The R Series

Geocomputation with R

Second Edition

Robin Lovelace
Jakub Nowosad
Jannes Muenchow

A Chapman & Hall Book

CRC Press
Taylor & Francis Group

Geocomputation with R

Geocomputation with R is for people who want to analyze, visualize, and model geographic data with open source software. The book provides a foundation for learning how to solve a wide range of geographic data analysis problems in a reproducible, and therefore scientifically sound and scalable way. The second edition features numerous updates, including the adoption of the high-performance terra package for all raster data processing, detailed coverage of the spherical geometry engine s2, updated information on coordinate reference systems and new content on openEO, STAC, COG, and gdalcubes. The data visualization chapter has been revamped around version 4 of the tmap package, providing a fresh perspective on creating publication-quality maps from the command line. The importance of the book is also highlighted in a new foreword by Edzer Pebesma.

The book equips you with the knowledge and skills necessary to tackle a wide range of issues manifested in geographic data, including those with scientific, societal, and environmental implications. The book is especially well-suited to:

- Data scientists and engineers interested in upskilling to handle spatial data.
- People with existing geographic data skills interested in developing powerful geosolutions via code.
- Anyone who needs to work with spatial data in a reproducible and scalable way.

The book is divided into three parts: Foundations, Extensions, and Applications, covering progressively more advanced topics. The exercises at the end of each chapter provide the necessary skills to address various geospatial problems, with solutions and supplementary materials available at r.geocompx.org/solutions/.

The authors are well-established geospatial researchers and teachers. Dr. Robin Lovelace is an Associate Professor at the University of Leeds, is the developer of high impact applications for more data-driven transport planning and policy. Dr. Jakub Nowosad is an Associate Professor at the Adam Mickiewicz University in Poznan, where his focus is on methods for analyzing and understanding spatial patterns in the environment. Dr. Jannes Muenchow is a researcher, engineer and developer at the data science consultancy Cynkra, where he holds a dual role as a senior data scientist and supporting DevOps engineer.

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

Behavior Analysis with Machine Learning Using R
Enrique Garcia Ceja

Rasch Measurement Theory Analysis in R: Illustrations and Practical Guidance for Researchers and Practitioners
Stefanie Wind and Cheng Hua

Spatial Sampling with R
Dick R. Brus

A Criminologist's Guide to R: Crime by the Numbers
Jacob Kaplan

Analyzing US Census Data: Methods, Maps, and Models in R
Kyle Walker

ANOVA and Mixed Models: A Short Introduction Using R
Lukas Meier

Tidy Finance with R
Christoph Scheuch, Stefan Voigt, and Patrick Weiss

Deep Learning and Scientific Computing with R torch
Sigrid Keydana

Model-Based Clustering, Classification, and Density Estimation Using mclust in R
Lucca Scrucca, Chris Fraley, T. Brendan Murphy, and Adrian E. Raftery

Spatial Data Science: With Applications in R
Edzer Pebesma and Roger Bivand

Modern Data Visualization with R
Robert Kabacoff

Learn R: As a Language, Second Edition
Pedro J. Aphalo

Spatial Analysis in Geology Using R
Pedro M. Nogueira

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

Geocomputation with R, Second Edition
Robin Lovelace, Jakub Nowosad, Jannes Muenchow

For more information about this series, please visit: https://www.crcpress.com/Chapman--HallCRC-The-R-Series/book-series/CRCTHERSER

Geocomputation with R
Second Edition

Robin Lovelace, Jakub Nowosad, and
Jannes Muenchow

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

Designed cover image: © Benjamin Nowak

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

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

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

First edition published by CRC Press 2019

Library of Congress Cataloging-in-Publication Data
Names: Lovelace, Robin, author. | Nowosad, Jakub, author. | Münchow, Jannes, author.
Title: Geocomputation with R / Robin Lovelace, Jakub Nowosad, and Jannes Muenchow.
Description: Second edition. | Boca Raton, FL : CRC Press, 2025. | Series: Chapman & Hall/CRC the R series | Includes bibliographical references and index.
Identifiers: LCCN 2024027469 (print) | LCCN 2024027470 (ebook) | ISBN 9781032229799 (hardback) | ISBN 9781032248882 (paperback) | ISBN 9781003280569 (ebook)
Subjects: LCSH: Geographic information systems. | Information storage and retrieval systems--Geography. | Spatial analysis (Statistics)--Data processing. | R (Computer program language)
Classification: LCC G70.2 .L69 2025 (print) | LCC G70.2 (ebook) | DDC 910.285/5133--dc23/eng/20240807
LC record available at https://lccn.loc.gov/2024027469
LC ebook record available at https://lccn.loc.gov/2024027470

ISBN: 978-1-032-22979-9 (hbk)
ISBN: 978-1-032-24888-2 (pbk)
ISBN: 978-1-003-28056-9 (ebk)

DOI: 10.1201/9781003280569

Typeset in Latin Modern font
by KnowledgeWorks Global Ltd

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

Access the Support Material: r.geocompx.org/solutions/

For Katy

Dla mojej rodziny

Für meine Katharina und alle unsere Kinder

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

Contents

Foreword (2nd Edition) **xiii**

Preface **xv**

1 Introduction **1**
1.1 What is geocomputation? . . . 2
1.2 Why use open source tools for geocomputation? . . . 4
1.3 Why use R for geocomputation? . . . 6
1.4 Software for geocomputation . . . 7
1.5 R's spatial ecosystem . . . 9
1.6 History of R-spatial . . . 10
1.7 Exercises . . . 14

I Foundations **15**

2 Geographic data in R **17**
2.1 Introduction . . . 18
2.2 Vector data . . . 19
2.2.1 Introduction to simple features . . . 21
2.2.2 Why simple features? . . . 26
2.2.3 Basic maps . . . 27
2.2.4 Geometry types . . . 30
2.2.5 The sf class . . . 32
2.2.6 Simple feature geometries (sfg) . . . 33
2.2.7 Simple feature columns (sfc) . . . 35
2.2.8 The sfheaders package . . . 37
2.2.9 Spherical geometry operations with S2 . . . 39
2.3 Raster data . . . 41
2.3.1 R packages for working with raster data . . . 42
2.3.2 Introduction to terra . . . 43
2.3.3 Basic map-making . . . 45
2.3.4 Raster classes . . . 45
2.4 Coordinate Reference Systems . . . 47
2.4.1 Geographic coordinate reference systems . . . 47
2.4.2 Projected coordinate reference systems . . . 48

2.5 Units . . . 50
2.6 Exercises . . . 52

3 Attribute data operations 53
3.1 Introduction . . . 53
3.2 Vector attribute manipulation . . . 54
3.2.1 Vector attribute subsetting . . . 56
3.2.2 Chaining commands with pipes . . . 59
3.2.3 Vector attribute aggregation . . . 60
3.2.4 Vector attribute joining . . . 62
3.2.5 Creating attributes and removing spatial information . . . 65
3.3 Manipulating raster objects . . . 67
3.3.1 Raster subsetting . . . 68
3.3.2 Summarizing raster objects . . . 70
3.4 Exercises . . . 71

4 Spatial data operations 73
4.1 Introduction . . . 73
4.2 Spatial operations on vector data . . . 74
4.2.1 Spatial subsetting . . . 74
4.2.2 Topological relations . . . 77
4.2.3 Distance relations . . . 81
4.2.4 DE-9IM strings . . . 82
4.2.5 Spatial joining . . . 85
4.2.6 Distance-based joins . . . 86
4.2.7 Spatial aggregation . . . 89
4.2.8 Joining incongruent layers . . . 90
4.3 Spatial operations on raster data . . . 92
4.3.1 Spatial subsetting . . . 92
4.3.2 Map algebra . . . 94
4.3.3 Local operations . . . 95
4.3.4 Focal operations . . . 98
4.3.5 Zonal operations . . . 100
4.3.6 Global operations and distances . . . 101
4.3.7 Map algebra counterparts in vector processing . . . 101
4.3.8 Merging rasters . . . 101
4.4 Exercises . . . 102

5 Geometry operations 105
5.1 Introduction . . . 105
5.2 Geometric operations on vector data . . . 106
5.2.1 Simplification . . . 106
5.2.2 Centroids . . . 109
5.2.3 Buffers . . . 109
5.2.4 Affine transformations . . . 111

5.2.5 Clipping . . . 113
5.2.6 Subsetting and clipping . . . 115
5.2.7 Geometry unions . . . 117
5.2.8 Type transformations . . . 118
5.3 Geometric operations on raster data . . . 122
5.3.1 Geometric intersections . . . 123
5.3.2 Extent and origin . . . 123
5.3.3 Aggregation and disaggregation . . . 125
5.3.4 Resampling . . . 127
5.4 Exercises . . . 130

6 Raster-vector interactions 133
6.1 Introduction . . . 133
6.2 Raster cropping . . . 133
6.3 Raster extraction . . . 135
6.4 Rasterization . . . 140
6.5 Spatial vectorization . . . 142
6.6 Exercises . . . 145

7 Reprojecting geographic data 147
7.1 Introduction . . . 147
7.2 Coordinate reference systems . . . 148
7.3 Querying and setting coordinate systems . . . 151
7.4 Geometry operations on projected and unprojected data . . . 153
7.5 When to reproject? . . . 158
7.6 Which CRS to use? . . . 158
7.7 Reprojecting vector geometries . . . 161
7.8 Reprojecting raster geometries . . . 163
7.9 Custom map projections . . . 167
7.10 Exercises . . . 169

8 Geographic data I/O 171
8.1 Introduction . . . 171
8.2 File formats . . . 172
8.3 Data input (I) . . . 174
8.3.1 Vector data . . . 175
8.3.2 Raster data . . . 179
8.4 Data output (O) . . . 180
8.4.1 Vector data . . . 180
8.4.2 Raster data . . . 181
8.5 Geoportals . . . 183
8.6 Geographic data packages . . . 184
8.7 Geographic metadata . . . 187

8.8 Geographic web services . . . 188
8.9 Visual outputs . . . 191
8.10 Exercises . . . 191

II Extensions 193

9 Making maps with R 195
9.1 Introduction . . . 196
9.2 Static maps . . . 197
9.2.1 tmap basics . . . 197
9.2.2 Map objects . . . 199
9.2.3 Visual variables . . . 200
9.2.4 Scales . . . 203
9.2.5 Legends . . . 208
9.2.6 Layouts . . . 209
9.2.7 Faceted maps . . . 210
9.2.8 Inset maps . . . 212
9.3 Animated maps . . . 216
9.4 Interactive maps . . . 219
9.5 Mapping applications . . . 224
9.6 Other mapping packages . . . 229
9.7 Exercises . . . 234

10 Bridges to GIS software 237
10.1 Introduction . . . 237
10.2 **qgisprocess**: a bridge to QGIS and beyond . . . 240
10.2.1 Vector data . . . 242
10.2.2 Raster data . . . 247
10.3 SAGA . . . 249
10.4 GRASS GIS . . . 252
10.5 When to use what? . . . 257
10.6 Bridges to GDAL . . . 258
10.7 Bridges to spatial databases . . . 259
10.8 Bridges to cloud technologies and services . . . 263
10.8.1 STAC, COGs, and data cubes in the cloud . . . 264
10.8.2 openEO . . . 265
10.9 Exercises . . . 267

11 Scripts, algorithms and functions 269
11.1 Introduction . . . 269
11.2 Scripts . . . 270
11.3 Geometric algorithms . . . 273
11.4 Functions . . . 278
11.5 Programming . . . 280
11.6 Exercises . . . 282

12 Statistical learning **285**
12.1 Introduction . . . 286
12.2 Case study: Landslide susceptibility . . . 287
12.3 Conventional modeling approach in R . . . 289
12.4 Introduction to (spatial) cross-validation . . . 292
12.5 Spatial CV with **mlr3** . . . 292
12.5.1 Generalized linear model . . . 293
12.5.2 Spatial tuning of machine-learning hyperparameters . . . 298
12.6 Conclusions . . . 303
12.7 Exercises . . . 304

III Applications **307**

13 Transportation **309**
13.1 Introduction . . . 309
13.2 A case study of Bristol . . . 311
13.3 Transport zones . . . 313
13.4 Desire lines . . . 317
13.5 Nodes . . . 320
13.6 Routes . . . 322
13.6.1 Routes, legs and segments . . . 323
13.6.2 In-memory routing with R . . . 324
13.6.3 Locally hosted dedicated routing engines . . . 324
13.6.4 Remotely hosted dedicated routing engines . . . 325
13.6.5 Contraction hierarchies and traffic assigment . . . 325
13.6.6 Routing: A worked example . . . 326
13.7 Route networks . . . 328
13.8 Prioritizing new infrastructure . . . 332
13.9 Future directions of travel . . . 333
13.10 Exercises . . . 335

14 Geomarketing **337**
14.1 Introduction . . . 337
14.2 Case study: bike shops in Germany . . . 338
14.3 Tidy the input data . . . 339
14.4 Create census rasters . . . 339
14.5 Define metropolitan areas . . . 342
14.6 Points of interest . . . 344
14.7 Identify suitable locations . . . 347
14.8 Discussion and next steps . . . 347
14.9 Exercises . . . 349

15 Ecology **351**
15.1 Introduction . . . 351

15.2 Data and data preparation . . . 353
15.3 Reducing dimensionality . . . 357
15.4 Modeling the floristic gradient . . . 360
15.4.1 **mlr3** building blocks . . . 361
15.4.2 Predictive mapping . . . 364
15.5 Conclusions . . . 366
15.6 Exercises . . . 367

16 Conclusion **369**
16.1 Introduction . . . 369
16.2 Package choice . . . 369
16.3 Gaps and overlaps . . . 372
16.4 Getting help . . . 373
16.4.1 Searching for solutions online . . . 374
16.4.2 Places to search for (and ask) for help . . . 374
16.4.3 Reproducible examples with **reprex** . . . 375
16.4.4 Defining and sketching the problem . . . 376
16.5 Where to go next? . . . 376
16.6 The open source approach . . . 378

Bibliography **381**

Index **395**

Foreword (2nd Edition)

Writing books about open source data science software that constantly changes in uncontrolled ways is a brave undertaking: it feels like running a race while someone else constantly moves the finish line. This second edition of *Geocomputation with R* is timely: it not only catches up with many recent changes, but also embraces new R packages, and new topical developments in the computing landscape. It now includes a chapter on raster-vector interactions, discussing the package **terra** which is replacing package **raster** for raster (and vector) data processing. It also keeps up with the **tmap** package for creating high quality maps, which is completing a full rewrite cycle.

Besides updating the contents of this book, the authors have also been very active in helping to streamline and focus those changes in software by extensively testing it, helping improve it, writing issues and pull requests on GitHub, sharing benchmark results, and helping to improve software documentation.

The first edition of this book has been a great success. It was the first book to popularize spatial analysis with the **sf** package and **tidyverse**. Its enthusiastic tone reached a wide audience, and helped people at various levels of experience solving new problems and moving to their next level. Being available entirely freely online in addition to the printed volume gave it a large reach, and enabled users to try out the presented methodology on their own datasets. In addition to that, the authors have encouraged the readership to reach out by ways of GitHub issues, social media posts, and discussions in a discord channel. This has led to 75 people contributing to the book's source code in one way or the other, including several providing longer reviews or contributing full sections, including on Cloud-optimized GeoTIFFs, STAC and openEO; the **sfheaders** package; OGC APIs and metadata; and the `CycleHire` shiny app. on Discord, it has led to lively and spontaneous discussions in threads that include topics ranging from highly technical to "look what I built".

Beyond this, the authors have initiated the companion volume *Geocomputation with Python*, stressing that geocomputation happens with data science languages, and is by no means restricted to one of them. Geocomputation is on the rise, and as part of fostering a growing geocomputation community, writing books like this one is indispensable.

Edzer Pebesma

Münster, Germany,
May 2024

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

Preface

Who this book is for

This book is for people who want to analyze, visualize and model geographic data with open source software. It is based on R, a statistical programming language that has powerful data processing, visualization and geospatial capabilities. The book covers a wide range of topics and will be of interest to a wide range of people from many different backgrounds, especially:

- People who have learned spatial analysis skills using a desktop Geographic Information System (GIS), such as QGIS, ArcGIS, GRASS GIS or SAGA, who want access to a powerful (geo)statistical and visualization programming language and the benefits of a command line approach (Sherman, 2008):

> With the advent of 'modern' GIS software, most people want to point and click their way through life. That's good, but there is a tremendous amount of flexibility and power waiting for you with the command line.

- Graduate students and researchers from fields specializing in geographic data including Geography, Remote Sensing, Planning, GIS and Spatial Data Science
- Academics and post-graduate students working with geographic data — in fields such as Geology, Regional Science, Biology and Ecology, Agricultural Sciences, Archaeology, Epidemiology, Transport Modeling, and broadly defined Data Science — who require the power and flexibility of R for their research
- Applied researchers and analysts in public, private or third-sector organizations who need the reproducibility, speed and flexibility of a command

line language such as R in applications dealing with spatial data as diverse as Urban and Transport Planning, Logistics, Geo-marketing (store location analysis) and Emergency Planning

The book is designed for intermediate-to-advanced R users interested in geocomputation and R beginners who have prior experience with geographic data. If you are new to both R and geographic data, do not be discouraged: we provide links to further materials and describe the nature of spatial data from a beginner's perspective in Chapter 2 and in links provided below.

How to read this book

The book is divided into three parts:

1. Part I: Foundations, aimed at getting you up-to-speed with geographic data in R.
2. Part II: Advanced techniques, including spatial data visualization, bridges to GIS software, programming with spatial data, and statistical learning.
3. Part III: Applications to real-world problems, including transportation, geomarketing and ecological modeling.

The chapters get harder from one part to the next. We recommend reading all chapters in Part I in order before tackling the more advanced topics in Part II and Part III. The chapters in Part II and Part III benefit slightly from being read in order, but can be read independently if you are interested in a specific topic. A major barrier to geographical analysis in R is its steep learning curve. The chapters in Part I aim to address this by providing reproducible code on simple datasets that should ease the process of getting started.

An important aspect of the book from a teaching/learning perspective is the **exercises** at the end of each chapter. Completing these will develop your skills and equip you with the confidence needed to tackle a range of geospatial problems. Solutions to the exercises can be found in an online booklet that accompanies Geocomputation with R, hosted at r.geocompx.org/solutions. To learn how this booklet was created, and how to update solutions in files such as _01-ex.Rmd, see our blog post on Geocomputation with R solutions. More blog posts and examples can be found at geocompx.org.

Impatient readers are welcome to dive straight into the practical examples, starting in Chapter 2. However, we recommend reading about the wider context of *Geocomputation with R* in Chapter 1 first. If you are new to R, we also recommend learning more about the language before attempting to run the

code chunks provided in each chapter (unless you're reading the book for an understanding of the concepts). Fortunately for beginners, R has a supportive community that has developed a wealth of resources that can help. We particularly recommend three tutorials: R for Data Science (Grolemund and Wickham, 2016) Efficient R Programming (Gillespie and Lovelace, 2016), and An introduction to R (R Core Team, 2021).

Why R?

Although R has a steep learning curve, the command line approach advocated in this book can quickly pay off. As you'll learn in subsequent chapters, R is an effective tool for tackling a wide range of geographic data challenges. We expect that, with practice, R will become the program of choice in your geospatial toolbox for many applications. Typing and executing commands at the command line is, in many cases, faster than pointing-and-clicking around the graphical user interface (GUI) of a desktop GIS. For some applications such as Spatial Statistics and modeling, R may be the *only* realistic way to get the work done.

As outlined in Section 1.3, there are many reasons for using R for geocomputation: R is well suited to the interactive use required in many geographic data analysis workflows compared with other languages. R excels in the rapidly growing fields of Data Science (which includes data carpentry, statistical learning techniques and data visualization) and Big Data (via efficient interfaces to databases and distributed computing systems). Furthermore, R enables a reproducible workflow: sharing scripts underlying your analysis will allow others to build on your work. To ensure reproducibility in this book, we have made its source code available at github.com/geocompx/geocompr. There you will find script files in the `code/` folder that generate figures: when code generating a figure is not provided in the main text of the book, the name of the script file that generated it is provided in the caption (see for example the caption for Figure 13.2).

Other languages such as Python, Java and C++ can be used for geocomputation. There are excellent resources for learning geocomputation *without R*, as discussed in Section 1.4. None of these provide the unique combination of package ecosystem, statistical capabilities, and visualization options offered by the R community. Furthermore, by teaching how to use one language (R) in depth, this book will equip you with the concepts and confidence needed to do geocomputation in other languages.

Real-world impact

Geocomputation with R will equip you with knowledge and skills to tackle a wide range of issues, including those with scientific, societal and environmental implications, manifested in geographic data. As described in Section 1.1, geocomputation is not only about using computers to process geographic data, it is also about real-world impact. The wider context and motivations underlying this book are covered in Chapter 1.

Acknowledgments

Many thanks to everyone who contributed directly and indirectly via the code hosting and collaboration site GitHub, including the following people who contributed direct via pull requests: prosoitos, tibbles-and-tribbles, florisvdh, babayoshihiko, katygregg, Lvulis, rsbivand, iod-ine, KiranmayiV, cuixueqin, defuneste, smkerr, zmbc, marcosci, darrellcarvalho, dcooley, FlorentBedecarratsNM, erstearns, appelmar, MikeJohnPage, eyesofbambi, krystof236, nickbearman, tylerlittlefield, sdesabbata, howardbaik, edzer, pat-s, giocomai, KHwong12, LaurieLBaker, eblondel, MarHer90, mdsumner, ahmohil, richfitz, VLucet, wdearden, yihui, adambhouston, chihinl, cshancock, e-clin, ec-nebi, gregor-d, jasongrahn, p-kono, pokyah, schuetzingit, tim-salabim, tszberkowitz, vlarmet, ateucher, annakrystalli, andtheWings, kant, gavinsimpson, Himanshuteli, yutannihilation, jimr1603, jbixon13, jkennedyie, olyerickson, yvkschaefer, katiejolly, kwhkim, layik, mpaulacaldas, mtennekes, mvl22, and ganes1410. Thanks to Marco Sciaini who created the front cover image for the first edition and to Benjamin Nowak who created the cover image for the second edition. See `code/frontcover.R` and `code/frontcover2.R` for the reproducible code that generated these visualizations. Dozens more people contributed online, by raising and commenting on issues, and by providing feedback via social media. The `#geocompr` and `geocompx` hashtags will live on!

