schulz, J., and Ram, K. (2019). *IRkernel: Native R Kernel for the 'Jupyter Notebook'*. R package version 1.1.

Knuth, D. E. (1990). The future of tex and metafont. *NTG: Maps*, 5:145.

Knuth, D. E. (1992). *Literate Programming*. CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, CA.

Kross, S. (2018). *The Unix Workbench*. self published. Accessible at: `https://seankross.com/the-unix-workbench/`.

Leifeld, P. (2017). *texreg: Conversion of R Regression Output to LaTeX or HTML Tables*. R package version 1.36.23.

Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In Härdle, W. and Rönz, B., editors, *Compstat 2002: Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg. `http://www.stat.uni-muenchen.de/~leisch/Sweave`.

Lykken, D. T. (1968). Statistical significance in psychological research. *Psychologial Bulletin*, 70:151–159.

MacFarlane, J. (2019). *Pandoc: A Universal Document Converter*. Version 2.7.3.

Makel, M. C. and Plucker, J. A. (2014). Facts are more important than novelty: Replication in the education sciences. *Educational Researcher*, 43(6):304–316.

Matloff, N. (2011). *The Art of Programming in R: A Tour of Statistical Programming Design*. No Starch Press, San Francisco.

Mesirov, J. P. (2010). Accessible reproducible research. *Science*, 327(5964):415–416.

Meyer, A. (2006). Repeating patterns of mimicry. *PLoS Biol*, 4(10).

Munzert, S., Rubba, C., Meißner, P., and Nyhuis, D. (2015). *Automated Data Collection with R: A Practical Guide to Web Scraping and Text Mining*. Wiley, Chichester.

Murrell, P. (2011). *R Graphics*. Chapman & Hall/CRC Press, Boca Raton, FL, 2nd edition.

Müller, K. (2017). *here: A Simpler Way to Find Your Files*. R package version 0.1.

Müller, K. and Walthert, L. (2019). *styler: Non-Invasive Pretty Printing of R Code*. R package version 1.2.0.

Müller, K. and Wickham, H. (2019). *tibble: Simple Data Frames*. R package version 2.1.3.

Nagler, J. (1995). Coding style and good computing practices. *PS: Political Science and Politics*, 28(3):488–492.

Nosek, B. A., Spies, J. R., and Motyl, M. (2012). Scientific utopia: II. Restructring incentives and practices to promote truth over publishability. *Perspectives on Psychological Science*, 7(6):615–631.

O'Neal, C. and Schutt, R. (2013). *Doing Data Science: Straight Talk from the Frontline*. O'Reilly Media Inc., Sebastopol, CA.

Ooms, J., Temple Lang, D., and Hilaiel, L. (2018). *jsonlite: A Robust, High Performance JSON Parser and Generator for R*. R package version 1.6.

Pemstein, D., Meserve, S. A., and Melton, J. (2010). Democratic compromise: A latent variable analysis of ten measures of regime type. *Political Analysis*, 18(4):426–449.

Peng, R. D. (2009). Reproducible research and biostatistics. *Biostatistics*, 10(3):405–408.

Peng, R. D. (2011). Reproducible research in computational science. *Science*, 334:1226–1227.

Peng, R. D. (2014). The real reason reproducible research is important. *Simply Statistics*. `http://simplystatistics.org/2014/06/06/the-real-reason-reproducible-research-is-important/`.

Piwowar, H. A., Day, R. S., and Fridsma, D. B. (2007). Sharing detailed research data is associated with increased citation rate. *PLoS ONE*, 2(3):1–5.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. `http://www.R-project.org/`.

Ramsey, N. (2011). Noweb: A simple, extensible tool for literate programming. `http://www.cs.tufts.edu/~nr/noweb/`.

Reinhart, C. and Rogoff, K. (2010). Growth in a time of debt. *American Economic Review: Papers & Proceedings*, 100.

Rinker, T. and Kurkiewicz, D. (2019). *pacman: Package Management Tool*. R package version 0.5.1.

Rokem, A., Marwick, B., and Staneva, V. (2018). Assessing reproducibility. In Kitzes, J., Turek, D., and Deniz, F., editors, *The Practice of Reproducible Research: Case Studies and Lessons from the Data-Intensive Sciences*, pages 3–18. University of California Press, Oakland, CA.