We would like to thank John Kimmel and Lara Spieker from CRC Press and Taylor & Francis for taking our ideas from an early book plan into production via four rounds of peer review for each edition. The reviewers deserve special mention here for their detailed feedback and expertise substantially improved the book's structure and content.

We thank Patrick Schratz and Alexander Brenning from the University of Jena for fruitful discussions on and contributions to Chapters 12 and 15. We thank Emmanuel Blondel from the Food and Agriculture Organization of the United Nations for expert contributions to the section on web services;

Michael Sumner for critical contributions to many areas of the book, especially the discussion of algorithms in Chapter 11; Tim Appelhans, David Cooley and Kiranmayi Vadlamudi for key contributions to the visualization chapter (Chapter 9); Marius Appel for his contributions to Chapter 10; and Katy Gregg, who proofread every chapter and greatly improved the readability of the book.

Countless others could be mentioned who contributed in myriad ways. The final thank you is for all the software developers who make geocomputation with R possible. Especially, Edzer Pebesma (who created the **sf** package), Robert Hijmans (who created **terra**) and Roger Bivand (who laid the foundations for much R-spatial software) who have made high performance geographic computing possible in R.

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

1

Introduction

This book is about using the power of computers to *do things* with geographic data. It teaches a range of spatial skills, including: reading, writing and manipulating geographic file formats; making static and interactive maps; and applying geocomputation to support more evidence-based decision-making related to a range of geographic phenomena, from transport systems to ecosystems. By demonstrating how various geographic operations can be linked, in 'code chunks' that intersperse the prose, the book also teaches reproducible, open and thus scientific workflows.

The book is not just about using the wealth of *existing tools* for geocomputation: it's also about understanding the geographic data structures and software needed to build *new tools*. The approach we teach throughout, and programming techniques covered in Chapter 11 in particular, can remove constraints on your creativity imposed by software. After reading the book and completing the exercises, you should be ready to tackle real-world problems, communicate your work in maps and code, and contribute to the open source communities developing tools and documentation for reproducible geocomputation.

Over the last few decades, free and open source software for geospatial (FOSS4G) has progressed at an astonishing rate. Thanks to organizations such as OSGeo, advanced geographic techniques are no longer the preserve of those with expensive hardware and software: anyone can now download and run high-performance software for geocomputation. Open source Geographic Information Systems (GIS), such as QGIS, have made geographic analysis accessible worldwide. GIS software products are powerful, but they tend to emphasize a graphical user interface (GUI) approach over the command-line interface (CLI) approach advocated in this book. The 'GUI focus' of many GIS products has the unintended consequence of disabling many users from making their work fully reproducible, a problem that can be overcome by calling 'geoalgorithms' contained in GIS software from the command line, as we'll see in Chapter 10. A simplistic comparison between the different approaches is illustrated in Table 1.1.

R is not the only language providing a CLI for geocomputation. Other command environments with powerful geographic capabilities exist, including Python (covered in the book Geocomputation with Python), Julia, and

TABLE 1.1 Differences in emphasis between software packages (Graphical User Interface (GUI) of Geographic Information Systems (GIS) and R).

Attribute	Desktop GIS (GUI)	R
Home disciplines	Geography	Computing, Statistics
Software focus	Graphical User Interface	Command line
Reproducibility	Minimal	Maximal

JavaScript. However, R has advantages that make it a good language for learning geocomputation and for many geocomputation tasks, especially for statistics, modeling and visualization, as outlined in Section 1.2.

This book is also motivated by the importance of reproducibility for scientific research. It aims to make reproducible geographic data analysis workflows more accessible, and demonstrate the power of open geospatial software available from the command line. R provides ways to interface with other languages (Eddelbuettel and Balamuta, 2018), enabling numerous spatial software libraries to be called from R, as explained in Section 1.3 and demonstrated in Chapter 10. Before going into the details of the software, however, it is worth taking a step back and thinking about what we mean by geocomputation.

Reproducibility is a major advantage of command-line interfaces, but what does it mean in practice? We define it as follows: "A process in which the same results can be generated by others using publicly accessible code". This may sound simple and easy to achieve (which it is if you carefully maintain your R code in script files), but it has profound implications for teaching and the scientific process (Pebesma et al., 2012).

1.1 What is geocomputation?

We define geocomputation as

> Academic research, software development and practical applications that use geographic data to solve problems, with a focus on reproducibility, flexibility and tool development.

Geocomputation is a young term, dating back to the first conference on the subject in 1996.[1] What distinguished geocomputation from the (at the time) commonly used term 'quantitative geography' was its emphasis on "creative and experimental" applications (Longley et al., 1998) and the development of new tools and methods. In the words of Stan Openshaw, a pioneer in the field who was an advocate (and possibly originator) of the term, "GeoComputation is about using the various different types of geodata and about developing relevant geo-tools within the overall context of a 'scientific' approach" (Openshaw and Abrahart, 2000). Building on this early definition, *Geocomputation with R* goes beyond data analysis and modeling to include the development of new tools and methods for work that is not just interesting academically but beneficial.

Our approach differs from early definitions of geocomputation in one important way, however: in its emphasis on reproducibility and collaboration. At the turn of the 21st Century, it was unrealistic to expect readers to be able to reproduce code examples, due to barriers preventing access to the necessary hardware, software and data. Fast-forward to today and things have progressed rapidly. Anyone with access to a laptop with sufficient RAM (at least eight GB recommended) can install and run software for geocomputation, and reproduce the contents of this book. Financial and hardware barriers to geocomputation that existed in 1990s and early 2000s, when high-performance computers were too expensive for most people, have been removed.[2] Geocomputation is also more accessible because publicly accessible datasets are more widely available than ever before, as we will see in Chapter 8. Unlike early works in the field, all the work presented in this book is reproducible using code and example data supplied alongside the book, in R packages such as **spData**, the installation of which is covered in Chapter 2.

Geocomputation is closely related to other terms including: Geographic Information Science (GIScience); Geomatics; Geoinformatics; Spatial Information Science; Geoinformation Engineering (Longley, 2015); and Spatial Data Science (SDS). Each term shares an emphasis on a 'scientific' (implying reproducible and falsifiable) approach influenced by GIS, although their origins and main fields of application differ. SDS, for example, emphasizes 'data science' skills and large datasets, while Geoinformatics tends to focus on data structures. But the overlaps between the terms are larger than the differences between them and we use geocomputation as a rough synonym encapsulating all of them: they all seek to use geographic data for applied scientific work.

[1] The first 'GeoComputation' conference took place at the University of Leeds, where one of the authors (Robin) is currently based. In 2017 the GeoComputation conference returned to University of Leeds, providing a chance for us to work on and present the book (see www.geocomputation.org for more on the conference series, and papers/presentations spanning more than two decades).

[2] A suitable laptop can be acquired second-hand for $100 or less in most countries today from websites such as Ebay. Guidance on installing R and a suitable code editor is provided in Chapter 2.

Unlike early users of the term, however, we do not seek to imply that there is any cohesive academic field called 'Geocomputation' (or 'GeoComputation' as Stan Openshaw called it).

Geocomputation is a recent term but is influenced by old ideas. It can be seen as a part of Geography, which has a 2000+ year history (Talbert, 2014); and an extension of GIS (Neteler and Mitasova, 2008), which emerged in the 1960s (Coppock and Rhind, 1991).

Geography has played an important role in explaining and influencing humanity's relationship with the natural world long before the invention of the computer. The famous explorer, early geographer and pioneering polymath Alexander von Humboldt (who has dozens of species, geographic features, places and even universities named after him, such was his influence) illustrates this role: not only did his travels to South America in the early 1800s and resulting observations lay the foundations for physical geography and ecology, they also paved the way towards policies to protect the natural world (Wulf, 2015). This book aims to contribute to the still-evolving 'Geographic Tradition' (Livingstone, 1992) by harnessing the power of modern computers and open source software.

The book's links to older disciplines were reflected in suggested titles for the book: *Geography with R* and *R for GIS*. Each has advantages. The former conveying the applied nature of the content, about more than where something is on the map. The latter communicates that this is a book about using R as a powerful command-line geographic information system, to perform spatial operations on *geographic data*. However, the term GIS has connotations which fail to communicate some of R's greatest strengths: its abilities to seamlessly switch between geographic and non-geographic data processing, modeling and visualization tasks while enabling reproducibility go far beyond the capabilities of GIS. Geocomputation implies working with geographic data in a reproducible code-driven environment and programming new results, methods and tools, which is what this book is all about.

1.2 Why use open source tools for geocomputation?

Early geographers used a variety of tools including barometers, compasses and sextants to advance knowledge about the world (Wulf, 2015). It was only with the invention of the marine chronometer in 1761 that it became possible to calculate longitude at sea, enabling ships to take more direct routes, for example. Before the turn of the century, there was an acute shortage of data and tools for geographic analysis.

Nowadays, researchers and practitioners have no such limitations and in some cases face the opposite problem: too much data and too many tools. Most phones now have a global positioning (GPS) receiver. Sensors ranging from satellites and semi-autonomous vehicles to citizen scientists incessantly measure every part of the world. The rate of data produced can be overwhelming, with emerging technologies such as autonomous vehicles generating hundreds or even thousands of gigabytes of data daily. Remote sensing datasets from satellites are too large to analyze with a single computer, as outlined in Chapter 10. This 'geodata revolution' drives demand for high performance computer hardware and efficient, scalable software to handle and extract signal from the noise. Evolving open source tools can import and process subsets from the vast geographic data stores directly, via application programming interfaces (APIs) and via interfaces to databases.

With the rapidly changing hardware, software and data landscapes, it's important to choose tools that are future-proof. A major advantage of open source software is its **rate of development and longevity**, with thousands of potential contributors. Hundreds of people submit bug reports and suggest new features as well as documentation improvements to open source projects every day — a rate of evolution that most proprietary solutions simply cannot keep up with.

A linked advantage is **interoperability**. While proprietary products tend to be monolithic 'empires' that are difficult to maintain (linked to the previously mentioned advantage), open source software is more like a 'federation' of modular tools that can be combined in different ways. This has allowed open source data science languages such as R to rapidly incorporate new developments such as interfaces to high performance visualization libraries and file formats, while proprietary solutions struggle to keep up.

Another major advantage is **reproducibility**. Being able to replicate findings is vital for scientific research, and open source software removes an important barrier of reproducibility by enabling others to check your findings or applying your methods in new contexts using the same tools. The combination of using tools that can be accessed by anyone for free with the ability to share code and data means that the results of your work can be checked and built upon by others, which is a huge advantage if you want your work to be used and cited.

The biggest advantage of open source software combined with sharing of reproducible code for many people, however, is the **community**. The community enables you to get support far quicker and often of higher quality than is possible with a centralized and budget-limited support team associated with proprietary software. The community can provide feedback, ideas and, as discussed in the Chapter 16), can help you to develop your own tools and methods.

R is an open source software project, a powerful language, and an ever-evolving community of statisticians and developers (Wickham, 2019). R is not the only

language enabling reproducible geocomputation with open source software, as outlined in Section 1.4). Many of the reasons for using R also apply to other open source languages for reproducible data science, such as Python and Julia. However, R has some key advantages, as outlined in Section 1.3.

1.3 Why use R for geocomputation?

R is a multi-platform, open source language and environment for statistical computing and graphics (r-project.org/). With a wide range of packages, R also supports advanced geospatial statistics, modeling and visualization. Integrated development environments (IDEs) such as RStudio have made R more user-friendly for many, easing map-making with a panel dedicated to interactive visualization.

At its core, R is an object-oriented, functional programming language (Wickham, 2019) and was specifically designed as an interactive interface to other software (Chambers, 2016). The latter also includes many 'bridges' to a treasure trove of GIS software, 'geolibraries' and functions (see Chapter 10). It is thus ideal for quickly creating 'geo-tools', without needing to master lower level languages (compared to R) such as C, FORTRAN or Java (see Section 1.4). This can feel like breaking free from the metaphorical 'glass ceiling' imposed by GUI-based or proprietary geographic information systems (see Table 1.1 for a definition of GUI). Furthermore, R facilitates access to other languages: the packages **Rcpp** and **reticulate** enable access to C++ and Python code, for example. This means R can be used as a 'bridge' to a wide range of geospatial programs (see Section 1.4).

Another example showing R's flexibility and evolving geographic capabilities is interactive map-making. As we'll see in Chapter 9, the statement that R has "limited interactive [plotting] facilities" (Bivand et al., 2013) is no longer true. This is demonstrated by the following code chunk, which creates Figure 1.1 (the functions that generate the plot are covered in Section 9.4).

```
library(leaflet)
popup = c("Robin", "Jakub", "Jannes")
leaflet() |>
  addProviderTiles("NASAGIBS.ViirsEarthAtNight2012") |>
  addMarkers(lng = c(-3, 23, 11),
             lat = c(52, 53, 49),
             popup = popup)
```

It would have been difficult to produce Figure 1.1 using R (or any open source language for data science) a few years ago, let alone as an interactive map.

FIGURE 1.1 The blue markers indicate where the authors are from. The basemap is a tiled image of the Earthat night provided by NASA. Interact with the online version at r.geocompx.org, for example by zooming in and clicking on the pop-ups.

This illustrates R's flexibility and how, thanks to developments such as **knitr** and **leaflet**, it can be used as an interface to other software, a theme that will recur throughout this book. The use of R code, therefore, enables teaching geocomputation with reference to reproducible examples representing real-world phenomena, rather than just abstract concepts.

The 'R-spatial stack' is easy to install and has comprehensive, well-maintained and highly interoperable packages. R has 'batteries included' with statistical functions as part of the base installation and hundreds of well-maintained packages implementing many cutting edge methods. With R, you can dive and get things working with surprisingly few lines of code, enabling you to focus on the geographic methods and data, rather than debugging and managing package dependencies. A particular strength of R is the ease with which it allows you to create publication quality interactive maps thanks to excellent mapping packages, as outlined in Chapter 9.

1.4 Software for geocomputation

R is a powerful language for geocomputation, but there are many other options for geographic data analysis providing thousands of geographic functions. Awareness of other languages for geocomputation will help decide when a different tool may be more appropriate for a specific task, and will place R in

the wider geospatial ecosystem. This section briefly introduces the languages C++, Java and Python for geocomputation, in preparation for Chapter 10.

An important feature of R (and Python) is that it is an interpreted language. This is advantageous because it enables interactive programming in a Read–Eval–Print Loop (REPL): code entered into the console is immediately executed and the result is printed, rather than waiting for the intermediate stage of compilation. On the other hand, compiled languages such as C++ and Java tend to run faster (once they have been compiled).

C++ provides the basis for many GIS packages such as QGIS, GRASS GIS and SAGA, so it is a sensible starting point. Well-written C++ is very fast, making it a good choice for performance-critical applications such as processing large geographic datasets, but is harder to learn than Python or R. C++ has become more accessible with the **Rcpp** package, which provides a good 'way in' to C programming for R users. Proficiency with such low-level languages opens the possibility of creating new, high-performance 'geoalgorithms' and a better understanding of how GIS software works (see Chapter 11). However, it is not necessary to learn C++ to use R for geocomputation.

Python is an important language for geocomputation, especially because many Desktop GIS such as GRASS GIS, SAGA and QGIS provide a Python API (see Chapter 10). Like R, Python is a popular language for data science. Both languages are object-oriented, and have many areas of overlap, leading to initiatives such as the **reticulate** package that facilitates access to Python from R and the Ursa Labs initiative to support portable libraries to the benefit of the entire open source data science ecosystem.

In practice both R and Python have their strengths. To some extent which you use is less important than the domain of application and communication of results. Learning either will provide a head-start in learning the other. However, there are major advantages of R over Python for geocomputation. This includes its much better support of the geographic raster data model in the language itself (see Chapter 2) and corresponding visualization possibilities (see Chapters 2 and 9). Equally important, R has unparalleled support for statistics, including spatial statistics, with hundreds of packages (unmatched by Python) supporting thousands of statistical methods.

The major advantage of Python is that it is a *general-purpose* programming language. It is used in many domains, including desktop software, computer games, websites and data science. Python is often the only shared language between different (geocomputation) communities and can be seen as the 'glue' that holds many GIS programs together. Many geoalgorithms, including those in QGIS and ArcMap, can be accessed from the Python command line, making it well suited as a starter language for command line GIS.[3]

[3]Python modules providing access to geoalgorithms include `grass.script` for GRASS GIS, `saga-python` for SAGA-GIS, `processing` for QGIS and `arcpy` for ArcGIS.

For spatial statistics and predictive modeling, however, R is second-to-none. This does not mean you must choose either R or Python: Python supports most common statistical techniques (though R tends to support new developments in spatial statistics earlier) and many concepts learned from Python can be applied to the R world. Like R, Python also supports geographic data analysis and manipulation with packages such as **shapely**, **geopandas**, **rasterio** and **xarray**.

1.5 R's spatial ecosystem

There are many ways to handle geographic data in R, with dozens of packages in the area.[4] In this book, we endeavor to teach the state-of-the-art in the field whilst ensuring that the methods are future-proof. Like many areas of software development, R's spatial ecosystem is rapidly evolving (Figure 1.2). Because R is open source, these developments can easily build on previous work, by 'standing on the shoulders of giants', as Isaac Newton put it in 1675. This approach is advantageous because it encourages collaboration and avoids 'reinventing the wheel'. The package **sf** (covered in Chapter 2), for example, builds on its predecessor **sp**.

A surge in development time (and interest) in 'R-spatial' has followed the award of a grant by the R Consortium for the development of support for *simple features*, an open-source standard and model to store and access vector geometries. This resulted in the **sf** package (covered in Section 2.2.1). Multiple places reflect the immense interest in **sf**. This is especially true for the R-sig-Geo Archives, a long-standing open access email list containing much R-spatial wisdom accumulated over the years.

It is noteworthy that shifts in the wider R community, as exemplified by the data processing package **dplyr** (released in 2014), influenced shifts in R's spatial ecosystem. Alongside other packages that have a shared style and emphasis on 'tidy data' (including, e.g., **ggplot2**), **dplyr** was placed in the **tidyverse** 'metapackage' in late 2016. The **tidyverse** approach, with its focus on long-form data and fast intuitively named functions, has become immensely popular. This has led to a demand for 'tidy geographic data' which has been partly met by **sf**. An obvious feature of the **tidyverse** is the tendency for packages to work in harmony. There is no equivalent 'geoverse', but the modern R-spatial ecosystem has consolidated around **sf**, as illustrated by key packages that depend on it shown in Table 1.2, and **terra**, both of which are taught in

[4] An overview of R's spatial ecosystem can be found in the CRAN Task View on the Analysis of Spatial Data (see https://cran.r-project.org/view=Spatial).

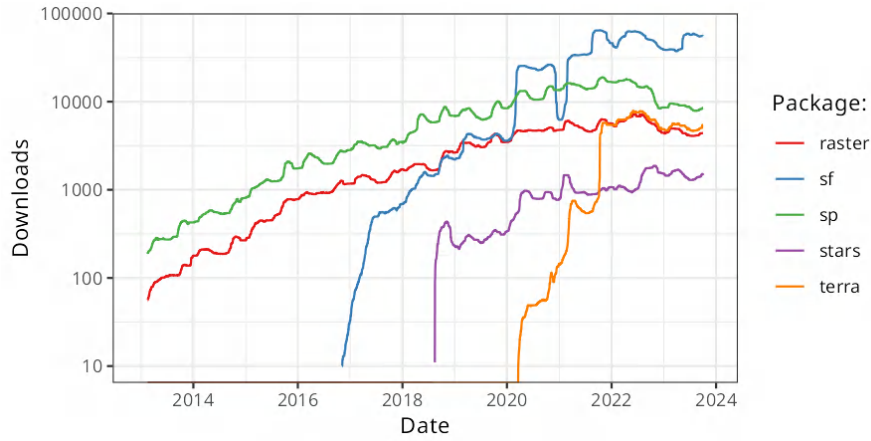

FIGURE 1.2 Downloads of selected R packages for working with geographic data from early 2013 to present. The y axis shows the average number of daily downloads from the popular cloud.r-project.org CRAN mirror with a 91-day rolling window (log scale).

TABLE 1.2 The top 5 most downloaded packages that depend on sf, in terms of average number of downloads per day over the previous month. As of 2023-11-14 , there are 526 packages which import sf.

Package	Downloads
r5r	4774
stars	1354
leafem	1013
spdep	881
tmap	873

this book. The stack is highly interoperable both between packages and with other languages, as outlined in Chapter 10.

1.6 History of R-spatial

There are many benefits of using modern spatial packages such as **sf**, but there is value in understanding the history of R's spatial capabilities. Many functions, use cases and teaching materials are contained in older packages, many of which are still useful, provided you know where to look.

R's spatial capabilities originated in early spatial packages in the S language (Bivand and Gebhardt, 2000). The 1990s saw the development of numerous S scripts and a handful of packages for spatial statistics. By the year 2000, there were R packages for various spatial methods, including "point pattern analysis, geostatistics, exploratory spatial data analysis and spatial econometrics" (Bivand and Neteler, 2000). Some of these, notably **spatial**, **sgeostat** and **splancs** are still available on CRAN (Rowlingson and Diggle, 1993, 2017; Venables and Ripley, 2002; Majure and Gebhardt, 2016). Key spatial packages were described in Ripley (2001), which outlined R packages for spatial smoothing and interpolation and point pattern analysis. One of these (**spatstat**) is still being actively maintained, more than 20 years after its first release.