RStudio, Inc. (2019). *RStudio: Integrated development environment for R*. Boston, MA. Version 1.2.1572.

Shotts Jr., W. E. (2012). *The Linux Command-line: A Complete Introduction*. No Starch Press, San Francisco.

Stodden, V. (2009a). The reproducible research standard: Reducing legal barriers to scientific knowledge and innovation. In *Communia: Global Science & Economics of Knowledge-Sharing Institutions Torino, Italy June 30*. `http://www.stanford.edu/~vcs/talks/VictoriaStoddenCommuniaJune2009-2.pdf`.

Stodden, V. (2009b). The legal framework for reproducible scientific research. *Computing in Science & Engineering*, 11(1):35–40.

Temple Lang, D. (2020). *XML: Tools for Parsing and Generating XML Within R and S-Plus*. R package version 3.99-0.2.

Temple Lang, D. and the CRAN team (2020). *RCurl: General Network (HTTP/FTP/...) Client Interface for R*. R package version 1.95-4.13.

Therneau, T. M. (2019). *survival: Survival Analysis*. R package version 3.1-8.

Tufte, E. R. (2001). *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 2nd edition.

Ushey, K., McPherson, J., Cheng, J., Atkins, A., and Allaire, J. (2018). *packrat: A Dependency Management System for Projects and their R Package Dependencies*. R package version 0.5.0.

Vaidyanathan, R., Xie, Y., Allaire, J., Cheng, J., and Russell, K. (2019). *htmlwidgets: HTML Widgets for R*. R package version 1.5.1.

van Belle, G. (2008). *Statistical Rules of Thumb*. John Wiley & Sons, Hoboken, NJ, 2nd edition.

Vandewalle, P. (2012). Code sharing is associated with research impact in image processing. *Computing in Science & Engineering*, 14(4):42–47.

Vandewalle, P., Barrenetxea, G., Jovanovic, I., Ridolfi, A., and Vetterli, M. (2007). Experiences with reproducible research in various facets of signal processing research. *Acoustics, Speech and Signal Processing*, 4:1253–1256.

White, J. M. (2019). *ProjectTemplate: Automates the Creation of New Statistical Analysis Projects*. R package version 0.9.0.

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer, New York, 2nd edition.

Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.

Wickham, H. (2014a). *Advanced R*. Chapman & Hall/CRC Press, Boca Raton, FL.

Wickham, H. (2014b). Tidy Data. *Journal of Statistical Software*, 59(10):1–23.

Wickham, H. (2019a). *httr: Tools for Working with URLs and HTTP*. R package version 1.4.1.

Wickham, H. (2019b). *rvest: Easily Harvest (Scrape) Web Pages*. R package version 0.3.5.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., and Yutani, H. (2019a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.2.1.

Wickham, H., François, R., Henry, L., and Müller, K. (2019b). *dplyr: A Grammar of Data Manipulation*. R package version 0.8.3.

Wickham, H. and Henry, L. (2019). *tidyr: Tidy Messy Data*. R package version 1.0.0.

Wickham, H., Hester, J., and Chang, W. (2019c). *devtools: Tools to Make Developing R Packages Easier*. R package version 2.2.1.

Wickham, H. and Ruiz, E. (2019). *dbplyr: A 'dplyr' Back End for Databases*. R package version 1.4.2.

Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., and Wilson, P. (2012). Best practices for scientific computing. *arXiv*, 29 November 2012:1–6. Available at: `http://arxiv.org/pdf/1210.0530v3`.

World Bank (2018). *World Development Indicators*.

Xie, Y. (2015). *R Markdown: The Definitive Guide*. Chapman & Hall/CRC, Boca Raton, Florida.

Xie, Y. (2018). *animation: A Gallery of Animations in Statistics and Utilities to Create Animations*. R package version 2.6.

Xie, Y. (2020a). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.17.

Xie, Y. (2020b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.27.

Xie, Y. (2020c). *tinytex: Helper Functions to Install and Maintain TeX Live, and Compile LaTeX Documents*. R package version 0.19.

Xie, Y. (2020d). *xfun: Miscellaneous Functions by 'Yihui Xie'*. R package version 0.12.