A following commentary outlined the future prospects of spatial statistics (Bivand, 2001), setting the stage for the development of the popular **spdep** package (Bivand, 2017). Notably, the commentary mentioned the need for standardization of spatial interfaces, efficient mechanisms for exchanging data with GIS, and handling of spatial metadata such as coordinate reference systems (CRS). These aims have largely been achieved.

maptools (Bivand and Lewin-Koh, 2017) is another important package from this time, which provided an interface to the shapelib library for reading the Shapefile file format and which fed into **sp**. An extended review of spatial packages proposed a class system to support the "data objects offered by GDAL", including fundamental point, line, polygon, and raster types, and interfaces to external libraries (Bivand, 2003). To a large extent, these ideas were realized in the packages **rgdal** and **sp**, providing a foundation for the seminal book *Applied Spatial Data Analysis with R* (ASDAR) (Bivand et al., 2013), first published in 2008. R's spatial capabilities have evolved substantially since then, but they still build on the ideas of early pioneers. Interfaces to GDAL and PROJ, for example, still power R's high-performance geographic data I/O and CRS transformation capabilities, as outlined in Chapters 7 and 8, respectively.

rgdal, released in 2003, provided GDAL bindings for R which greatly enhanced its ability to import data from previously unavailable geographic data formats. The initial release supported only raster drivers, but subsequent enhancements provided support for CRSs (via the PROJ library), reprojections and import of vector file formats. Many of these additional capabilities were developed by Barry Rowlingson and released in the **rgdal** codebase in 2006, as described in Rowlingson et al. (2003) and the R-help email list.

The **sp** package, released in 2005, was a significant advancement in R's spatial capabilities. It introduced classes and generic methods for handling geographic coordinates, including points, lines, polygons, and grids, as well as attribute data. With the S4 class system, **sp** stores information such as bounding box, coordinate reference system (CRS), and attributes in slots within `Spatial` objects. This allows for efficient data operations on geographic data. The package

also provided generic methods like `summary()` and `plot()` for working with geographic data.

In the following decade, **sp** classes rapidly became popular for geographic data in R and the number of packages that depended on it increased from around 20 in 2008 to over 100 in 2013 (Bivand et al., 2013). By 2019 more than 500 packages imported **sp**. Although the number of packages that depend on **sp** has decreased since the release of **sf** it is still used by prominent R packages, including **gstat** (for spatial and spatiotemporal geostatistics) and **geosphere** (for spherical trigonometry) (Pebesma and Graeler, 2023; Hijmans, 2016).

While **rgdal** and **sp** solved many spatial issues, it was not until **rgeos** was developed during a Google Summer of Code project in 2010 (Bivand and Rundel, 2023) that geometry operations could be undertaken on **sp** objects. Functions such as `gIntersection()` enabled users to find spatial relationships between geographic objects and to modify their geometries (see Chapter 5 for details on geometric operations with **sf**).

A limitation of the **sp** ecosystem was its limited support for raster data. This was overcome by **raster**, first released in 2010 (Hijmans, 2023b). **raster**'s class system and functions enabled a range of raster operations, capabilities now implemented in the **terra** package, which supersedes **raster**, as outlined in Section 2.3. An important capability of **raster** and **terra** is their ability to work with datasets that are too large to fit into RAM by supporting off-disk operations. **raster** and **terra** also supports map algebra, as described in Section 4.3.2.

In parallel with these developments of class systems and methods, came the support for R as an interface to dedicated GIS software. **GRASS** (Bivand, 2000) and follow-on packages **spgrass6**, **rgrass7** and **rgrass** were prominent examples in this direction (Bivand, 2016a,b, 2023). Other examples of bridges between R and GIS include bridges to QGIS via **qgisprocess** (Dunnington et al., 2024), SAGA via **Rsagacmd** (Pawley, 2023) or **RSAGA** (Brenning et al., 2022) and ArcGIS via **RPyGeo** (Brenning, 2012a, first published in 2008), and more (see Chapter 10).

Visualization was not a focus initially, with the bulk of R-spatial development focused on analysis and geographic operations. **sp** provided methods for map-making using both the base and lattice plotting system, but demand was growing for advanced map-making capabilities. **RgoogleMaps** first released in 2009, allowed to overlay R spatial data on top of 'basemap' tiles from online services such as Google Maps or OpenStreetMap (Loecher and Ropkins, 2015). It was followed by the **ggmap** package that added similar 'basemap' tiles capabilities to **ggplot2** (Kahle and Wickham, 2013). Though **ggmap** facilitated map-making with **ggplot2**, its utility was limited by the need to `fortify` spatial objects, which means converting them into long data frames. While this works well for points, it is computationally inefficient for lines and

polygons, since each coordinate (vertex) is converted into a row, leading to huge data frames to represent complex geometries. Although geographic visualization tended to focus on vector data, raster visualization was supported in **raster** and received a boost with the release of **rasterVis** (Lamigueiro, 2018). Since then map-making in R has become a hot topic, with dedicated packages such as **tmap**, **leaflet** and **mapview** gaining popularity, as highlighted in Chapter 9.

Since 2018, when the First Edition of Geocomputation with R was published, the development of geographic R packages has accelerated. **terra**, a successor of the **raster** package, was firstly released in 2020 (Hijmans, 2023c), bringing several benefits to R users working with raster datasets: it is faster and has more a straightforward user interface than its predecessor, as described in Section 2.3.

In mid-2021, **sf** started using the S2 spherical geometry engine for geometry operations on unprojected datasets, as described in Section 2.2.9. Additional ways of representing and working with geographic data in R since 2018 have been developed, including with the **stars** and **lidR** packages (Pebesma, 2021; Roussel et al., 2020).

Such developments have been motivated by the emergence of new technologies, standards and software outside of the R environment (Bivand, 2021). Major updates to the PROJ library beginning in 2018 forced the replacement of 'proj-string' representations of CRSs with 'Well Known Text', as described in Section 2.4 and Chapter 7.

Since the publication of the first version of Geocomputation with R in 2018, several packages for spatial data visualization have been developed and improved. The **rayshader** package, for example, enables the development of striking and easy-to-animate 3D visualizations via raytracing and multiple hill-shading methods (Morgan-Wall, 2021). The very popular **ggplot2** package gained new spatial capabilities, thanks to work on the **ggspatial** package, which provides scale bars and north arrows (Dunnington, 2021). **gganimate** enables smooth and customizable spatial animations (Pedersen and Robinson, 2020).

Existing visualization packages have also been improved or rewritten. Large raster objects are automatically downscaled in **tmap** and high-performance interactive maps are now possible thanks to packages including **leafgl** and **mapdeck**. The **mapsf** package (successor of **cartography**) was rewritten to reduce dependencies and improve performance (Giraud, 2021); and **tmap** underwent a major update in Version 4, in which most of the internal code was revised.

In late 2021, the planned retirement of **rgdal**, **rgeos** and **maptools** was announced and in October 2023 they were archived on CRAN. This retirement at the end of 2023 not only has had a large impact on existing workflows

applying these packages, but also influenced the packages that depend on them. Modern R packages such as **sf** and **terra**, described in Chapter 2 provide a strong and future-proof foundation for geocomputation that we build on in this book.

1.7 Exercises

E1. Think about the terms 'GIS', 'GDS' and 'geocomputation' described above. Which (if any) best describes the work you would like to do using geo* methods and software and why?

E2. Provide three reasons for using a scriptable language such as R for geocomputation instead of using a graphical user interface (GUI) based GIS such as QGIS.

E3. In the year 2000, Stan Openshaw wrote that geocomputation involved "practical work that is beneficial or useful" to others. Think about a practical problem and possible solutions that could be informed with new evidence derived from the analysis, visualization or modeling of geographic data. With a pen and paper (or computational equivalent) sketch inputs and possible outputs illustrating how geocomputation could help.

Part I

Foundations

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

2

Geographic data in R

Prerequisites

This is the first practical chapter of the book, and therefore it comes with some software requirements. You need access to a computer with a recent version of R installed (R 4.3.2 or a later version). We recommend not only reading the prose but also *running the code* in each chapter to build your geocomputational skills.

To keep track of your learning journey, it may be worth starting by creating a new folder on your computer to save your R scripts, outputs and other things related to Geocomputation with R as you go. You can also download or clone the source code underlying the book to support your learning. We strongly recommend using R with an integrated development environment (IDE) such as RStudio (quicker to get up and running) or VS Code (which requires additional setup).

If you are new to R, we recommend following introductory R resources such as Hands on Programming with R and Introduction to R before you dive into Geocomputation with R code. These resources cover in detail how to install R, which simply involves downloading the latest version from the Comprehensive R Archive Network (CRAN). See the note below for more information on installing R for geocomputation on Mac and Linux. Organize your work into projects and give scripts sensible names such as `chapter-02.R` (or equivalent RMarkdown or Quarto file names) to document the code as you learn.

Mac and Linux operating systems (OSs) have additional systems requirements, which can be found in the README of the **sf** package. See also OS-specific instructions such as that provided by the website rtask.thinkr.fr, which covers installing R on the open source OS Ubuntu.

After you have got a good set-up, it's time to run some code! Unless you already have these packages installed, the first thing to do is to

install foundational R packages used in this chapter, with the following commands:[1]

```
install.packages("sf")
install.packages("terra")
install.packages("spData")
install.packages("spDataLarge", repos = "https://geocompr.r-universe.dev")
```

The packages needed to reproduce Part I of this book can be installed with the following command: `remotes::install_github("geocompx/geocompkg")`. This command uses the function `install_packages()` from the **remotes** package to install source code hosted on the GitHub code hosting, version and collaboration platform. The following command will install **all** dependencies required to reproduce the entire book (warning: this may take several minutes): `remotes::install_github("geocompx/geocompkg", dependencies = TRUE)`.

The packages needed to run the code presented in this chapter can be 'loaded' (technically they are attached) with the `library()` function as follows:

```
library(sf)           # classes and functions for vector data
```

The output from `library(sf)` reports which versions of key geographic libraries such as GEOS the package is using, as outlined in Section 2.2.1.

```
library(terra)        # classes and functions for raster data
```

The other packages that were installed contain data that will be used in the book:

```
library(spData)       # load geographic data
library(spDataLarge)  # load larger geographic data
```

2.1 Introduction

This chapter will provide explanations of the fundamental geographic data models: vector and raster. We will introduce the theory behind each data model and the disciplines in which they predominate, before demonstrating their implementation in R.

[1] **spDataLarge** is not on CRAN, meaning it must be installed via *r-universe* or with the following command: `remotes::install_github("Nowosad/spDataLarge")`.

The *vector data model* represents the world using points, lines and polygons. These have discrete, well-defined borders, meaning that vector datasets usually have a high level of precision (but not necessarily accuracy as we will see in Section 2.5). The *raster data model* divides the surface up into cells of constant size. Raster datasets are the basis of background images used in web mapping and have been a vital source of geographic data since the origins of aerial photography and satellite-based remote sensing devices. Rasters aggregate spatially specific features to a given resolution, meaning that they are consistent over space and scalable (many worldwide raster datasets are available).

Which to use? The answer likely depends on your domain of application:

- Vector data tends to dominate the social sciences because human settlements tend to have discrete borders
- Raster dominates many environmental sciences partially because of the reliance on remote sensing data

Both raster and vector datasets are used in many fields and raster and vector datasets can be used together: ecologists and demographers, for example, commonly use both vector and raster data. Furthermore, it is possible to convert between the two forms (see Chapter 6). Whether your work involves more use of vector or raster datasets, it is worth understanding the underlying data model before using them, as discussed in subsequent chapters. This book uses **sf** and **terra** packages to work with vector data and raster datasets, respectively.

2.2 Vector data

Take care when using the word 'vector', as it can have two meanings in this book: geographic vector data and the `vector` class (note the `monospace` font) in R. The former is a data model, the latter is an R class just like `data.frame` and `matrix`. Still, there is a link between the two: the spatial coordinates which are at the heart of the geographic vector data model can be represented in R using `vector` objects.

The geographic vector data model is based on points located within a coordinate reference system (CRS). Points can represent self-standing features (e.g., the location of a bus stop) or they can be linked together to form more complex geometries such as lines and polygons. Most point geometries contain only two dimensions (much less prominent three-dimensional geometries contain an additional z value, typically representing height above sea level).

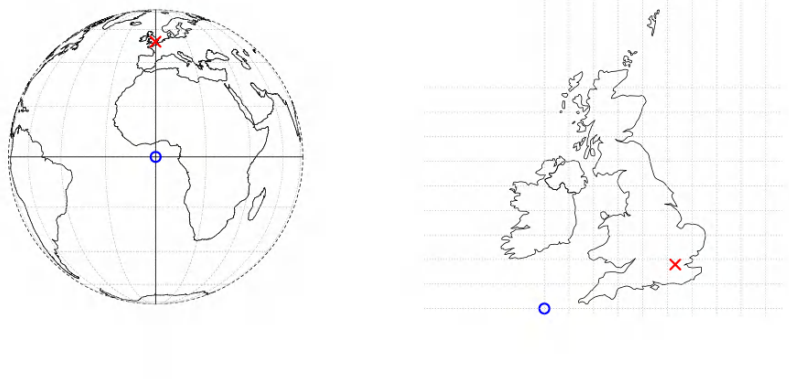

FIGURE 2.1 Vector (point) data in which the location of London (red X) is represented with reference to an origin (blue circle). The left plot represents a geographic CRS with an origin at 0° longitude and latitude. The right plot represents a projected CRS with an origin located in the sea west of the South West Peninsula.

In this system, for example, London can be represented by the coordinates `c(-0.1, 51.5)`. This means that its location is −0.1 degrees east and 51.5 degrees north of the origin. The origin in this case is at 0 degrees longitude (Prime Meridian) and 0 degrees latitude (Equator) in a geographic ('lon/lat') CRS (Figure 2.1, left panel). The same point could also be approximated in a projected CRS with 'Easting/Northing' values of `c(530000, 180000)` in the British National Grid, meaning that London is located 530 km *East* and 180 km *North* of the *origin* of the CRS. This can be verified visually: slightly more than 5 'boxes' — square areas bounded by the gray grid lines 100 km in width — separate the point representing London from the origin (Figure 2.1, right panel).

The location of National Grid's origin, in the sea beyond the South West peninsula, ensures that all locations in the UK have positive Easting and Northing values.[2] There is more to CRSs, as described in Section 2.4 and Chapter 7, For this section, it is sufficient to know that coordinates consist of two numbers representing distance from an origin, usually in x then y dimensions.

The **sf** package provides classes for geographic vector data and a consistent command line interface to important low-level libraries for geocomputation:

[2]The origin we are referring to, depicted in blue in Figure 2.1, is in fact the 'false' origin. The 'true' origin, the location at which distortions are at a minimum, is located at 2° W and 49° N. This was selected by the Ordnance Survey to be roughly in the center of the British landmass longitudinally.

- GDAL, for reading, writing and manipulating a wide range of geographic data formats, covered in Chapter 8
- PROJ, a powerful library for coordinate system transformations, which underlies the content covered in Chapter 7
- GEOS, a planar geometry engine for operations such as calculating buffers and centroids on data with a projected CRS, covered in Chapter 5
- S2, a spherical geometry engine written in C++ developed by Google, via the **s2** package, covered in Section 2.2.9 below and in Chapter 7

Information about these interfaces is printed by **sf** the first time the package is loaded: the message that appears below the `library(sf)` command at the beginning of this chapter tells us the versions of linked GEOS, GDAL and PROJ libraries (these vary between computers and over time) and whether or not the S2 interface is turned on. We may these low-level libraries for granted, but without their tight integration with languages such as R much reproducible geocomputation would be impossible.

A neat feature of **sf** is that you can change the default geometry engine used on unprojected data: 'switching off' S2 can be done with the command `sf::sf_use_s2(FALSE)`, meaning that the planar geometry engine GEOS will be used by default for all geometry operations, including geometry operations on unprojected data. As we will see in Section 2.2.9, planar geometry is based on two-dimensional space. Planar geometry engines such as GEOS assume 'flat' (projected) coordinates, while spherical geometry engines such as S2 assume unprojected (lon/lat) coordinates.

This section introduces **sf** classes in preparation for subsequent chapters (Chapters 5 and 8 cover the GEOS and GDAL interface, respectively).

2.2.1 Introduction to simple features

Simple features is an open standard developed and endorsed by the Open Geospatial Consortium (OGC), a not-for-profit organization whose activities we will revisit in a later chapter (Section 8.2). Simple features is a hierarchical data model that represents a wide range of geometry types. Of 18 geometry types supported by the specification, only seven are used in the vast majority of geographic research (see Figure 2.2); these core geometry types are fully supported by the R package **sf** (Pebesma, 2018).[3]

sf can represent all common vector geometry types (raster data classes are not supported by **sf**): points, lines, polygons and their respective 'multi' versions (which group together features of the same type into a single feature). **sf** also

[3]The full OGC standard includes rather exotic geometry types including 'surface' and 'curve' geometry types, which currently have limited application in real-world applications. You can find the whole list of possible feature types in the PostGIS manual. All 18 types can be represented with the **sf** package, although at the time of writing (2024), plotting only works for the 'core 7'.

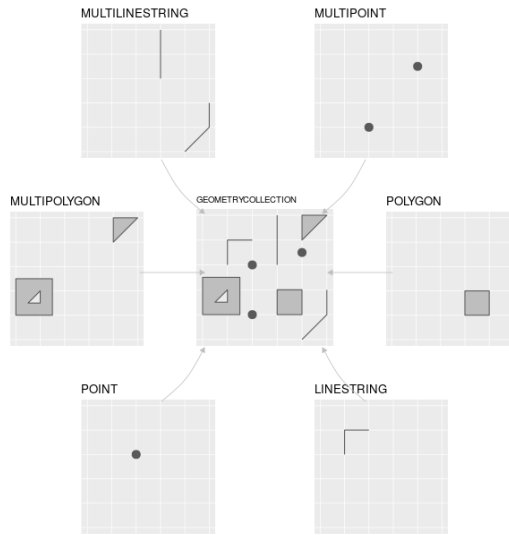

FIGURE 2.2 Simple feature types fully supported by sf.

supports geometry collections, which can contain multiple geometry types in a single object. **sf** provides the same functionality (and more) previously provided in three packages — **sp** for data classes (Pebesma and Bivand, 2023a), **rgdal** for data read/write via an interface to GDAL and PROJ (Bivand et al., 2023a) and **rgeos** for spatial operations via an interface to GEOS (Bivand and Rundel, 2023).

To reiterate the message from Chapter 1, geographic R packages have a long history of interfacing with lower level libraries, and **sf** continues this tradition with a unified interface to recent versions of GEOS for geometry operations, the GDAL library for reading and writing geographic data files, and the PROJ library for representing and transforming projected CRSs. Through **s2**, an R interface to Google's spherical geometry library, `s2`, **sf** also has access to fast and accurate "measurements and operations on non-planar geometries" (Bivand, 2021). Since **sf** version 1.0.0, launched in June 2021, **s2** functionality is now used by default on geometries with geographic (longitude/latitude) coordinate systems, a unique feature of **sf** that differs from spatial libraries that only support GEOS for geometry operations such as the Python package GeoPandas. We will discuss **s2** in subsequent chapters.

sf's ability to integrate multiple powerful libraries for geocomputation into a single framework is a notable achievement that reduces 'barriers to entry' into the world of reproducible geographic data analysis with high-performance libraries. **sf**'s functionality is well documented on its website at r-spatial.github.io/sf/ which contains seven vignettes. These can be viewed offline as follows:

```
vignette(package = "sf") # see which vignettes are available
vignette("sf1")          # an introduction to the package
```

As the first vignette explains, simple feature objects in R are stored in a data frame, with geographic data occupying a special column, usually named 'geom' or 'geometry'. We will use the `world` dataset provided by **spData** (Bivand et al., 2023b), loaded at the beginning of this chapter, to show what `sf` objects are and how they work. `world` is an '`sf` data frame' containing spatial and attribute columns, the names of which are returned by the function `names()` (the last column in this example contains the geographic information).

```
class(world)
#> [1] "sf"         "tbl_df"     "tbl"        "data.frame"
names(world)
#>  [1] "iso_a2"    "name_long" "continent" "region_un" "subregion" "type"
#>  [7] "area_km2"  "pop"       "lifeExp"   "gdpPercap" "geom"
```

The contents of this `geom` column give `sf` objects their spatial powers: `world$geom` is a 'list column' that contains all the coordinates of the country polygons.

`sf` objects can be plotted quickly with the function `plot()`. Although part of R's default installation (base R), `plot()` is a *generic* that is extended by other packages. **sf** contains the non-exported (hidden from users most of the time) `plot.sf()` function which is what is called behind the scenes in the following command, which creates Figure 2.3.

```
plot(world)
```

Note that instead of creating a single map by default for geographic objects, as most GIS programs do, `plot()`ing `sf` objects results in a map for each variable in the datasets. This behavior can be useful for exploring the spatial distribution of different variables and is discussed further in Section 2.2.3.