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

Index

absolute file path, 71
author-year citations, 228
Awk, 160

Bash, 159, 160
beamer, 248, 250–251
bibliography, 245
Bootswatch, 245
    Flatly, 245
Bourne shell, 160

child directory, 70
comma separated file format, 89
component selector, 130, 199
contingency table, 131
CRAN, 47
    mirror, 20
CSS, 195, 245, 247, 248
CSV, 88, 89

Dropbox, 90

escape character, 71
Excel, 122

Gawk, 160
GDPR, 10
GIF, 209
git
    .gitignore, 102
    add, 256
    branches, 101
    checkout, 99
    clone, 105
    commit, 256
    ignore, 102
    orphan branch, 255
    pull, 108
    push, 256
    remote repository, 105
    repo, 93
    repos, 96
    repositories, 96
    repository, 93
    tags, 100
GitHub, xix, 58, 255
    gh-pages branch, 255
    Markdown, 58
    Pages, 255
Google Chrome
    Developer Tools, 247
Google Drive, 90

Harvard style citations, 228
Haskell, 160
HTML, 247
    alt, 194
    caption, 176
    height, 195
    img, 195
    px, 195
    src, 195
    table border, 176
    tag, 195
    tags, 175
    tbody, 176
    thead, 176
    width, 195
HTML5, 248

ImageMagick, 21
in-text citation, 245
infinity, 140

JavaScript, 247, 248
JPEG, 192

Julia, 29, 65, 159, 160
Jupyter, 58

kebab-case, 73
knitr option
  cache, 155, 164
  cache.extra, 156
  dependson, 156
  dev, 200
  echo, 155, 179
  engine, 159
  error, 155
  eval, 155
  fig.align, 197
  fig.cap, 197
  fig.ext, 238
  fig.path, 196
  include, 154
  message, 155
  out.height, 196
  out.width, 197
  results, 155, 168, 179
  root.dir, 77
  size, 156
  warning, 155

LaTeX
  ampersand, 170
  beamer slides, 233
  begin document, 218
  caption, 172, 193
  center, 172
  centre, 169
  cite, 194
  cross-references, 226
  environment, 169
  figure, 199
  figure environment, 193
  float, 193–194
  footnotes, 226
  includegraphics, 192
  label, 193
  landscape, 184
  list appear, 234
  outlines, 233
  packages, 218
  preamble, 244
  scriptsize, 194
  Sexpr, 158
  table, 169, 171, 180, 182
  table of contents, 233
  tabular, 169, 170, 180, 182
  texttt, 157
  textwidth, 193, 197
  verb, 157
LaTeX command
  allowframebreaks, 233
  author, 218
  bibliography, 228
  bibliographystyle, 228
  chapter, 222
  citation, 229
  cite, 228
  date, 218
  documentclass, 217, 233
  emph, 221, 223
  frame, 233, 234
  frametitle, 233
  hline, 222
  href, 221
  hrulefill, 222
  hspace, 222
  includegraphics, 24
  item, 225
  label, 226
  maketitle, 218
  pageref, 226
  paragraph, 222
  part, 222
  ref, 226
  section, 222, 233
  Sexpr, 221
  subparagraph, 222
  subsection, 222
  tableofcontents, 233
  textbf, 223
  thanks, 221
  title, 218
  titlepage, 233
  usecolortheme, 233

useinnertheme, 233
useoutertheme, 233
usepackage, 218
usetheme, 233
vspace, 222
LaTeX environment
abstract, 219
body, 218
document, 233
enumerate, 225
figure, 204, 226
frame, 233
itemize, 225
table, 226
tabular, 222
verbatim, 234
LaTeX package
array, 170
graphics, 185
graphicx, 192
hyperref, 226
lscape, 184
natbib, 245
url, 218
LaTeXbeamer, 250–251
link rot, 90
Linux, 21
literate programming, 28, 30
long formatted data, 132

Makefile, xx
Markdown
bold, 241
footnotes, 246
italics, 241
lines, 240
new line, 241
special characters, 241
matrix transpose, 133
missing values, 130

NA, 130
Netlify, 254

outliers, 137

Pandoc, 14, 15
footnotes, 246
parent directory, 70
PATH, 21
pdfLaTeX, 61, 241
pipe, 44, 76
PNG, 192
PowerPoint, 248
Python, 29, 65, 159, 160