More broadly, treating geographic objects as regular data frames with spatial powers has many advantages, especially if you are already used to working with data frames. The commonly used `summary()` function, for example, provides a useful overview of the variables within the `world` object.

```
summary(world["lifeExp"])
#>     lifeExp                   geom
#>  Min.   :50.6   MULTIPOLYGON :177
#>  1st Qu.:65.0   epsg:4326    :  0
#>  Median :72.9   +proj=long...:  0
#>  Mean   :70.9
```

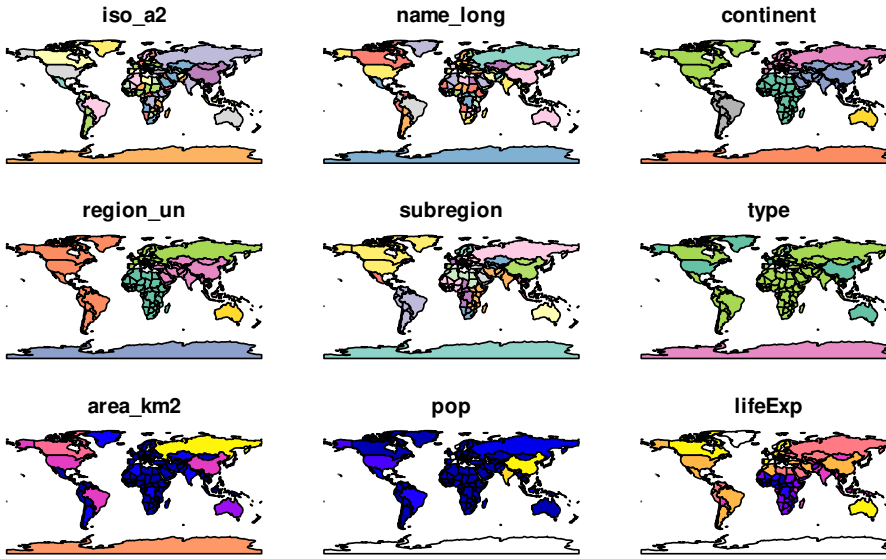

FIGURE 2.3 Map of the world using the sf package, with a facet for each attribute.

```
#>  3rd Qu.:76.8
#>  Max.   :83.6
#>  NA's   :10
```

Although we have only selected one variable for the `summary()` command, it also outputs a report on the geometry. This demonstrates the 'sticky' behavior of the geometry columns of **sf** objects: they are kept unless the user deliberately removes them, as we'll see in Section 3.2. The result provides a quick summary of both the non-spatial and spatial data contained in `world`: the mean average life expectancy is 71 years (ranging from less than 51 to more than 83 years with a median of 73 years) across all countries.

The word `MULTIPOLYGON` in the summary output above refers to the geometry type of features (countries) in the `world` object. This representation is necessary for countries with islands such as Indonesia and Greece. Other geometry types are described in Section 2.2.4.

It is also worth taking a deeper look at the basic behavior and contents of this simple feature object, which can usefully be thought of as a '**s**patial data **f**rame'.

`sf` objects are easy to subset: the code below shows how to return an object containing only the first two rows and the first three columns of the

`world` object. The output shows two major differences compared with a regular `data.frame`: the inclusion of additional geographic metadata (`Geometry type`, `Dimension`, `Bounding box` and coordinate reference system information), and the presence of a 'geometry column', here named `geom`:

```
world_mini = world[1:2, 1:3]
world_mini
#> Simple feature collection with 2 features and 3 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -180 ymin: -18.3 xmax: 180 ymax: -0.95
#> Geodetic CRS:  WGS 84
#> # A tibble: 2 x 4
#>   iso_a2 name_long continent                                        geom
#>   <chr>  <chr>     <chr>                                  <MULTIPOLYGON [°]>
#> 1 FJ     Fiji      Oceania   (((-180 -16.6, -180 -16.5, -180 -16, -180 -16.1, -~
#> 2 TZ     Tanzania  Africa    (((33.9 -0.95, 31.9 -1.03, 30.8 -1.01, 30.4 -1.13,~
```

All this may seem rather complex, especially for a class system that is supposed to be 'simple'! However, there are good reasons for organizing things this way and using **sf** to work with vector geographic datasets.

Before describing each geometry type that the **sf** package supports, it is worth taking a step back to understand the building blocks of `sf` objects. Section 2.2.5 shows how simple features objects are data frames, with special geometry columns. These spatial columns are often called `geom` or `geometry`: `world$geom` refers to the spatial element of the `world` object described above. These geometry columns are 'list columns' of class `sfc` (see Section 2.2.7). In turn, `sfc` objects are composed of one or more objects of class `sfg`: simple feature geometries that we describe in Section 2.2.6.

To understand how the spatial components of simple features work, it is vital to understand simple feature geometries. For this reason, we cover each currently supported simple features geometry type in Section 2.2.4 before moving on to describe how these can be represented in R using `sf` objects, which are based on `sfg` and `sfc` objects.

The preceding code chunk uses `=` to create a new object called `world_mini` in the command `world_mini = world[1:2, 1:3]`. This is called assignment. An equivalent command to achieve the same result is `world_mini <- world[1:2, 1:3]`. Although 'arrow assignment' is more commonly used, we use 'equals assignment' because it's slightly faster to type and easier to teach due to compatibility with commonly used languages such as Python and JavaScript. Which to use is largely a matter of preference as long as you're consistent (packages such as **styler** can be used to change style).

2.2.2 Why simple features?

Simple features is a widely supported data model that underlies data structures in many GIS applications including QGIS and PostGIS. A major advantage of this is that using the data model ensures your work is cross-transferable to other setups, for example importing from and exporting to spatial databases.

A more specific question from an R perspective is "why use the **sf** package?" There are many reasons (linked to the advantages of the simple features model):

- Fast reading and writing of data
- Enhanced plotting performance
- **sf** objects can be treated as data frames in most operations
- **sf** function names are relatively consistent and intuitive (all begin with `st_`)
- **sf** functions can be combined with the `|>` operator and works well with the tidyverse collection of R packages.

sf's support for **tidyverse** packages is exemplified by `read_sf()`, a function for importing geographic vector data covered in detail in Section 8.3.1. Unlike the function `st_read()`, which returns attributes stored in a base R `data.frame` (and which emits verbose messages, not shown in the code chunk below), `read_sf()` silently returns data as a **tidyverse** `tibble`. This is demonstrated below:

```
world_dfr = st_read(system.file("shapes/world.gpkg", package = "spData"))
#> Reading layer `world' from data source
#>   `/home/jn/R/x86_64-redhat-linux-gnu-library/4.4/spData/shapes/world.gpkg'
#>   using driver `GPKG'
#> Simple feature collection with 177 features and 10 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -180 ymin: -89.9 xmax: 180 ymax: 83.6
#> Geodetic CRS:  WGS 84
world_tbl = read_sf(system.file("shapes/world.gpkg", package = "spData"))
class(world_dfr)
#> [1] "sf"         "data.frame"
class(world_tbl)
#> [1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

As described in Chapter 3, which shows how to manipulate `sf` objects with **tidyverse** functions, **sf** is now the go-to package for analysis of spatial vector data in R. **spatstat**, a package ecosystem which provides numerous functions for spatial statistics, and **terra** both have vector geographic data classes, but neither has the same level of uptake as **sf** does for working with vector data. Many popular packages build on **sf**, as shown by the rise in its popularity in

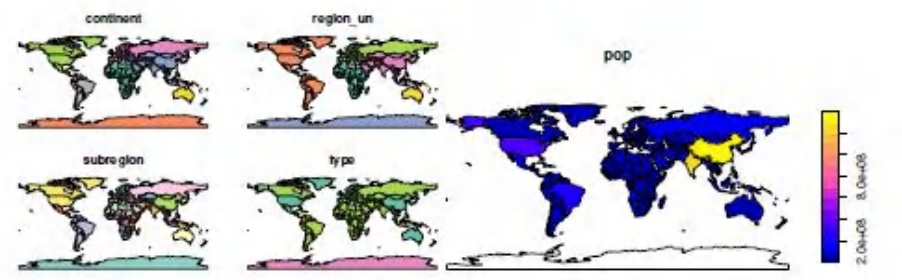

FIGURE 2.4 Plotting with sf, with multiple variables (left) and a single variable (right).

terms of number of downloads per day, as shown in Section 1.5 in the previous chapter.

2.2.3 Basic maps

Basic geographic visualizations (maps) are created in **sf** with base R's `plot()` function. By default, this creates a multi-panel plot, one sub-plot for each variable of the object, as illustrated in the left-hand panel in Figure 2.4. A legend or 'key' with a continuous color is produced if the object to be plotted has a single variable (see the right-hand panel). You can also set fixed colors in `plot()` commands with `col` and `border` arguments.

```
plot(world[3:6])
plot(world["pop"])
```

Plots are added as layers to existing images by setting `add = TRUE`.[4] To demonstrate this, and to provide an insight into the contents of Chapters 3 and 4 on attribute and spatial data operations, the subsequent code chunk filters countries in Asia and combines them into a single feature:

```
world_asia = world[world$continent == "Asia", ]
asia = st_union(world_asia)
```

We can now plot the Asian continent over a map of the world. Note that the first plot must only have one facet for `add = TRUE` to work. If the first plot has a key, `reset = FALSE` must be used:

[4] `plot()`ing of **sf** objects uses `sf:::plot.sf()` behind the scenes. `plot()` is a generic method that behaves differently depending on the class of object being plotted.

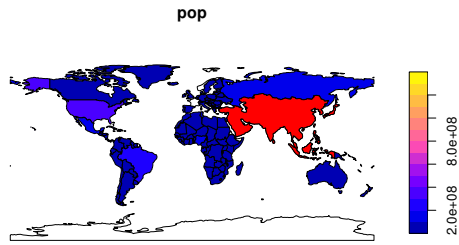

FIGURE 2.5 Plot of Asia added as a layer on top of countries worldwide.

```
plot(world["pop"], reset = FALSE)
plot(asia, add = TRUE, col = "red")
```

Adding layers in this way can be used to verify the geographic correspondence between layers: the `plot()` function is fast and requires few lines of code, but its functionality is limited. For more advanced map-making we recommend using dedicated visualization packages such as **tmap** (Tennekes, 2018) (see Chapter 9).

There are various ways to modify maps with **sf**'s `plot()` method. Because **sf** extends base R plotting methods, `plot()`'s arguments work with `sf` objects (see `?graphics::plot` and `?par` for information on arguments such as `main =`).[5] Figure 2.6 illustrates this flexibility by overlaying circles, whose diameters (set with `cex =`) represent country populations, on a map of the world. An unprojected version of this figure can be created with the following commands (see exercises at the end of this chapter and the script `02-contplot.R` to reproduce Figure 2.6):

```
plot(world["continent"], reset = FALSE)
cex = sqrt(world$pop) / 10000
world_cents = st_centroid(world, of_largest = TRUE)
plot(st_geometry(world_cents), add = TRUE, cex = cex)
```

The code above uses the function `st_centroid()` to convert one geometry type (polygons) to another (points) (see Chapter 5), the aesthetics of which are varied with the `cex` argument.

sf's plot method also has arguments specific to geographic data. `expandBB`, for example, can be used to plot an `sf` object in context: it takes a numeric vector of length four that expands the bounding box of the plot relative to zero in the following order: bottom, left, top, right. This is used to plot India in the

[5] Note: many plot arguments are ignored in facet maps, when more than one `sf` column is plotted.

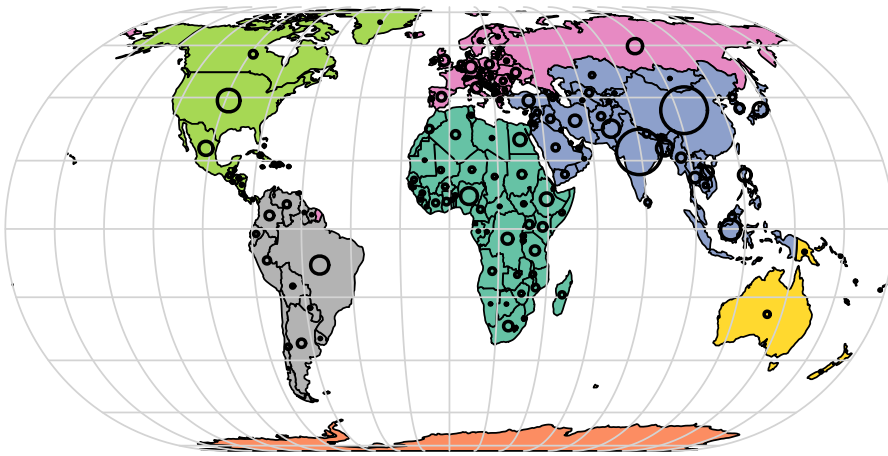

FIGURE 2.6 Country continents (represented by fill color) and 2015 populations (represented by circles, with area proportional to population).

FIGURE 2.7 India in context, demonstrating the expandBB argument.

context of its giant Asian neighbors, with an emphasis on China to the east, in the following code chunk, which generates Figure 2.7 (see exercises below on adding text to plots):[6]

```
india = world[world$name_long == "India", ]
plot(st_geometry(india), expandBB = c(0, 0.2, 0.1, 1), col = "gray", lwd = 3)
plot(st_geometry(world_asia), add = TRUE)
```

Note the use of `lwd` to emphasize India in the plotting code. See Section 9.2 for other visualization techniques for representing a range of geometry types, the subject of the next section.

[6]Note the use of `st_geometry(india)` to return only the geometry associated with the object to prevent attributes being plotted in a simple feature column (`sfc`) object. An alternative is to use `india[0]`, which returns an `sf` object that contains no attribute data..

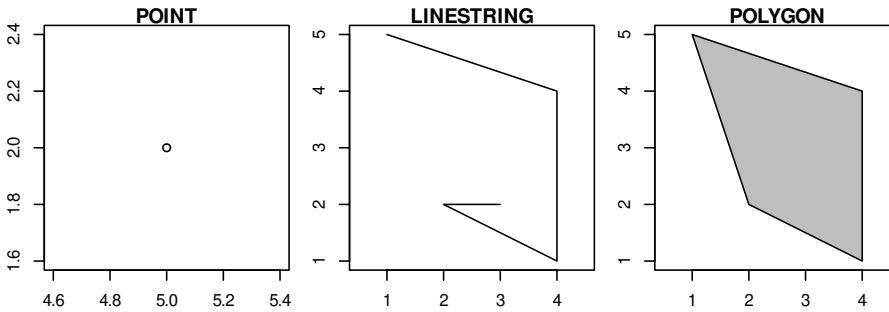

FIGURE 2.8 Point, linestring and polygon geometries.

2.2.4 Geometry types

Geometries are the basic building blocks of simple features. Simple features in R can take on one of the 18 geometry types supported by the **sf** package. In this chapter we will focus on the seven most commonly used types: `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON` and `GEOMETRYCOLLECTION`.

Generally, well-known binary (WKB) or well-known text (WKT) are the standard encoding for simple feature geometries. WKB representations are usually hexadecimal strings easily readable for computers. This is why GIS and spatial databases use WKB to transfer and store geometry objects. WKT, on the other hand, is a human-readable text markup description of simple features. Both formats are exchangeable, and if we present one, we will naturally choose the WKT representation.

The basis for each geometry type is the point. A point is simply a coordinate in two-, three-, or four-dimensional space (see `vignette("sf1")` for more information) such as (Figure 2.8, left panel):

- `POINT (5 2)`

A linestring is a sequence of points with a straight line connecting the points, for example (Figure 2.8, middle panel):

- `LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)`

A polygon is a sequence of points that form a closed, non-intersecting ring. Closed means that the first and the last point of a polygon have the same coordinates (Figure 2.8, right panel).[7]

- Polygon without a hole: `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))`

[7] By definition, a polygon has one exterior boundary (outer ring) and can have zero or more interior boundaries (inner rings), also known as holes. A polygon with a hole would be, for example, `POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5), (2 4, 3 4, 3 3, 2 3, 2 4))`

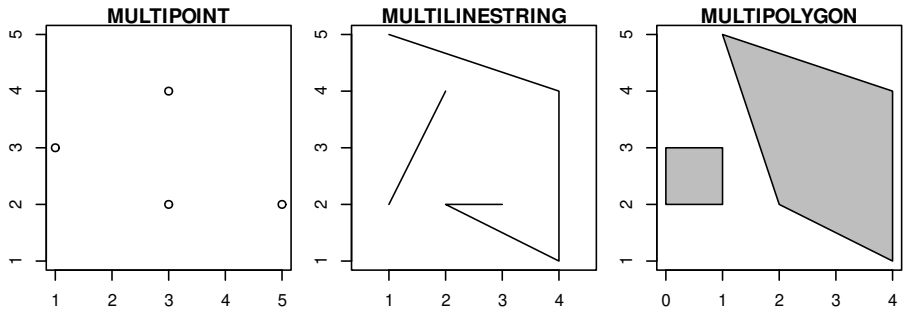

FIGURE 2.9 Illustration of multi* geometries.

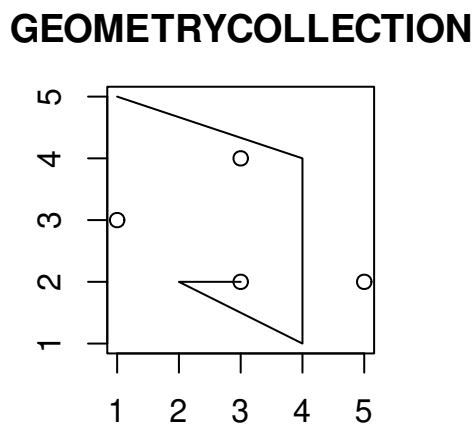

FIGURE 2.10 Illustration of a geometry collection.

So far we have created geometries with only one geometric entity per feature. Simple feature standard also allows multiple geometries of a single type to exist within a single feature within "multi" version of each geometry type (Figure 2.9):

- Multipoint: `MULTIPOINT (5 2, 1 3, 3 4, 3 2)`
- Multilinestring: `MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))`
- Multipolygon: `MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5), (0 2, 1 2, 1 3, 0 3, 0 2)))`

Finally, a geometry collection can contain any combination of geometries including (multi)points and linestrings (see Figure 2.10):

- Geometry collection: `GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2), LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2))`

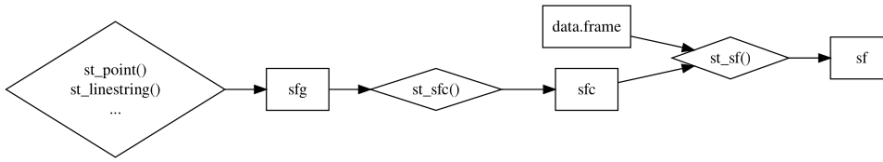

FIGURE 2.11 Building blocks of sf objects.

2.2.5 The sf class

Simple features consist of two main parts: geometries and non-geographic attributes. Figure 2.11 shows how an sf object is created – geometries come from an `sfc` object, while attributes are taken from a `data.frame` or `tibble`.[8]

Non-geographic attributes represent the name of the feature or other attributes such as measured values, groups, and other things. To illustrate attributes, we will represent a temperature of 25°C in London on June 21, 2023. This example contains a geometry (coordinates), and three attributes with three different classes (place name, temperature and date).[9] Objects of class `sf` represent such data by combining the attributes (`data.frame`) with the simple feature geometry column (`sfc`). They are created with `st_sf()` as illustrated below, which creates the London example described above:

```
lnd_point = st_point(c(0.1, 51.5))                 # sfg object
lnd_geom = st_sfc(lnd_point, crs = "EPSG:4326")    # sfc object
lnd_attrib = data.frame(                           # data.frame object
  name = "London",
  temperature = 25,
  date = as.Date("2023-06-21")
)
lnd_sf = st_sf(lnd_attrib, geometry = lnd_geom)    # sf object
```

What just happened? First, the coordinates were used to create the simple feature geometry (`sfg`). Second, the geometry was converted into a simple feature geometry column (`sfc`), with a CRS. Third, attributes were stored in a `data.frame`, which was combined with the `sfc` object with `st_sf()`. This results in an `sf` object, as demonstrated below (some output is omitted):

```
lnd_sf
#> Simple feature collection with 1 features and 3 fields
```

[8] To learn more about building sf geometries from scratch, see the following Sections 2.2.6 and 2.2.7.

[9] Other attributes might include an urbanity category (city or village), or a remark if the measurement was made using an automatic station.

```
#> ...
#>     name temperature       date         geometry
#> 1 London          25 2023-06-21 POINT (0.1 51.5)
```

```
class(lnd_sf)
#> [1] "sf"         "data.frame"
```

The result shows that `sf` objects actually have two classes, `sf` and `data.frame`. Simple features are simply data frames (square tables), but with spatial attributes stored in a list column, usually called `geometry` or `geom`, as described in Section 2.2.1. This duality is central to the concept of simple features: most of the time a `sf` can be treated as and behaves like a `data.frame`. Simple features are, in essence, data frames with a spatial extension.

2.2.6 Simple feature geometries (sfg)

The `sfg` class represents the different simple feature geometry types in R: point, linestring, polygon (and their 'multi' equivalents, such as multipoints) or geometry collection.

Usually you are spared the tedious task of creating geometries on your own since you can simply import an already existing spatial file. However, there are a set of functions to create simple feature geometry objects (`sfg`) from scratch, if needed. The names of these functions are simple and consistent, as they all start with the `st_` prefix and end with the name of the geometry type in lowercase letters:

- A point: `st_point()`
- A linestring: `st_linestring()`
- A polygon: `st_polygon()`
- A multipoint: `st_multipoint()`
- A multilinestring: `st_multilinestring()`
- A multipolygon: `st_multipolygon()`
- A geometry collection: `st_geometrycollection()`

`sfg` objects can be created from three base R data types:

1. A numeric vector: a single point
2. A matrix: a set of points, where each row represents a point, a multipoint or linestring
3. A list: a collection of objects such as matrices, multilinestrings or geometry collections

The function `st_point()` creates single points from numeric vectors:

```
st_point(c(5, 2))                 # XY point
#> POINT (5 2)
st_point(c(5, 2, 3))              # XYZ point
#> POINT Z (5 2 3)
st_point(c(5, 2, 1), dim = "XYM") # XYM point
#> POINT M (5 2 1)
st_point(c(5, 2, 3, 1))           # XYZM point
#> POINT ZM (5 2 3 1)
```

The results show that XY (2D coordinates), XYZ (3D coordinates) and XYZM (3D with an additional variable, typically measurement accuracy) point types are created from vectors of lengths 2, 3, and 4, respectively. The XYM type must be specified using the `dim` argument (which is short for dimension).

By contrast, use matrices in the case of multipoint (`st_multipoint()`) and linestring (`st_linestring()`) objects:

```
# the rbind function simplifies the creation of matrices
## MULTIPOINT
multipoint_matrix = rbind(c(5, 2), c(1, 3), c(3, 4), c(3, 2))
st_multipoint(multipoint_matrix)
#> MULTIPOINT ((5 2), (1 3), (3 4), (3 2))
## LINESTRING
linestring_matrix = rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2))
st_linestring(linestring_matrix)
#> LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)
```

Finally, use lists for the creation of multilinestrings, (multi-)polygons and geometry collections:

```
## POLYGON
polygon_list = list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5)))
st_polygon(polygon_list)
#> POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))


## POLYGON with a hole
polygon_border = rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5))
polygon_hole = rbind(c(2, 4), c(3, 4), c(3, 3), c(2, 3), c(2, 4))
polygon_with_hole_list = list(polygon_border, polygon_hole)
st_polygon(polygon_with_hole_list)
#> POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5), (2 4, 3 4, 3 3, 2 3, 2 4))
```

```
## MULTILINESTRING
multilinestring_list = list(rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2)),
                            rbind(c(1, 2), c(2, 4)))
st_multilinestring(multilinestring_list)
#> MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))


## MULTIPOLYGON
multipolygon_list = list(list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5))),
                         list(rbind(c(0, 2), c(1, 2), c(1, 3), c(0, 3), c(0, 2))))
st_multipolygon(multipolygon_list)
#> MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5)), ((0 2, 1 2, 1 3, 0 3, 0 2)))


## GEOMETRYCOLLECTION
geometrycollection_list = list(st_multipoint(multipoint_matrix),
                               st_linestring(linestring_matrix))
st_geometrycollection(geometrycollection_list)
#> GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2),
#>   LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2))
```

2.2.7 Simple feature columns (sfc)

One `sfg` object contains only a single simple feature geometry. A simple feature geometry column (`sfc`) is a list of `sfg` objects, which is additionally able to contain information about the CRS in use. For instance, to combine two simple features into one object with two features, we can use the `st_sfc()` function. This is important since `sfc` represents the geometry column in **sf** data frames:

```
# sfc POINT
point1 = st_point(c(5, 2))
point2 = st_point(c(1, 3))
points_sfc = st_sfc(point1, point2)
points_sfc
#> Geometry set for 2 features
#> Geometry type: POINT
#> Dimension:     XY
#> Bounding box:  xmin: 1 ymin: 2 xmax: 5 ymax: 3
#> CRS:           NA
#> POINT (5 2)
#> POINT (1 3)
```

In most cases, an `sfc` object contains objects of the same geometry type. Therefore, when we convert `sfg` objects of type polygon into a simple feature geometry column, we would also end up with an `sfc` object of type polygon, which

can be verified with `st_geometry_type()`. Equally, a geometry column of multilinestrings would result in an `sfc` object of type multilinestring:

```
# sfc POLYGON
polygon_list1 = list(rbind(c(1, 5), c(2, 2), c(4, 1), c(4, 4), c(1, 5)))
polygon1 = st_polygon(polygon_list1)
polygon_list2 = list(rbind(c(0, 2), c(1, 2), c(1, 3), c(0, 3), c(0, 2)))
polygon2 = st_polygon(polygon_list2)
polygon_sfc = st_sfc(polygon1, polygon2)
st_geometry_type(polygon_sfc)
#> [1] POLYGON POLYGON
#> 18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE


# sfc MULTILINESTRING
multilinestring_list1 = list(rbind(c(1, 5), c(4, 4), c(4, 1), c(2, 2), c(3, 2)),
                             rbind(c(1, 2), c(2, 4)))
multilinestring1 = st_multilinestring((multilinestring_list1))
multilinestring_list2 = list(rbind(c(2, 9), c(7, 9), c(5, 6), c(4, 7), c(2, 7)),
                             rbind(c(1, 7), c(3, 8)))
multilinestring2 = st_multilinestring((multilinestring_list2))
multilinestring_sfc = st_sfc(multilinestring1, multilinestring2)
st_geometry_type(multilinestring_sfc)
#> [1] MULTILINESTRING MULTILINESTRING
#> 18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE
```

It is also possible to create an `sfc` object from `sfg` objects with different geometry types:

```
# sfc GEOMETRY
point_multilinestring_sfc = st_sfc(point1, multilinestring1)
st_geometry_type(point_multilinestring_sfc)
#> [1] POINT           MULTILINESTRING
#> 18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE
```

As mentioned before, `sfc` objects can additionally store information on the CRS. The default value is `NA` (*Not Available*), as can be verified with `st_crs()`:

```
st_crs(points_sfc)
#> Coordinate Reference System: NA
```

All geometries in `sfc` objects must have the same CRS. A CRS can be specified with the `crs` argument of `st_sfc()` (or `st_sf()`), which takes a **CRS identifier**

provided as a text string, such as `crs = "EPSG:4326"` (see Section 7.2 for other CRS representations and details on what this means).

```
# Set the CRS with an identifier referring to an 'EPSG' CRS code:
points_sfc_wgs = st_sfc(point1, point2, crs = "EPSG:4326")
st_crs(points_sfc_wgs) # print CRS (only first 4 lines of output shown)
#> Coordinate Reference System:
#>   User input: EPSG:4326
#>   wkt:
#> GEOGCRS["WGS 84",
#> ...
```

2.2.8 The sfheaders package

sfheaders is an R package that speeds-up the construction, conversion and manipulation of `sf` objects (Cooley, 2020). It focuses on building `sf` objects from vectors, matrices and data frames, rapidly, and without depending on the **sf** library; and exposing its underlying C++ code through header files (hence the name, **sfheaders**). This approach enables others to extend it using compiled and fast-running code. Every core **sfheaders** function has a corresponding C++ implementation, as described in the `Cpp` vignette. For most people, the R functions will be more than sufficient to benefit from the computational speed of the package. **sfheaders** was developed separately from **sf**, but aims to be fully compatible, creating valid `sf` objects of the type described in preceding sections.

The simplest use case for **sfheaders** is demonstrated in the code chunks below with examples of building `sfg`, `sfc`, and `sf` objects showing:

- A vector converted to `sfg_POINT`
- A matrix converted to `sfg_LINESTRING`
- A data frame converted to `sfg_POLYGON`

We will start by creating the simplest possible `sfg` object, a single coordinate pair, assigned to a vector named `v`:

```
v = c(1, 1)
v_sfg_sfh = sfheaders::sfg_point(obj = v)
v_sfg_sfh # printing without sf loaded
#>      [,1] [,2]
#> [1,]    1    1
#> attr(,"class")
#> [1] "XY"    "POINT" "sfg"
```

The example above shows how the `sfg` object `v_sfg_sfh` is printed when **sf** is not loaded, demonstrating its underlying structure. When **sf** is loaded (as is the case here), the result of the above command is indistinguishable from `sf` objects:

```
v_sfg_sf = st_point(v)
print(v_sfg_sf) == print(v_sfg_sfh)
#> POINT (1 1)
#> POINT (1 1)
#> [1] TRUE
```

The next examples shows how **sfheaders** creates `sfg` objects from matrices and data frames:

```
# matrices
m = matrix(1:8, ncol = 2)
sfheaders::sfg_linestring(obj = m)
#> LINESTRING (1 5, 2 6, 3 7, 4 8)
# data frames
df = data.frame(x = 1:4, y = 4:1)
sfheaders::sfg_polygon(obj = df)
#> POLYGON ((1 4, 2 3, 3 2, 4 1, 1 4))
```

Reusing the objects `v`, `m`, and `df` we can also build simple feature columns (`sfc`) as follows (outputs not shown):

```
sfheaders::sfc_point(obj = v)
sfheaders::sfc_linestring(obj = m)
sfheaders::sfc_polygon(obj = df)
```

Similarly, `sf` objects can be created as follows:

```
sfheaders::sf_point(obj = v)
sfheaders::sf_linestring(obj = m)
sfheaders::sf_polygon(obj = df)
```

In each of these examples, the CRS is not defined. If you plan on doing any calculations or geometric operations using **sf** functions, we encourage you to set the CRS (see Chapter 7 for details):

```
df_sf = sfheaders::sf_polygon(obj = df)
st_crs(df_sf) = "EPSG:4326"
```

sfheaders is also good at 'deconstructing' and 'reconstructing' `sf` objects, meaning converting geometry columns into data frames that contain data on the coordinates of each vertex and geometry feature (and multi-feature) ids. It is fast and reliable at 'casting' geometry columns to different types, a topic covered in Chapter 5. Benchmarks, in the package's documentation and in test code developed for this book, show it is much faster than the `sf` package for such operations.

2.2.9 Spherical geometry operations with S2

Spherical geometry engines are based on the fact that the world is round, while simple mathematical procedures for geocomputation, such as calculating a straight line between two points or the area enclosed by a polygon, assume planar (projected) geometries. Since **sf** version 1.0.0, R supports spherical geometry operations 'out of the box' (and by default), thanks to its interface to Google's S2 spherical geometry engine via the **s2** interface package . S2 is perhaps best known as an example of a Discrete Global Grid System (DGGS). Another example is the H3 global hexagonal hierarchical spatial index (Bondaruk et al., 2020).

Although potentially useful for describing locations anywhere on Earth using character strings, the main benefit of **sf**'s interface to S2 is its provision of drop-in functions for calculations such as distance, buffer, and area calculations, as described in **sf**'s built-in documentation which can be opened with the command `vignette("sf7")`.

sf can run in two modes with respect to S2: on and off. By default the S2 geometry engine is turned on, as can be verified with the following command:

```
sf_use_s2()
#> [1] TRUE
```

An example of the consequences of turning the geometry engine off is shown below, by creating buffers around the `india` object created earlier in the chapter (note the warnings emitted when S2 is turned off) (Figure 2.12):

```
india_buffer_with_s2 = st_buffer(india, 1) # 1 meter
sf_use_s2(FALSE)
#> Spherical geometry (s2) switched off
india_buffer_without_s2 = st_buffer(india, 1) # 1 degree
#> Warning in st_buffer.sfc(st_geometry(x), dist, nQuadSegs, endCapStyle =
#> endCapStyle, : st_buffer does not correctly buffer longitude/latitude data
#> dist is assumed to be in decimal degrees (arc_degrees).
```

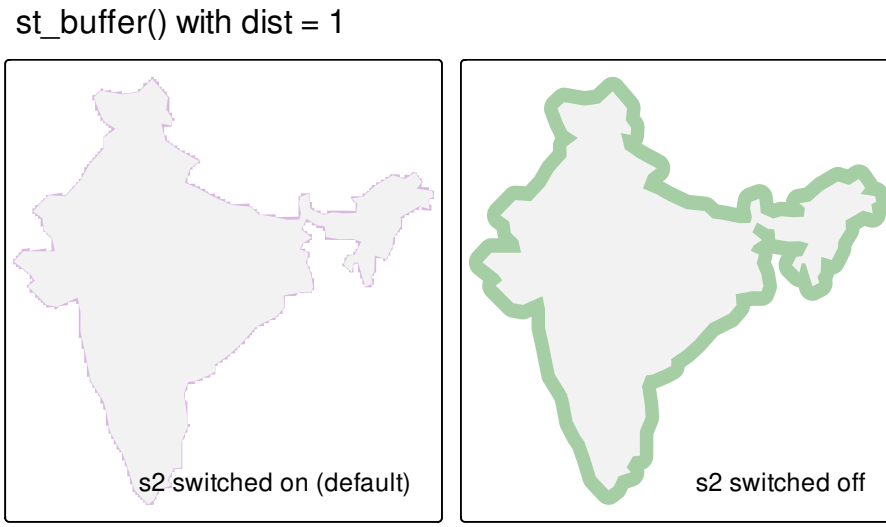

FIGURE 2.12 Example of the consequences of turning off the S2 geometry engine. Both representations of a buffer around India were created with the same command but the purple polygon object was created with S2 switched on, resulting in a buffer of 1 m. The larger light green polygon was created with S2 switched off, resulting in a buffer of 1 degree, which is not accurate.

The right panel of Figure 2.12 is incorrect, as the buffer of 1 degree does not return the equal distance around the `india` polygon (for more explanation of this issue, see Section 7.4).

Throughout this book, we will assume that S2 is turned on, unless explicitly stated. Turn it on again with the following command.

```
sf_use_s2(TRUE)
#> Spherical geometry (s2) switched on
```

Although the **sf**'s use of S2 makes sense in many cases, in some cases there are good reasons for turning S2 off for the duration of an R session or even for an entire project. As documented in issue 1771 in **sf**'s GitHub repo, the default behavior can make code that would work with S2 turned off (and with older versions of **sf**) fail. These edge cases include operations on polygons that are not valid according to S2's stricter definition. If you see error messages such as `#> Error in s2_geography_from_wkb ...` it may be worth trying the command that generated the error message again, after turning off S2. To turn off S2 for the entirety of a project, you can create a file called .Rprofile in the root directory (the main folder) of your project containing the command `sf::sf_use_s2(FALSE)`.

2.3 Raster data

The spatial raster data model represents the world with the continuous grid of cells (often also called pixels; Figure 2.13:A). This data model often refers to so-called regular grids, in which each cell has the same, constant size – and we will focus on the regular grids in this book only. However, several other types of grids exist, including rotated, sheared, rectilinear, and curvilinear grids (see chapter 1 of Pebesma and Bivand (2023b) or chapter 2 of Tennekes and Nowosad (2024)).

The raster data model usually consists of a raster header and a matrix (with rows and columns) representing equally spaced cells (often also called pixels; Figure 2.13:A).[10] The raster header defines the CRS, the extent and the origin. The origin (or starting point) is frequently the coordinate of the lower left corner of the matrix (the **terra** package, however, uses the upper left corner, by default (Figure 2.13:B)). The header defines the extent via the number of columns, the number of rows and the cell size resolution.

The resolution can be calculated as follows:

$$\text{resolution} = \frac{\text{xmax} - \text{xmin}}{\text{ncol}}, \frac{\text{ymax} - \text{ymin}}{\text{nrow}}$$

Starting from the origin, we can easily access and modify each single cell by either using the ID of a cell (Figure 2.13:B) or by explicitly specifying the rows and columns. This matrix representation avoids storing explicitly the coordinates for the four corner points (in fact, it only stores one coordinate, namely the origin) of each cell corner as would be the case for rectangular vector polygons. This and map algebra (Section 4.3.2) make raster processing much more efficient and faster than vector data processing.

In contrast to vector data, the cell of one raster layer can only hold a single value.[11] The value might be continuous or categorical (Figure 2.13C).

Raster maps usually represent continuous phenomena such as elevation, temperature, population density or spectral data. Discrete features such as soil or land-cover classes can also be represented in the raster data model. Both uses of raster datasets are illustrated in Figure 2.14, which shows how the borders of discrete features may become blurred in raster datasets. Depending on the nature of the application, vector representations of discrete features may be more suitable.

[10] Depending on the file format, the header is part of the actual image data file, e.g., GeoTIFF, or stored in an extra header or world file, e.g., ASCII grid formats. There is also the headerless (flat) binary raster format which should facilitate the import into various software programs.

[11] Thus, to store many values for a single location we need to have many raster layers.

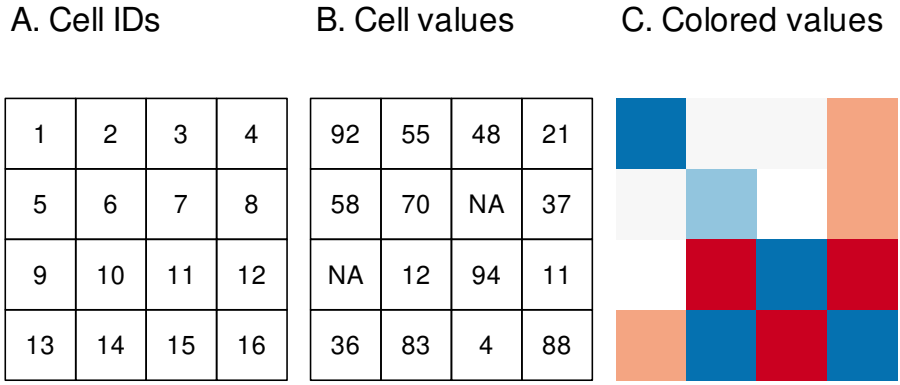

FIGURE 2.13 Raster data types.

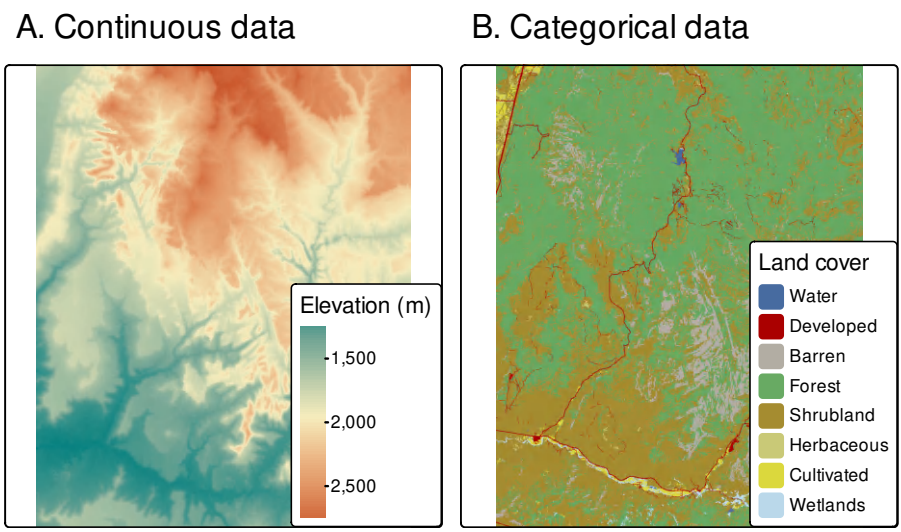

FIGURE 2.14 Examples of (A) continuous and (B) categorical rasters.

2.3.1 R packages for working with raster data

Over the last two decades, several packages for reading and processing raster datasets have been developed. As outlined in Section 1.6, chief among them was **raster**, which led to a step change in R's raster capabilities when it was launched in 2010 and the premier package in the space until the development of **terra** and **stars**. Both more recently developed packages provide powerful and performant functions for working with raster datasets, and there is substantial overlap between their possible use cases. In this book, we focus on **terra**, which replaces the older and (in most cases) slower **raster**. Before learning about how **terra**'s class system works, this section describes similarities and

differences between **terra** and **stars**; this knowledge will help decide which is most appropriate in different situations.

First, **terra** focuses on the most common raster data model (regular grids), while **stars** also allows storing less popular models (including regular, rotated, sheared, rectilinear, and curvilinear grids). While **terra** usually handles one or multi-layered rasters[12], the **stars** package provides ways to store raster data cubes – a raster object with many layers (e.g., bands), for many moments in time (e.g., months), and many attributes (e.g., sensor type A and sensor type B). Importantly, in both packages, all layers or elements of a data cube must have the same spatial dimensions and extent. Second, both packages allow to either read all of the raster data into memory or just to read its metadata – this is usually done automatically based on the input file size. However, they store raster values very differently. **terra** is based on C++ code and mostly uses C++ pointers. **stars** stores values as lists of arrays for smaller rasters or just a file path for larger ones. Third, **stars** functions are closely related to the vector objects and functions in **sf**, while **terra** uses its own class of objects for vector data, namely `SpatVector`, but also accepts `sf` ones.[13] Fourth, both packages have a different approach for how various functions work on their objects. The **terra** package mostly relies on a large number of built-in functions, where each function has a specific purpose (e.g., resampling or cropping). On the other hand, **stars** uses some built-in functions (usually with names starting with `st_`), some existing **dplyr** functions (e.g., `filter()` or `slice()`), and also has its own methods for existing R functions (e.g., `split()` or `aggregate()`).

Importantly, it is straightforward to convert objects from **terra** to **stars** (using `st_as_stars()`) and the other way round (using `rast()`). We also encourage you to read Pebesma and Bivand (2023b) for the most comprehensive introduction to the **stars** package.

2.3.2 Introduction to terra

The **terra** package supports raster objects in R. It provides an extensive set of functions to create, read, export, manipulate and process raster datasets. **terra**'s functionality is largely the same as the more mature **raster** package, but there are some differences: **terra** functions are usually more computationally efficient than **raster** equivalents. On the other hand, the **raster** class system is popular and used by many other packages. You can seamlessly translate between the two types of object to ensure backward compatibility with older scripts and packages, for example, with the functions `raster()`, `stack()`, and `brick()` in the **raster** package (see the previous chapter for more on the evolution of R packages for working with geographic data).

[12] It also has an additional class `SpatRasterDataset` for storing many collections of datasets.

[13] It is also possible to convert between these classes with `vect()` (from `sf` to `SpatVector`) and `st_as_sf()` (from `SpatVector` to `sf`).

In addition to functions for raster data manipulation, **terra** provides many low-level functions that can form a foundation for developing new tools for working with raster datasets. **terra** also lets you work on large raster datasets that are too large to fit into the main memory. In this case, **terra** provides the possibility to divide the raster into smaller chunks, and processes these iteratively instead of loading the whole raster file into RAM.

For the illustration of **terra** concepts, we will use datasets from the **spDataLarge** (Nowosad and Lovelace, 2023). It consists of a few raster objects and one vector object covering an area of Zion National Park (Utah, USA). For example, `srtm.tif` is a digital elevation model of this area (for more details, see its documentation `?srtm`). First, let's create a `SpatRaster` object named `my_rast`:

```
raster_filepath = system.file("raster/srtm.tif", package = "spDataLarge")
my_rast = rast(raster_filepath)
class(my_rast)
#> [1] "SpatRaster"
#> attr(,"package")
#> [1] "terra"
```

Typing the name of the raster into the console, will print out the raster header (dimensions, resolution, extent, CRS) and some additional information (class, data source, summary of the raster values):

```
my_rast
#> class       : SpatRaster
#> dimensions  : 457, 465, 1  (nrow, ncol, nlyr)
#> resolution  : 0.000833, 0.000833  (x, y)
#> extent      : -113, -113, 37.1, 37.5  (xmin, xmax, ymin, ymax)
#> coord. ref. : lon/lat WGS 84 (EPSG:4326)
#> source      : srtm.tif
#> name        : srtm
#> min value   : 1024
#> max value   : 2892
```

Dedicated functions report each component: `dim()` returns the number of rows, columns and layers; `ncell()` the number of cells (pixels); `res()` the spatial resolution; `ext()` its spatial extent; and `crs()` its CRS (raster reprojection is covered in Section 7.8). `inMemory()` reports whether the raster data is stored in memory or on disk, and `sources` specifies the file location.

`help("terra-package")` returns a full list of all available **terra** functions.

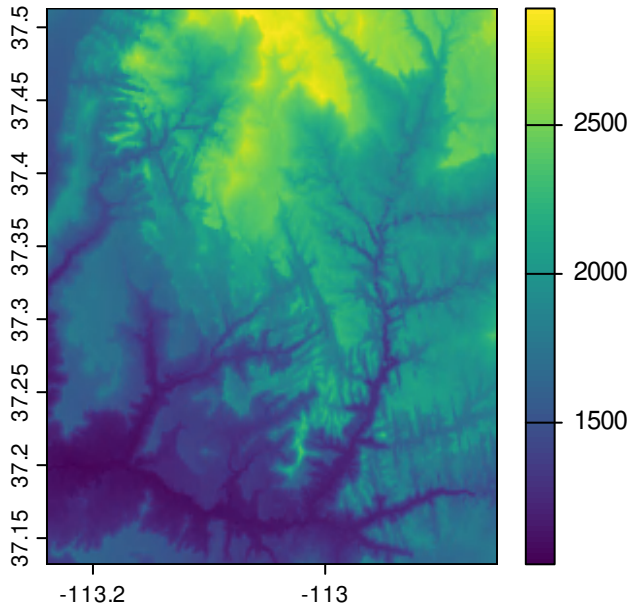

FIGURE 2.15 Basic raster plot.

2.3.3 Basic map-making

Similar to the **sf** package, **terra** also provides `plot()` methods for its own classes. As shown in the following command, the `plot()` function creates a basic raster plot, resulting in Figure 2.15.