R
component selector, 39
data frame, 37, 132
inf, 140
order, 136
ordering data, 136
packages, 19
renaming variables, 135
reshaping data, 132
session info, xx
sort, 136
subset, 137
subsetting data, 137
R function
<-, 35
?, 43
$, 39
%>%, 44, 76, 80
aes, 204
apply, 127
arrange, 136
as.factor, 143
attach, 40
boxplot, 198
brm, 185
c, 188
cat, 78, 188
cbind, 38, 143
character, 143
class, 36
combine, 37
cord_flip, 208
cut, 142
data.frame, 37, 38, 132
desc, 136

detach, 41
dir.create, 77, 78
download.file, 123
download.file(), xx
duplicated, 146
factor, 141, 143
file.copy, 79
file.create, 77
file.rename, 78
fix, 131
geom_hline, 208
geom_line, 204
geom_pointrange, 207
getURL, 24
getwd, 75
ggplot, 202–208
ggplot2, 208
gvisGeoChart, 209
head, 40, 76
here, 76
hist, 164, 198
history, 46
import, xx, 24, 91, 106, 123
include_graphics, 24, 195
install.packages, 47
kable, 167, 177–178, 238
knit, 62
knit2html, 62, 63
knit2pdf, 63
knit2wp, 254
lapply, 231
library, 47, 231
list.files, 76
lm, 178
load, 45
ls, 45
markdownToHTML, 62, 248
mean, 39, 158
merge, 24, 143, 144
methods, 179
names, 38, 129, 130
ncol, 130
nrow, 130
numberic, 143
options, 46, 158
order, 136
pairs, 198, 200
paste0, 107, 124
pivot_longer, 133
pivot_wider, 134
plot, 198
plot.survfit, 205
print, 180, 210
print.xtable, 180
publish, 255
read.csv, 123
read.dta, 121
read.table, 24, 121, 123
readHTMLTable, 126
rename, 135
render, 63
reorder, 208
reshape, 133
rio, 122
rm, 45
rnorm, 163
round, 44, 158
save, 46
save.image, 45
scale_color_discrete, 204
scale_linetype, 204
search, 40
select, 147
sessionInfo, 74
set.seed, 163
setwd, 76, 114
source, 49, 114, 115, 161
source(), xx
source_gist, 162
source_url, 91, 162, 200, 201
stanplot, 208
str, 129
subset, 137
summary, 130, 178
system, 83, 161
table, 131
tail, 129
tempfile, 123
texi2pdf, 62
texreg, 175

tibble, 131
toBibTeX, 230
toLatex, 74
union, 145
unlink, 78
View, 131
WDI, xix
WDIsearch, 125
with, 40
write.csv, 89, 177
write.table, 88
write_bib, 230
xlab, 204
xtable, 178–181
ylab, 204
R package
brms, 185
data.table, 145
dbplyr, 145
dplyr, 44
ggplot2, 202–208
googleVis, 248
installr, 21
magrittr, 44
markdown, 16
ProjectTemplate, 73
rio, 24
rmarkdown, 15
Rtools, 21
shiny, 256
styler, 30
tidyr, 133
xfun, 20
R Presentation, 251
RCurl, 21
README file, xx
relative file path, 71
rename variable, 135
reshape data, 132
root directory, 70
RPubs, 58
RStudio
Environment tab, 131
Presentation pane, 254
R Presentation, 251
Ruby, 159, 160

SAS, 160
Scala, 36
shell command, 80
cd, 80, 96
cp, 83
echo, 81, 96
git add, 97, 102
git branch, 101
git checkout, 99, 101
git clone, 105
git commit, 97
git init, 97
git merge, 102
git pull, 108
git push, 105, 106
git remote, 105
git rm, 104
git status, 99
git tag, 100
head, 80
ls, 80
make, 119
mkdir, 81, 96
mv, 82
options, 82
pwd, 80
rm, 82
tree, 70
Shiny, 48
SQL, 48
Stan, 159, 160

tab separated file format, 89
Terminal, 21
tidy data, 132
Tidyverse, 132
time-series cross-sectional, 132
TSCS, 132
TSV, 88
Twitter, 65
type safe, 36

UTF-8, 61

wide formatted data, 132
wildcard, 117
Windows, 21
working directory, 71

XeLaTeX, 61, 245

YAML, 50, 198

Zenodo, 90
Zsh, 79