```
plot(my_rast)
```

There are several other approaches for plotting raster data in R that are outside the scope of this section, including:

- `plotRGB()` function from the **terra** package to create a plot based on three layers in a `SpatRaster` object
- Packages such as **tmap** to create static and interactive maps of raster and vector objects (see Chapter 9)
- Functions, for example `levelplot()` from the **rasterVis** package, to create facets, a common technique for visualizing change over time

2.3.4 Raster classes

The `SpatRaster` class represents rasters object of **terra**. The easiest way to create a raster object in R is to read-in a raster file from disk or from a server (Section 8.3.2).

```
single_raster_file = system.file("raster/srtm.tif", package = "spDataLarge")
single_rast = rast(raster_filepath)
```

The **terra** package supports numerous drivers with the help of the GDAL library. Rasters from files are usually not read entirely into RAM, with an exception of their header and a pointer to the file itself.

Rasters can also be created from scratch, using the same `rast()` function. This is illustrated in the subsequent code chunk, which results in a new `SpatRaster` object. The resulting raster consists of 36 cells (6 columns and 6 rows specified by `nrows` and `ncols`) centered around the Prime Meridian and the Equator (see `xmin`, `xmax`, `ymin` and `ymax` parameters). Values (`vals`) are assigned to each cell: 1 to cell 1, 2 to cell 2, and so on. Remember: `rast()` fills cells row-wise (unlike `matrix()`) starting at the upper left corner, meaning the top row contains the values 1 to 6, the second 7 to 12, etc. For other ways of creating raster objects, see `?rast`.

```
new_raster = rast(nrows = 6, ncols = 6,
                  xmin = -1.5, xmax = 1.5, ymin = -1.5, ymax = 1.5,
                  vals = 1:36)
```

Given the number of rows and columns as well as the extent (`xmin`, `xmax`, `ymin`, `ymax`), the resolution has to be 0.5. The unit of the resolution is that of the underlying CRS. Here, it is degrees, because the default CRS of raster objects is WGS84. However, one can specify any other CRS with the `crs` argument.

The `SpatRaster` class also handles multiple layers, which typically correspond to a single multi-spectral satellite file or a time-series of rasters.

```
multi_raster_file = system.file("raster/landsat.tif", package = "spDataLarge")
multi_rast = rast(multi_raster_file)
multi_rast
#> class       : SpatRaster
#> dimensions  : 1428, 1128, 4  (nrow, ncol, nlyr)
#> resolution  : 30, 30  (x, y)
#> extent      : 301905, 335745, 4111245, 4154085  (xmin, xmax, ymin, ymax)
#> coord. ref. : WGS 84 / UTM zone 12N (EPSG:32612)
#> source      : landsat.tif
#> names       : landsat_1, landsat_2, landsat_3, landsat_4
#> min values  :      7550,      6404,      5678,      5252
#> max values  :     19071,     22051,     25780,     31961
```

`nlyr()` retrieves the number of layers stored in a `SpatRaster` object:

```
nlyr(multi_rast)
#> [1] 4
```

For multi-layer raster objects, layers can be selected with the `[[` and `$` operators, for example with commands `multi_rast[["landsat_1"]]` and `multi_rast$landsat_1`. The `terra::subset()` can also be used to select layers. It accepts a layer number or its name as the second argument:

```
multi_rast3 = subset(multi_rast, 3)
multi_rast4 = subset(multi_rast, "landsat_4")
```

The opposite operation, combining several `SpatRaster` objects into one, can be done using the `c` function:

```
multi_rast34 = c(multi_rast3, multi_rast4)
```

Most `SpatRaster` objects do not store raster values, but rather a pointer to the file itself. This has a significant side-effect – they cannot be directly saved to `".rds"` or `".rda"` files or used in cluster computing. In these cases, there are two main possible solutions: (1) use of the `wrap()` function that creates a special kind of temporary object that can be saved as an R object or used in cluster computing, or (2) save the object as a regular raster with `writeRaster()`.

2.4 Coordinate Reference Systems

Vector and raster spatial data types share concepts intrinsic to spatial data. Perhaps the most fundamental of these is the coordinate reference systems (CRSs), which defines how the spatial elements of the data relate to the surface of the Earth (or other bodies). CRSs are either geographic or projected, as introduced at the beginning of this chapter (see Figure 2.1). This section explains each type, laying the foundations for Chapter 7, which provides a deep dive into setting, transforming and querying CRSs.

2.4.1 Geographic coordinate reference systems

Geographic CRSs identify any location on the Earth's surface using two values — longitude and latitude (Figure 2.17, left panel). *Longitude* is location in the East-West direction in angular distance from the Prime Meridian plane.

Latitude is angular distance North or South of the equatorial plane. Distances in geographic CRSs are therefore not measured in meters. This has important consequences, as demonstrated in Section 7.

The surface of the Earth in geographic CRSs is represented by a spherical or ellipsoidal surface. Spherical models assume that the Earth is a perfect sphere of a given radius – they have the advantage of simplicity but, at the same time, they are inaccurate as the Earth is not exactly a sphere. Ellipsoidal models are slightly more accurate, and are defined by two parameters: the equatorial radius and the polar radius. These are suitable because the Earth is compressed: the equatorial radius is around 11.5 km longer than the polar radius (Maling, 1992).[14]

Ellipsoids are part of a wider component of CRSs: the *datum*. This contains information on what ellipsoid to use and the precise relationship between the coordinates and location on the Earth's surface. There are two types of datum — geocentric (such as `WGS84`) and local (such as `NAD83`). You can see examples of these two types of datums in Figure 2.16. Black lines represent a *geocentric datum*, whose center is located in the Earth's center of gravity and is not optimized for a specific location. In a *local datum*, shown as a purple dashed line, the ellipsoidal surface is shifted to align with the surface at a particular location. These allow local variations in Earth's surface, for example due to large mountain ranges, to be accounted for in a local CRS. This can be seen in Figure 2.16, where the local datum is fitted to the area of Philippines, but is misaligned with most of the rest of the planet's surface. Both datums in Figure 2.16 are put on top of a geoid — a model of global mean sea level.[15]

2.4.2 Projected coordinate reference systems

All projected CRSs are based on a geographic CRS, described in the previous section, and rely on map projections to convert the three-dimensional surface of the Earth into Easting and Northing (x and y) values in a projected CRS. Projected CRSs are based on Cartesian coordinates on an implicitly flat surface (Figure 2.17, right panel). They have an origin, x and y axes, and a linear unit of measurement such as meters.

This transition cannot be done without adding some deformations. Therefore, some properties of the Earth's surface are distorted in this process, such as area, direction, distance, and shape. A projected coordinate reference system can preserve only one or two of those properties. Projections are often named

[14] The degree of compression is often referred to as *flattening*, defined in terms of the equatorial radius (a) and polar radius (b) as follows: $f = (a - b)/a$. The terms *ellipticity* and *compression* can also be used. Because f is a rather small value, digital ellipsoid models use the 'inverse flattening' ($rf = 1/f$) to define the Earth's compression. Values of a and rf in various ellipsoidal models can be seen by executing `sf_proj_info(type = "ellps")`.

[15] Note that the geoid in Figure 2.16 is exaggerated by a factor of 10,000 to highlight the irregular shape of the planet.

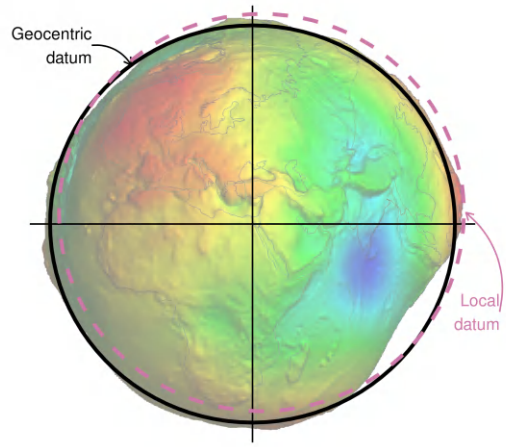

FIGURE 2.16 Geocentric and local geodetic datums shown on top of a geoid (in false color and the vertical exaggeration by 10,000 scale factor). Image of the geoid is adapted from the work of Ince et al. (2019).

based on a property they preserve: equal-area preserves area, azimuthal preserve direction, equidistant preserve distance, and conformal preserve local shape.

There are three main groups of projection types: conic, cylindrical, and planar (azimuthal). In a conic projection, the Earth's surface is projected onto a cone along a single line of tangency or two lines of tangency. Distortions are minimized along the tangency lines and rise with the distance from those lines in this projection. Therefore, it is the best suited for maps of mid-latitude areas. A cylindrical projection maps the surface onto a cylinder. This projection could also be created by touching the Earth's surface along a single line of tangency or two lines of tangency. Cylindrical projections are used most often when mapping the entire world. A planar projection projects data onto a flat surface touching the globe at a point or along a line of tangency. It is typically used in mapping polar regions. `sf_proj_info(type = "proj")` gives a list of the available projections supported by the PROJ library.

A quick summary of different projections, their types, properties, and suitability can be found at www.geo-projections.com. We will expand on CRSs and explain how to project from one CRS to another in Chapter 7. For now, it is sufficient to know:

- That coordinate systems are a key component of geographic objects
- Which CRS your data is in, and whether it is in geographic (lon/lat) or projected (typically meters), is important and has consequences for how R handles spatial and geometry operations

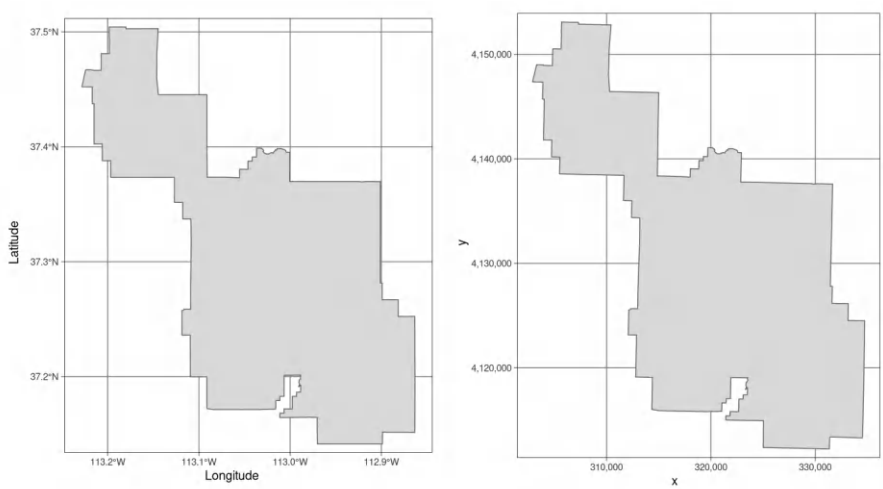

FIGURE 2.17 Examples of geographic (WGS 84; left) and projected (NAD83 / UTM zone 12N; right) coordinate systems for a vector data type.

- That CRSs of `sf` objects can be queried with the function `st_crs()` and CRSs of `terra` objects can be queried with the function `crs()`

2.5 Units

An important feature of CRSs is that they contain information about spatial units. Clearly, it is vital to know whether a house's measurements are in feet or meters, and the same applies to maps. It is good cartographic practice to add a *scale bar* or some other distance indicator onto maps to demonstrate the relationship between distances on the page or screen and distances on the ground. Likewise, it is important to formally specify the units in which the geometry data or cells are measured to provide context, and to ensure that subsequent calculations are done in context.

A novel feature of geometry data in `sf` objects is that they have *native support* for units. This means that distance, area and other geometric calculations in **sf** return values that come with a `units` attribute, defined by the **units** package (Pebesma et al., 2016). This is advantageous, preventing confusion caused by different units (most CRSs use meters, some use feet) and providing information on dimensionality. This is demonstrated in the code chunk below, which calculates the area of Luxembourg:

```
luxembourg = world[world$name_long == "Luxembourg", ]
```

```
st_area(luxembourg) # requires the s2 package in recent versions of sf
#> 2.41e+09 [m^2]
```

The output is in units of square meters (m^2), showing that the result represents two-dimensional space. This information, stored as an attribute (which interested readers can discover with `attributes(st_area(luxembourg))`), can feed into subsequent calculations that use units, such as population density (which is measured in people per unit area, typically per km^2). Reporting units prevents confusion. To take the Luxembourg example, if the units remained unspecified, one could incorrectly assume that the units were in hectares. To translate the huge number into a more digestible size, it is tempting to divide the results by a million (the number of square meters in a square kilometer):

```
st_area(luxembourg) / 1000000
#> 2409 [m^2]
```

However, the result is incorrectly given again as square meters. The solution is to set the correct units with the **units** package:

```
units::set_units(st_area(luxembourg), km^2)
#> 2409 [km^2]
```

Units are of equal importance in the case of raster data. However, so far **sf** is the only spatial package that supports units, meaning that people working on raster data should approach changes in the units of analysis (for example, converting pixel widths from imperial to decimal units) with care. The `my_rast` object (see above) uses a WGS84 projection with decimal degrees as units. Consequently, its resolution is also given in decimal degrees, but you have to know it, since the `res()` function simply returns a numeric vector.

```
res(my_rast)
#> [1] 0.000833 0.000833
```

If we used the Universal Transverse Mercator (UTM) projection, the units would change.

```
repr = project(my_rast, "EPSG:26912")
res(repr)
#> [1] 83.5 83.5
```

Again, the `res()` command gives back a numeric vector without any unit, forcing us to know that the unit of the UTM projection is meters.

2.6 Exercises

E1. Use `summary()` on the geometry column of the `world` data object that is included in the **spData** package. What does the output tell us about:

- Its geometry type?
- The number of countries?
- Its coordinate reference system (CRS)?

E2. Run the code that 'generated' the map of the world in Section 2.2.3 (Basic map-making). Find two similarities and two differences between the image on your computer and that in the book.

- What does the `cex` argument do (see `?plot`)?
- Why was `cex` set to the `sqrt(world$pop) / 10000`?
- Bonus: experiment with different ways to visualize the global population.

E3. Use `plot()` to create maps of Nigeria in context (see Section 2.2.3).

- Adjust the `lwd`, `col` and `expandBB` arguments of `plot()`.
- Challenge: read the documentation of `text()` and annotate the map.

E4. Create an empty `SpatRaster` object called `my_raster` with 10 columns and 10 rows. Assign random values between 0 and 10 to the new raster and plot it.

E5. Read-in the `raster/nlcd.tif` file from the **spDataLarge** package. What kind of information can you get about the properties of this file?

E6. Check the CRS of the `raster/nlcd.tif` file from the **spDataLarge** package. What kind of information you can learn from it?

3

Attribute data operations

Prerequisites

- This chapter requires the following packages to be installed and attached:

```
library(sf)      # vector data package introduced in Chapter 2
library(terra)   # raster data package introduced in Chapter 2
library(dplyr)   # tidyverse package for data frame manipulation
```

- It relies on **spData**, which loads datasets used in the code examples of this chapter:

```
library(spData)  # spatial data package introduced in Chapter 2
```

- Also ensure you have installed the **tidyr** package, or the **tidyverse** of which it is a part, if you want to run data 'tidying' operations in Section 3.2.5.

3.1 Introduction

Attribute data is non-spatial information associated with geographic (geometry) data. A bus stop provides a simple example: its position would typically be represented by latitude and longitude coordinates (geometry data), in addition to its name. The Elephant & Castle / New Kent Road stop in London, for example has coordinates of −0.098 degrees longitude and 51.495 degrees latitude which can be represented as `POINT (-0.098 51.495)` in the `sfc` representation described in Chapter 2. Attributes, such as *name*, of the POINT feature (to use simple features terminology) are the topic of this chapter.

Another example is the elevation value (attribute) for a specific grid cell in raster data. Unlike the vector data model, the raster data model stores the

coordinate of the grid cell indirectly, meaning the distinction between attribute and spatial information is less clear. To illustrate the point, think of a pixel in row 3 and column 4 of a raster matrix. Its spatial location is defined by its index in the matrix: move from the origin four cells in the x direction (typically east and right on maps) and three cells in the y direction (typically south and down). The raster's *resolution* defines the distance for each x- and y-step which is specified in a *header*. The header is a vital component of raster datasets which specifies how pixels relate to spatial coordinates (see also Chapter 4).

This chapter teaches how to manipulate geographic objects based on attributes such as the names of bus stops in a vector dataset and elevations of pixels in a raster dataset. For vector data, this means techniques such as subsetting and aggregation (see Sections 3.2.1 to 3.2.3). Sections 3.2.4 and 3.2.5 demonstrate how to join data onto simple feature objects using a shared ID and how to create new variables, respectively. Each of these operations has a spatial equivalent: the `[` operator in base R, for example, works equally for subsetting objects based on their attribute and spatial objects; you can also join attributes in two geographic datasets using spatial joins. This is good news: skills developed in this chapter are cross-transferable.

After a deep dive into various types of *vector* attribute operations in the next section, *raster* attribute data operations are covered. Creation of raster layers containing continuous and categorical attributes and extraction of cell values from one or more layer (raster subsetting) (Section 3.3.1) are demonstrated. Section 3.3.2 provides an overview of 'global' raster operations which can be used to summarize entire raster datasets. Chapter 4 extends the methods presented here to the spatial world.

3.2 Vector attribute manipulation

Geographic vector datasets are well supported in R thanks to the `sf` class, which extends base R's `data.frame`. Like data frames, `sf` objects have one column per attribute variable (such as 'name') and one row per observation or *feature* (e.g., per bus station). `sf` objects differ from basic data frames because they have a `geometry` column of class `sfc` which can contain a range of geographic entities (single and 'multi' point, line, and polygon features) per row. This was described in Chapter 2, which demonstrated how *generic methods* such as `plot()` and `summary()` work with `sf` objects. **sf** also provides generics that allow `sf` objects to behave like regular data frames, as shown by printing the class's methods:

```
methods(class = "sf") # methods for sf objects, first 12 shown
```

```
#> [1] [             [[<-          $<-           aggregate
#> [5] as.data.frame cbind         coerce        filter
#> [9] identify      initialize    merge         plot
```

Many of these (`aggregate()`, `cbind()`, `merge()`, `rbind()` and `[`) are for manipulating data frames. `rbind()`, for example, binds rows of data frames together, one 'on top' of the other. `$<-` creates new columns. A key feature of `sf` objects is that they store spatial and non-spatial data in the same way, as columns in a `data.frame`.

The geometry column of `sf` objects is typically called `geometry` or `geom`, but any name can be used. The following command, for example, creates a geometry column named g:

```
st_sf(data.frame(n = world$name_long), g = world$geom)
```

This enables geometries imported from spatial databases to have a variety of names such as `wkb_geometry` and `the_geom`.

`sf` objects can also extend the tidyverse classes for data frames, `tbl_df` and `tbl`. Thus **sf** enables the full power of R's data analysis capabilities to be unleashed on geographic data, whether you use base R or tidyverse functions for data analysis. **sf** objects can also be used with the high-performance data processing package **data.table** although, as documented in the issue `Rdatatable/data.table#2273`, is not fully compatible with `sf` objects. Before using these capabilities, it is worth recapping how to discover the basic properties of vector data objects. Let's start by using base R functions to learn about the `world` dataset from the **spData** package:

```
class(world) # it's an sf object and a (tidy) data frame
#> [1] "sf"         "tbl_df"     "tbl"        "data.frame"
dim(world)   # it is a two-dimensional object, with 177 rows and 11 columns
#> [1] 177  11
```

`world` contains ten non-geographic columns (and one geometry list column) with almost 200 rows representing the world's countries. The function `st_drop_geometry()` keeps only the attributes data of an `sf` object, in other words removing its geometry:

```
world_df = st_drop_geometry(world)
class(world_df)
#> [1] "tbl_df"     "tbl"        "data.frame"
ncol(world_df)
#> [1] 10
```

Dropping the geometry column before working with attribute data can be useful; data manipulation processes can run faster when they work only on the attribute data and geometry columns are not always needed. For most cases, however, it makes sense to keep the geometry column, explaining why the column is 'sticky' (it remains after most attribute operations unless specifically dropped). Non-spatial data operations on `sf` objects only change an object's geometry when appropriate (e.g., by dissolving borders between adjacent polygons following aggregation). Becoming skilled at geographic attribute data manipulation means becoming skilled at manipulating data frames.

For many applications, the tidyverse package **dplyr** (Wickham et al., 2023) offers an effective approach for working with data frames. Tidyverse compatibility is an advantage of **sf** over its predecessor **sp**, but there are some pitfalls to avoid (see the supplementary `tidyverse-pitfalls` vignette at geocompx.org for details).

3.2.1 Vector attribute subsetting

Base R subsetting methods include the operator `[` and the function `subset()`. The key **dplyr** subsetting functions are `filter()` and `slice()` for subsetting rows, and `select()` for subsetting columns. Both approaches preserve the spatial components of attribute data in `sf` objects, while using the operator `$` or the **dplyr** function `pull()` to return a single attribute column as a vector will lose the geometry data, as we will see. This section focuses on subsetting `sf` data frames; for further details on subsetting vectors and non-geographic data frames we recommend reading Section 2.7 of *An Introduction to R* (R Core Team, 2021) and chapter 4 of *Advanced R Programming* (Wickham, 2019), respectively.

The `[` operator can subset both rows and columns. Indices placed inside square brackets placed directly after a data frame object name specify the elements to keep. The command `object[i, j]` means 'return the rows represented by `i` and the columns represented by `j`', where `i` and `j` typically contain integers or `TRUE`s and `FALSE`s (indices can also be character strings, indicating row or column names). `object[5, 1:3]`, for example, means 'return data containing the 5th row and columns 1 to 3: the result should be a data frame with only 1 row and 3 columns, and a fourth geometry column if it's an `sf` object. Leaving `i` or `j` empty returns all rows or columns, so `world[1:5, ]` returns the first five rows and all 11 columns. The examples below demonstrate subsetting with base R. Guess the number of rows and columns in the `sf` data frames returned by each command and check the results on your own computer (see the end of the chapter for more exercises):

```
world[1:6, ]    # subset rows by position
world[, 1:3]    # subset columns by position
```

```
world[1:6, 1:3] # subset rows and columns by position
world[, c("name_long", "pop")] # columns by name
world[, c(T, T, F, F, F, F, F, T, T, F, F)] # by logical indices
world[, 888] # an index representing a non-existent column
```

A demonstration of the utility of using `logical` vectors for subsetting is shown in the code chunk below. This creates a new object, `small_countries`, containing nations whose surface area is smaller than 10,000 km^2.

```
i_small = world$area_km2 < 10000
summary(i_small) # a logical vector
#>    Mode   FALSE    TRUE
#> logical     170       7
small_countries = world[i_small, ]
```

The intermediary `i_small` (short for index representing small countries) is a logical vector that can be used to subset the seven smallest countries in the `world` by surface area. A more concise command, which omits the intermediary object, generates the same result:

```
small_countries = world[world$area_km2 < 10000, ]
```

The base R function `subset()` provides another way to achieve the same result:

```
small_countries = subset(world, area_km2 < 10000)
```

Base R functions are mature, stable and widely used, making them a rock solid choice, especially in contexts where reproducibility and reliability are key. **dplyr** functions enable 'tidy' workflows which some people (the authors of this book included) find intuitive and productive for interactive data analysis, especially when combined with code editors such as RStudio that enable auto-completion of column names. Key functions for subsetting data frames (including `sf` data frames) with **dplyr** functions are demonstrated below.

`select()` selects columns by name or position. For example, you could select only two columns, `name_long` and `pop`, with the following command:

```
world1 = select(world, name_long, pop)
names(world1)
#> [1] "name_long" "pop"       "geom"
```

Note: as with the equivalent command in base R (`world[, c("name_long", "pop")]`), the sticky `geom` column remains. `select()` also allows selecting a range of columns with the help of the `:` operator:

```
# all columns between name_long and pop (inclusive)
world2 = select(world, name_long:pop)
```

You can remove specific columns with the - operator:

```
# all columns except subregion and area_km2 (inclusive)
world3 = select(world, -subregion, -area_km2)
```

Subset and rename columns at the same time with the `new_name = old_name` syntax:

```
world4 = select(world, name_long, population = pop)
```

It is worth noting that the command above is more concise than base R equivalent, which requires two lines of code:

```
world5 = world[, c("name_long", "pop")] # subset columns by name
names(world5)[names(world5) == "pop"] = "population" # rename column manually
```

`select()` also works with 'helper functions' for more advanced subsetting operations, including `contains()`, `starts_with()` and `num_range()` (see the help page with `?select` for details).

Most **dplyr** verbs return a data frame, but you can extract a single column as a vector with `pull()`. You can get the same result in base R with the list subsetting operators `$` and `[[`, the three following commands return the same numeric vector:

```
pull(world, pop)
world$pop
world[["pop"]]
```

`slice()` is the row-equivalent of `select()`. The following code chunk, for example, selects rows 1 to 6:

```
slice(world, 1:6)
```

`filter()` is **dplyr**'s equivalent of base R's `subset()` function. It keeps only rows matching given criteria, e.g., only countries with an area below a certain threshold, or with a high average of life expectancy, as shown in the following examples:

TABLE 3.1 Comparison operators that return Boolean (true/false) values.

Symbol	Name
==	Equal to
!=	Not equal to
>, <	Greater/Less than
>=, <=	Greater/Less than or equal
&, |, !	Logical operators: And, Or, Not

```
world7 = filter(world, area_km2 < 10000)  # countries with a small area
world7 = filter(world, lifeExp > 82)      # with high life expectancy
```

The standard set of comparison operators can be used in the `filter()` function, as illustrated in Table 3.1.

3.2.2 Chaining commands with pipes

Key to workflows using **dplyr** functions is the 'pipe' operator `%>%` (or since R `4.1.0` the native pipe `|>`), which takes its name from the Unix pipe `|` (Grolemund and Wickham, 2016). Pipes enable expressive code: the output of a previous function becomes the first argument of the next function, enabling *chaining*. This is illustrated below, in which only countries from Asia are filtered from the `world` dataset, next the object is subset by columns (`name_long` and `continent`) and the first five rows (result not shown).

```
world7 = world |>
  filter(continent == "Asia") |>
  select(name_long, continent) |>
  slice(1:5)
```

The above chunk shows how the pipe operator allows commands to be written in a clear order: the above run from top to bottom (line-by-line) and left to right. An alternative to piped operations is nested function calls, which are harder to read:

```
world8 = slice(
  select(
    filter(world, continent == "Asia"),
    name_long, continent),
  1:5)
```

Another alternative is to split the operations into multiple self-contained lines, which is recommended when developing new R packages, an approach which has the advantage of saving intermediate results with distinct names which can be later inspected for debugging purposes (an approach which has disadvantages of being verbose and cluttering the global environment when undertaking interactive analysis):

```
world9_filtered = filter(world, continent == "Asia")
world9_selected = select(world9_filtered, continent)
world9 = slice(world9_selected, 1:5)
```

Each approach has advantages and disadvantages, the importance of which depend on your programming style and applications. For interactive data analysis, the focus of this chapter, we find piped operations fast and intuitive, especially when combined with RStudio/VSCode shortcuts for creating pipes and auto-completing variable names.

3.2.3 Vector attribute aggregation

Aggregation involves summarizing data with one or more 'grouping variables', typically from columns in the data frame to be aggregated (geographic aggregation is covered in the next chapter). An example of attribute aggregation is calculating the number of people per continent based on country-level data (one row per country). The `world` dataset contains the necessary ingredients: the columns `pop` and `continent`, the population and the grouping variable, respectively. The aim is to find the `sum()` of country populations for each continent, resulting in a smaller data frame (aggregation is a form of data reduction and can be a useful early step when working with large datasets). This can be done with the base R function `aggregate()` as follows:

```
world_agg1 = aggregate(pop ~ continent, FUN = sum, data = world,
                       na.rm = TRUE)
class(world_agg1)
#> [1] "data.frame"
```

The result is a non-spatial data frame with six rows, one per continent, and two columns reporting the name and population of each continent (see Table 3.2 with results for the top three most populous continents).

`aggregate()` is a generic function which means that it behaves differently depending on its inputs. **sf** provides the method `aggregate.sf()` which is activated automatically when `x` is an `sf` object and a `by` argument is provided:

```
world_agg2 = aggregate(world["pop"], by = list(world$continent), FUN = sum,
                       na.rm = TRUE)
```

TABLE 3.2 The top three most populous continents ordered by number of countries.

Continent	Pop	Area	N	Density
Africa	1154946633	29946198	51	39
Asia	4311408059	31252459	47	138
Europe	669036256	23065219	39	29

```
class(world_agg2)
#> [1] "sf"         "data.frame"
nrow(world_agg2)
#> [1] 8
```

The resulting `world_agg2` object is a spatial object containing eight features representing the continents of the world (and the open ocean).

`group_by() |> summarize()` is the **dplyr** equivalent of `aggregate()`. Grouping variables are defined in the `group_by()` function and to aggregation formula is defined in the `summarize()` function, as shown below:

```
world_agg3 = world |>
  group_by(continent) |>
  summarize(pop = sum(pop, na.rm = TRUE))
```

The approach may seem more complex, but it has benefits: flexibility, readability, and control over the new column names. This flexibility is illustrated in the command below, which calculates not only the population but also the area and number of countries in each continent:

```
world_agg4  = world |>
  group_by(continent) |>
  summarize(Pop = sum(pop, na.rm = TRUE), Area = sum(area_km2), N = n())
```

In the previous code chunk `Pop`, `Area` and `N` are column names in the result, and `sum()` and `n()` were the aggregating functions. These aggregating functions return `sf` objects with rows representing continents and geometries containing the multiple polygons representing each land mass and associated islands (this works thanks to the geometric operation 'union', as explained in Section 5.2.7).

Let's combine what we have learned so far about **dplyr** functions, by chaining multiple commands to summarize attribute data about countries worldwide by continent. The following command calculates population density (with `mutate()`), arranges continents by the number of countries they contain

(with `arrange()`), and keeps only the three most populous continents (with `slice_max()`), the result of which is presented in Table 3.2):

```
world_agg5 = world |>
  st_drop_geometry() |>                     # drop the geometry for speed
  select(pop, continent, area_km2) |> # subset the columns of interest
  group_by(Continent = continent) |>  # group by continent and summarize:
  summarize(Pop = sum(pop, na.rm = TRUE), Area = sum(area_km2), N = n()) |>
  mutate(Density = round(Pop / Area)) |>     # calculate population density
  slice_max(Pop, n = 3) |>                  # keep only the top 3
  arrange(desc(N))                          # arrange in order of n. countries
```

More details are provided in the help pages (which can be accessed via `?summarize` and `vignette(package = "dplyr")` and Chapter 5 of R for Data Science.

3.2.4 Vector attribute joining

Combining data from different sources is a common task in data preparation. Joins do this by combining tables based on a shared 'key' variable. **dplyr** has multiple join functions including `left_join()` and `inner_join()` — see `vignette("two-table")` for a full list. These function names follow conventions used in the database language SQL (Grolemund and Wickham, 2016, Chapter 13); using them to join non-spatial datasets to `sf` objects is the focus of this section. **dplyr** join functions work the same on data frames and `sf` objects, the only important difference being the `geometry` list column. The result of data joins can be either an `sf` or `data.frame` object. The most common type of attribute join on spatial data takes an `sf` object as the first argument and adds columns to it from a `data.frame` specified as the second argument.

To demonstrate joins, we will combine data on coffee production with the `world` dataset. The coffee data is in a data frame called `coffee_data` from the **spData** package (see `?coffee_data` for details). It has three columns: `name_long` names major coffee-producing nations and `coffee_production_2016` and `coffee_production_2017` contain estimated values for coffee production in units of 60-kg bags in each year. A 'left join', which preserves the first dataset, merges `world` with `coffee_data`.

```
world_coffee = left_join(world, coffee_data)
#> Joining with `by = join_by(name_long)`
class(world_coffee)
#> [1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

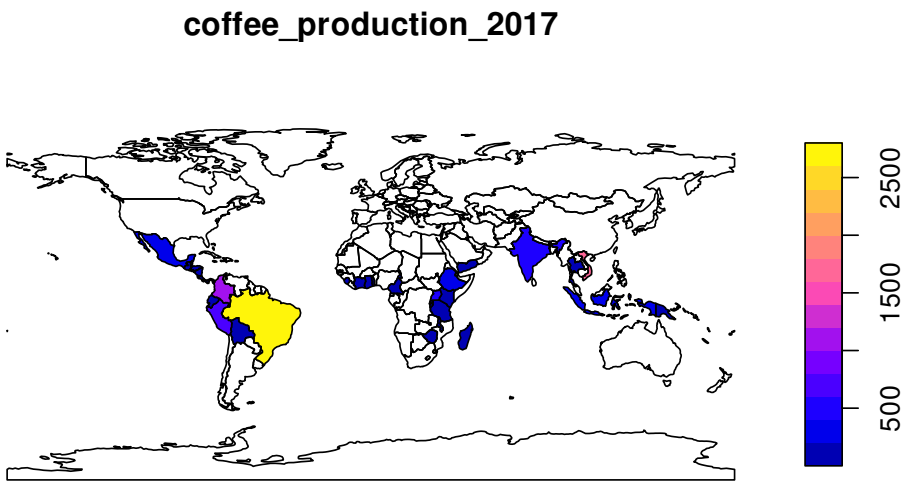

FIGURE 3.1 World coffee production (thousand 60-kg bags) by country, 2017. Source: International Coffee Organization.

Because the input datasets share a 'key variable' (`name_long`) the join worked without using the `by` argument (see `?left_join` for details). The result is an `sf` object identical to the original `world` object but with two new variables (with column indices 11 and 12) on coffee production. This can be plotted as a map, as illustrated in Figure 3.1, generated with the `plot()` function below.

```
names(world_coffee)
#>  [1] "iso_a2"                 "name_long"              "continent"
#>  [4] "region_un"              "subregion"              "type"
#>  [7] "area_km2"               "pop"                    "lifeExp"
#> [10] "gdpPercap"              "geom"                   "coffee_production_2016"
#> [13] "coffee_production_2017"
plot(world_coffee["coffee_production_2017"])
```

For joining to work, a 'key variable' must be supplied in both datasets. By default, **dplyr** uses all variables with matching names. In this case, both `coffee_data` and `world` objects contained a variable called `name_long`, explaining the message `Joining with 'by = join_by(name_long)'`. In the majority of cases where variable names are not the same, you have two options:

1. Rename the key variable in one of the objects so they match.
2. Use the `by` argument to specify the joining variables.

The latter approach is demonstrated below on a renamed version of `coffee_data`.

```
coffee_renamed = rename(coffee_data, nm = name_long)
world_coffee2 = left_join(world, coffee_renamed, by = join_by(name_long == nm))
```

Note that the name in the original object is kept, meaning that `world_coffee` and the new object `world_coffee2` are identical. Another feature of the result is that it has the same number of rows as the original dataset. Although there are only 47 rows of data in `coffee_data`, all 177 country records are kept intact in `world_coffee` and `world_coffee2`: rows in the original dataset with no match are assigned `NA` values for the new coffee production variables. What if we only want to keep countries that have a match in the key variable? In that case, an inner join can be used.

```
world_coffee_inner = inner_join(world, coffee_data)
#> Joining with `by = join_by(name_long)`
nrow(world_coffee_inner)
#> [1] 45
```

Note that the result of `inner_join()` has only 45 rows compared with 47 in `coffee_data`. What happened to the remaining rows? We can identify the rows that did not match using the `setdiff()` function as follows:

```
setdiff(coffee_data$name_long, world$name_long)
#> [1] "Congo, Dem. Rep. of" "Others"
```

The result shows that `Others` accounts for one row not present in the `world` dataset and that the name of the `Democratic Republic of the Congo` accounts for the other: it has been abbreviated, causing the join to miss it. The following command uses a string matching (*regex*) function from the **stringr** package to confirm what `Congo, Dem. Rep. of` should be.

```
drc = stringr::str_subset(world$name_long, "Dem*.+Congo")
drc
#> [1] "Democratic Republic of the Congo"
```

To fix this issue, we will create a new version of `coffee_data` and update the name. `inner_join()`ing the updated data frame returns a result with all 46 coffee-producing nations.

```
coffee_data$name_long[grepl("Congo,", coffee_data$name_long)] = drc
world_coffee_match = inner_join(world, coffee_data)
#> Joining with `by = join_by(name_long)`
nrow(world_coffee_match)
#> [1] 46
```

It is also possible to join in the other direction: starting with a non-spatial dataset and adding variables from a simple features object. This is demonstrated below, which starts with the `coffee_data` object and adds variables from the original `world` dataset. In contrast with the previous joins, the result is *not* another simple feature object, but a data frame in the form of a **tidyverse** tibble: the output of a join tends to match its first argument.

```
coffee_world = left_join(coffee_data, world)
#> Joining with `by = join_by(name_long)`
class(coffee_world)
#> [1] "tbl_df"     "tbl"        "data.frame"
```

In most cases, the geometry column is only useful in an `sf` object. The geometry column can only be used for creating maps and spatial operations if R 'knows' it is a spatial object, defined by a spatial package such as **sf**. Fortunately, non-spatial data frames with a geometry list column (like `coffee_world`) can be coerced into an `sf` object as follows: `st_as_sf(coffee_world)`.

This section covers the majority of joining use cases. For more information, we recommend reading the chapter Relational data in Grolemund and Wickham (2016), the join vignette in the **geocompkg** package that accompanies this book, and documentation describing joins with **data.table** and other packages. Additionally, spatial joins are covered in the next chapter (Section 4.2.5).

3.2.5 Creating attributes and removing spatial information

Often, we would like to create a new column based on already existing columns. For example, we want to calculate population density for each country. For this we need to divide a population column, here `pop`, by an area column, here `area_km2` with unit area in square kilometers. Using base R, we can type:

```
world_new = world # do not overwrite our original data
world_new$pop_dens = world_new$pop / world_new$area_km2
```

Alternatively, we can use one of **dplyr** functions: `mutate()` or `transmute()`. `mutate()` adds new columns at the penultimate position in the `sf` object (the last one is reserved for the geometry):

```
world_new2 = world |>
  mutate(pop_dens = pop / area_km2)
```

The difference between `mutate()` and `transmute()` is that the latter drops all other existing columns (except for the sticky geometry column).

`unite()` from the **tidyr** package (which provides many useful functions for reshaping datasets, including `pivot_longer()`) pastes together existing columns. For example, we want to combine the `continent` and `region_un` columns into a new column named `con_reg`. Additionally, we can define a separator (here, a colon `:`) which defines how the values of the input columns should be joined, and if the original columns should be removed (here, `TRUE`).

```
world_unite = world |>
  tidyr::unite("con_reg", continent:region_un, sep = ":", remove = TRUE)
```

The resulting `sf` object has a new column called `con_reg` representing the continent and region of each country, e.g., `South America:Americas` for Argentina and other South America countries. **tidyr**'s `separate()` function does the opposite of `unite()`: it splits one column into multiple columns using either a regular expression or character positions.

```
world_separate = world_unite |>
  tidyr::separate(con_reg, c("continent", "region_un"), sep = ":")
```

The **dplyr** function `rename()` and the base R function `setNames()` are useful for renaming columns. The first replaces an old name with a new one. The following command, for example, renames the lengthy `name_long` column to simply `name`:

```
world |>
  rename(name = name_long)
```

`setNames()` changes all column names at once, and requires a character vector with a name matching each column. This is illustrated below, which outputs the same `world` object, but with very short names:

```
new_names = c("i", "n", "c", "r", "s", "t", "a", "p", "l", "gP", "geom")
world_new_names = world |>
  setNames(new_names)
```

Each of these attribute data operations preserves the geometry of the simple features. Sometimes, it makes sense to remove the geometry, for example to speed up aggregation. Do this with `st_drop_geometry()`, **not** manually with commands such as `select(world, -geom)`, as shown below.[1]

[1] `st_geometry(world_st) = NULL` also works to remove the geometry from `world`, but overwrites the original object.

```
world_data = world |> st_drop_geometry()
class(world_data)
#> [1] "tbl_df"     "tbl"        "data.frame"
```

3.3 Manipulating raster objects

In contrast to the vector data model underlying simple features (which represents points, lines and polygons as discrete entities in space), raster data represent continuous surfaces. This section shows how raster objects work by creating them *from scratch*, building on Section 2.3.2. Because of their unique structure, subsetting and other operations on raster datasets work in a different way, as demonstrated in Section 3.3.1.

The following code recreates the raster dataset used in Section 2.3.4, the result of which is illustrated in Figure 3.2. This demonstrates how the `rast()` function works to create an example raster named `elev` (representing elevations).

```
elev = rast(nrows = 6, ncols = 6,
            xmin = -1.5, xmax = 1.5, ymin = -1.5, ymax = 1.5,
            vals = 1:36)
```

The result is a raster object with 6 rows and 6 columns (specified by the `nrow` and `ncol` arguments), and a minimum and maximum spatial extent in x and y direction (`xmin`, `xmax`, `ymin`, `ymax`). The `vals` argument sets the values that each cell contains: numeric data ranging from 1 to 36 in this case.

Raster objects can also contain categorical values of class `logical` or `factor` variables in R. The following code creates the raster datasets shown in Figure 3.2:

```
grain_order = c("clay", "silt", "sand")
grain_char = sample(grain_order, 36, replace = TRUE)
grain_fact = factor(grain_char, levels = grain_order)
grain = rast(nrows = 6, ncols = 6,
             xmin = -1.5, xmax = 1.5, ymin = -1.5, ymax = 1.5,
             vals = grain_fact)
```

The raster object stores the corresponding look-up table or "Raster Attribute Table" (RAT) as a list of data frames, which can be viewed with `cats(grain)` (see `?cats()` for more information). Each element of this list is a layer of the

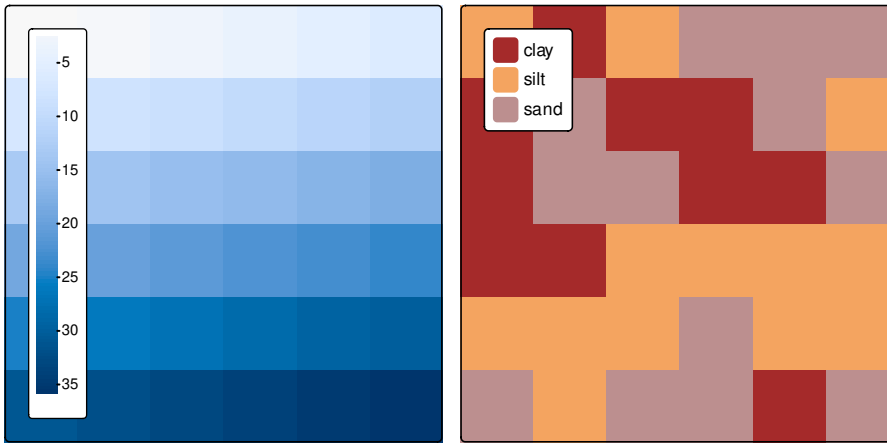

FIGURE 3.2 Raster datasets with numeric (left) and categorical values (right).

raster. It is also possible to use the function `levels()` for retrieving and adding new or replacing existing factor levels.

```
grain2 = grain # do not overwrite the original data
levels(grain2) = data.frame(value = c(0, 1, 2), wetness = c("wet", "moist", "dry"))
levels(grain2)
#> [[1]]
#>   value wetness
#> 1     0     wet
#> 2     1   moist
#> 3     2     dry
```

Categorical raster objects can also store information about the colors associated with each value using a color table. The color table is a data frame with three (red, green, blue) or four (alpha) columns, where each row relates to one value. Color tables in **terra** can be viewed or set with the `coltab()` function (see `?coltab`). Importantly, saving a raster object with a color table to a file (e.g., GeoTIFF) will also save the color information.

3.3.1 Raster subsetting

Raster subsetting is done with the base R operator `[`, which accepts a variety of inputs:

- Row-column indexing
- Cell IDs
- Coordinates
- Another spatial object

Here, we only show the first two options since these can be considered non-spatial operations. If we need a spatial object to subset another or the output is a spatial object, we refer to this as spatial subsetting. Therefore, the latter two options will be shown in the next chapter (see Section 4.3.1).

The first two subsetting options are demonstrated in the commands below — both return the value of the top left pixel in the raster object `elev` (results not shown).

```
# row 1, column 1
elev[1, 1]
# cell ID 1
elev[1]
```

Subsetting of multi-layered raster objects will return the cell value(s) for each layer. For example, `two_layers = c(grain, elev); two_layers[1]` returns a data frame with one row and two columns — one for each layer. To extract all values, you can also use `values()`.

Cell values can be modified by overwriting existing values in conjunction with a subsetting operation. The following code chunk, for example, sets the upper left cell of `elev` to 0 (results not shown):

```
elev[1, 1] = 0
elev[]
```

Leaving the square brackets empty is a shortcut version of `values()` for retrieving all values of a raster. Multiple cells can also be modified in this way:

```
elev[1, c(1, 2)] = 0
```

Replacing values of multi-layered rasters can be done with a matrix with as many columns as layers and rows as replaceable cells (results not shown):

```
two_layers = c(grain, elev)
two_layers[1] = cbind(c(1), c(4))
two_layers[]
```

3.3.2 Summarizing raster objects

terra contains functions for extracting descriptive statistics for entire rasters. Printing a raster object to the console by typing its name returns minimum and maximum values of a raster. `summary()` provides common descriptive statistics – minimum, maximum, quartiles and number of `NA`s for continuous rasters and the number of cells of each class for categorical rasters. Further summary operations such as the standard deviation (see below) or custom summary statistics can be calculated with `global()`.

```
global(elev, sd)
```

If you provide the `summary()` and `global()` functions with a multi-layered raster object, they will summarize each layer separately, as can be illustrated by running: `summary(c(elev, grain))`.

Additionally, the `freq()` function allows to get the frequency table of categorical values.

```
freq(grain)
#>   layer value count
#> 1     1  clay    10
#> 2     1  silt    13
#> 3     1  sand    13
```

Raster value statistics can be visualized in a variety of ways. Specific functions such as `boxplot()`, `density()`, `hist()` and `pairs()` work also with raster objects, as demonstrated in the histogram created with the command below (not shown).

```
hist(elev)
```

In case the desired visualization function does not work with raster objects, one can extract the raster data to be plotted with the help of `values()` (Section 3.3.1).

Descriptive raster statistics belong to the so-called global raster operations. These and other typical raster processing operations are part of the map algebra scheme, which are covered in the next chapter (Section 4.3.2).

Some function names clash between packages (e.g., functions with the name `extract()` exist in both **terra** and **tidyr** packages). This may lead to unexpected results when loading packages in a different order. In addition to calling functions verbosely with their full namespace (e.g., `tidyr::extract()`)

to avoid attaching packages with `library()`, another way to prevent function name clashes is by unloading the offending package with `detach()`. The following command, for example, unloads the **terra** package (this can also be done in the *package* tab which resides by default in the right-bottom pane in RStudio): `detach("package:terra", unload = TRUE, force = TRUE)`. The `force` argument makes sure that the package will be detached even if other packages depend on it. This, however, may lead to a restricted usability of packages depending on the detached package, and it is therefore not recommended.

3.4 Exercises

For these exercises we will use the `us_states` and `us_states_df` datasets from the **spData** package. You must have attached the package, and other packages used in the attribute operations chapter (**sf**, **dplyr**, **terra**) with commands such as `library(spData)` before attempting these exercises:

```
library(sf)
library(dplyr)
library(terra)
library(spData)
data(us_states)
data(us_states_df)
```

`us_states` is a spatial object (of class `sf`), containing geometry and a few attributes (including name, region, area, and population) of states within the contiguous United States. `us_states_df` is a data frame (of class `data.frame`) containing the name and additional variables (including median income and poverty level, for the years 2010 and 2015) of US states, including Alaska, Hawaii and Puerto Rico. The data comes from the United States Census Bureau, and is documented in `?us_states` and `?us_states_df`.

E1. Create a new object called `us_states_name` that contains only the `NAME` column from the `us_states` object using either base R (`[`) or tidyverse (`select()`) syntax. What is the class of the new object and what makes it geographic?

E2. Select columns from the `us_states` object which contain population data. Obtain the same result using a different command (bonus: try to find three ways of obtaining the same result). Hint: try to use helper functions, such as `contains` or `matches` from **dplyr** (see `?contains`).

E3. Find all states with the following characteristics (bonus: find *and* plot them):

- Belong to the Midwest region.
- Belong to the West region, have an area below 250,000 km^2 *and* in 2015 a population greater than 5,000,000 residents (hint: you may need to use the function `units::set_units()` or `as.numeric()`).
- Belong to the South region, had an area larger than 150,000 km^2 and a total population in 2015 larger than 7,000,000 residents.

E4. What was the total population in 2015 in the `us_states` dataset? What was the minimum and maximum total population in 2015?

E5. How many states are there in each region?

E6. What was the minimum and maximum total population in 2015 in each region? What was the total population in 2015 in each region?

E7. Add variables from `us_states_df` to `us_states`, and create a new object called `us_states_stats`. What function did you use and why? Which variable is the key in both datasets? What is the class of the new object?

E8. `us_states_df` has two more rows than `us_states`. How can you find them? (Hint: try to use the `dplyr::anti_join()` function.)

E9. What was the population density in 2015 in each state? What was the population density in 2010 in each state?

E10. How much has population density changed between 2010 and 2015 in each state? Calculate the change in percentages and map them.

E11. Change the columns' names in `us_states` to lowercase. (Hint: helper functions - `tolower()` and `colnames()` may help.)

E12. Using `us_states` and `us_states_df` create a new object called `us_states_sel`. The new object should have only two variables: `median_income_15` and `geometry`. Change the name of the `median_income_15` column to `Income`.

E13. Calculate the change in the number of residents living below the poverty level between 2010 and 2015 for each state. (Hint: See ?us_states_df for documentation on the poverty level columns.) Bonus: Calculate the change in the *percentage* of residents living below the poverty level in each state.

E14. What was the minimum, average and maximum state's number of people living below the poverty line in 2015 for each region? Bonus: What is the region with the largest increase in people living below the poverty line?

E15. Create a raster from scratch, with nine rows and columns and a resolution of 0.5 decimal degrees (WGS84). Fill it with random numbers. Extract the values of the four corner cells.

E16. What is the most common class of our example raster `grain`?

E17. Plot the histogram and the boxplot of the `dem.tif` file from the **spDataLarge** package (`system.file("raster/dem.tif", package = "spDataLarge")`).

4

Spatial data operations

Prerequisites

- This chapter requires the same packages used in Chapter 3:

```
library(sf)
library(terra)
library(dplyr)
library(spData)
```

4.1 Introduction

Spatial operations, including spatial joins between vector datasets and local and focal operations on raster datasets, are a vital part of geocomputation. This chapter shows how spatial objects can be modified in a multitude of ways based on their location and shape. Many spatial operations have a non-spatial (attribute) equivalent, so concepts such as subsetting and joining datasets demonstrated in the previous chapter are applicable here. This is especially true for *vector* operations: Section 3.2 on vector attribute manipulation provides the basis for understanding its spatial counterpart, namely spatial subsetting (covered in Section 4.2.1). Spatial joining (Sections 4.2.5, 4.2.6 and 4.2.8) and aggregation (Section 4.2.7) also have non-spatial counterparts, covered in the previous chapter.

Spatial operations differ from non-spatial operations in a number of ways, however: spatial joins, for example, can be done in a number of ways — including matching entities that intersect with or are within a certain distance of the target dataset — while the attribution joins discussed in Section 3.2.4 in the previous chapter can only be done in one way (except when using fuzzy joins, as described in the documentation of the **fuzzyjoin** package). Different *types* of spatial relationship between objects, including intersects and disjoint, are described in Sections 4.2.2 and 4.2.4. Another unique aspect of spatial objects

is distance: all spatial objects are related through space, and distance calculations can be used to explore the strength of this relationship, as described in the context of vector data in Section 4.2.3.

Spatial operations on raster objects include subsetting — covered in Section 4.3.1. *Map algebra* covers a range of operations that modify raster cell values, with or without reference to surrounding cell values. The concept of map algebra, vital for many applications, is introduced in Section 4.3.2; local, focal and zonal map algebra operations are covered in sections 4.3.3, 4.3.4, and 4.3.5, respectively. Global map algebra operations, which generate summary statistics representing an entire raster dataset, and distance calculations on rasters, are discussed in Section 4.3.6. Next, the relationships between map algebra and vector operations are discussed in Section 4.3.7. In the Section 4.3.8,1 the process of merging two raster datasets is discussed and demonstrated with reference to a reproducible example.

It is important to note that spatial operations that use two spatial objects rely on both objects having the same coordinate reference system, a topic that was introduced in Section 2.4 and which will be covered in more depth in Chapter 7.

4.2 Spatial operations on vector data

This section provides an overview of spatial operations on vector geographic data represented as simple features in the **sf** package. Section 4.3 presents spatial operations on raster datasets using classes and functions from the **terra** package.

4.2.1 Spatial subsetting

Spatial subsetting is the process of taking a spatial object and returning a new object containing only features that *relate* in space to another object. Analogous to *attribute subsetting* (covered in Section 3.2.1), subsets of `sf` data frames can be created with square bracket (`[`) operator using the syntax `x[y, , op = st_intersects]`, where `x` is an `sf` object from which a subset of rows will be returned, `y` is the 'subsetting object' and `, op = st_intersects` is an optional argument that specifies the topological relation (also known as the binary predicate) used to do the subsetting. The default topological relation used when an `op` argument is not provided is `st_intersects()`: the command `x[y, ]` is identical to `x[y, , op = st_intersects]` shown above but not `x[y, , op = st_disjoint]` (the meaning of these and other topological relations is described

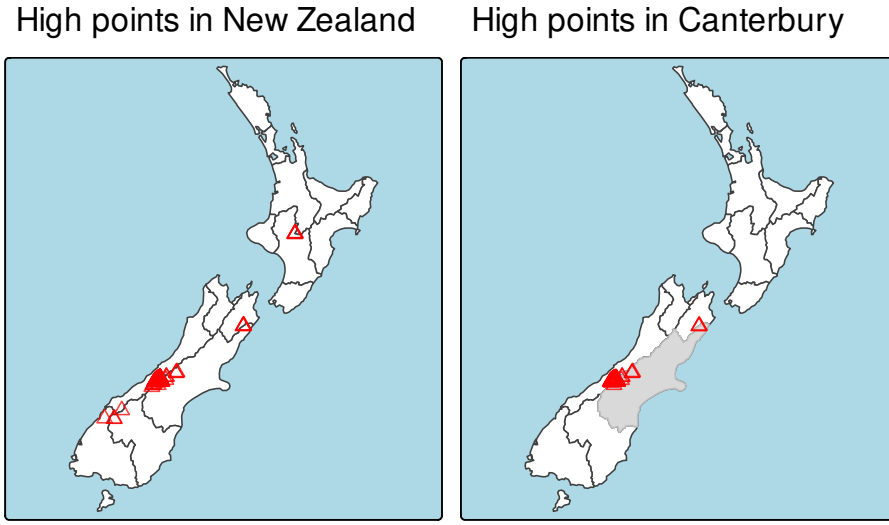

FIGURE 4.1 Spatial subsetting, with red triangles representing 101 high points in New Zealand, clustered near the central Canterbuy region (left). The points in Canterbury were created with the '[' subsetting operator (highlighted in gray, right).

in the next section). The `filter()` function from the **tidyverse** can also be used, but this approach is more verbose, as we will see in the examples below.

To demonstrate spatial subsetting, we will use the `nz` and `nz_height` datasets in the **spData** package, which contain geographic data on the 16 main regions and 101 highest points in New Zealand, respectively (Figure 4.1), in a projected coordinate reference system. The following code chunk creates an object representing Canterbury, then uses spatial subsetting to return all high points in the region.

```
canterbury = nz |> filter(Name == "Canterbury")
canterbury_height = nz_height[canterbury, ]
```

Like attribute subsetting, the command `x[y, ]` (equivalent to `nz_height[canterbury, ]`) subsets features of a *target* `x` using the contents of a *source* object `y`. Instead of `y` being a vector of class `logical` or `integer`, however, for spatial subsetting both `x` and `y` must be geographic objects. Specifically, objects used for spatial subsetting in this way must have the class `sf` or `sfc`: both `nz` and `nz_height` are geographic vector data frames and have the class `sf`, and the result of the operation returns another `sf` object representing the features in the target `nz_height` object that intersect with (in this case high points that are located within) the `canterbury` region.

Various *topological relations* can be used for spatial subsetting which determine the type of spatial relationship that features in the target object must have with the subsetting object to be selected. These include *touches*, *crosses* or *within*, as we will see shortly in Section 4.2.2. The default setting `st_intersects` is a 'catch all' topological relation that will return features in the target that *touch*, *cross* or are *within* the source 'subsetting' object. Alternative spatial operators can be specified with the `op =` argument, as demonstrated in the following command which returns the opposite of `st_intersects()`, points that do not intersect with Canterbury (see Section 4.2.2).

```
nz_height[canterbury, , op = st_disjoint]
```

Note the empty argument — denoted with `, ,` — in the preceding code chunk is included to highlight `op`, the third argument in `[` for `sf` objects. One can use this to change the subsetting operation in many ways. `nz_height[canterbury, 2, op = st_disjoint]`, for example, returns the same rows but only includes the second attribute column (see `sf::`[.sf`` and the `?sf` for details).

For many applications, this is all you'll need to know about spatial subsetting for vector data: it just works. If you are impatient to learn about more topological relations, beyond `st_intersects()` and `st_disjoint()`, skip to the next section (4.2.2). If you're interested in the details, including other ways of subsetting, read on.

Another way of doing spatial subsetting uses objects returned by topological operators. These objects can be useful in their own right, for example when exploring the graph network of relationships between contiguous regions, but they can also be used for subsetting, as demonstrated in the code chunk below.

```
sel_sgbp = st_intersects(x = nz_height, y = canterbury)
class(sel_sgbp)
#> [1] "sgbp" "list"
sel_sgbp
#> Sparse geometry binary predicate list of length 101, where the
#> predicate was `intersects'
#> first 10 elements:
#>  1: (empty)
#>  2: (empty)
#>  3: (empty)
#>  4: (empty)
#>  5: 1
#>  6: 1
....
```

```
sel_logical = lengths(sel_sgbp) > 0
canterbury_height2 = nz_height[sel_logical, ]
```

The above code chunk creates an object of class `sgbp` (a sparse geometry binary predicate, a list of length `x` in the spatial operation) and then converts it into a logical vector `sel_logical` (containing only `TRUE` and `FALSE` values, something that can also be used by **dplyr**'s filter function). The function `lengths()` identifies which features in `nz_height` intersect with *any* objects in `y`. In this case, 1 is the greatest possible value, but for more complex operations one could use the method to subset only features that intersect with, for example, 2 or more features from the source object.

Note: another way to return a logical output is by setting `sparse = FALSE` (meaning 'return a dense matrix not a sparse one') in operators such as `st_intersects()`. The command `st_intersects(x = nz_height, y = canterbury, sparse = FALSE)[, 1]`, for example, would return an output identical to `sel_logical`. Note: the solution involving `sgbp` objects is more generalizable though, as it works for many-to-many operations and has lower memory requirements.

The same result can be also achieved with the **sf** function `st_filter()` which was created to increase compatibility between `sf` objects and **dplyr** data manipulation code:

```
canterbury_height3 = nz_height |>
  st_filter(y = canterbury, .predicate = st_intersects)
```

At this point, there are three identical (in all but row names) versions of `canterbury_height`, one created using the `[` operator, one created via an intermediary selection object, and another using **sf**'s convenience function `st_filter()`. The next section explores different types of spatial relation, also known as binary predicates, that can be used to identify whether two features are spatially related or not.

4.2.2 Topological relations

Topological relations describe the spatial relationships between objects. "Binary topological relationships", to give them their full name, are logical statements (in that the answer can only be `TRUE` or `FALSE`) about the spatial relationships between two objects defined by ordered sets of points (typically forming points, lines and polygons) in two or more dimensions (Egenhofer and Herring, 1990). That may sound rather abstract and, indeed, the definition and classification of topological relations is based on mathematical foundations first

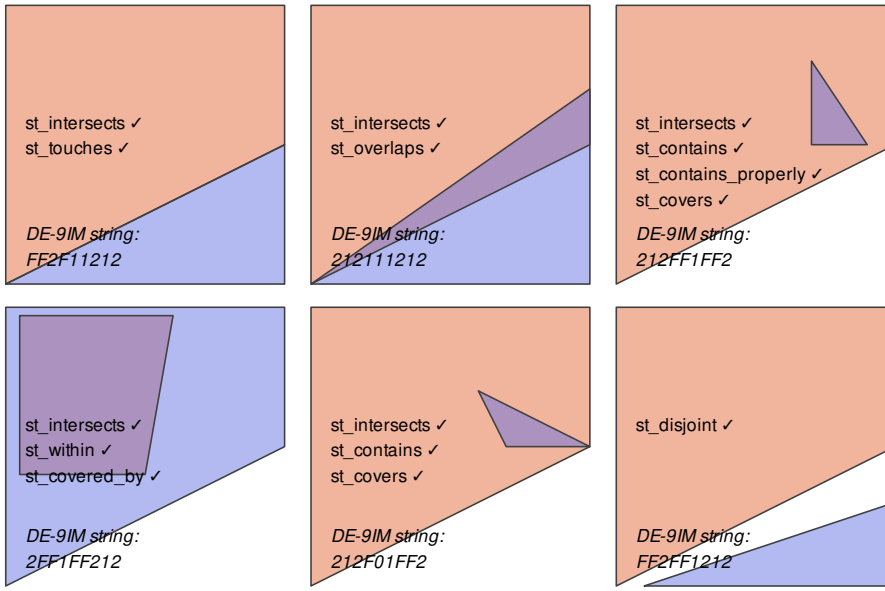

FIGURE 4.2 Topological relations between vector geometries, inspired by figures 1 and 2 in Egenhofer and Herring (1990). The relations for which the function(x, y) is true are printed for each geometry pair, with x represented in pink and y represented in blue. The nature of the spatial relationship for each pair is described by the Dimensionally Extended 9-Intersection Model string.

published in book form in 1966 (Spanier, 1995), with the field of algebraic topology continuing beyond the year 2000 (tom Dieck, 2008).

Despite their mathematical origins, topological relations can be understood intuitively with reference to visualizations of commonly used functions that test for common types of spatial relationships. Figure 4.2 shows a variety of geometry pairs and their associated relations. The third and fourth pairs in Figure 4.2 (from left to right and then down) demonstrate that, for some relations, order is important. While the relations *equals*, *intersects*, *crosses*, *touches* and *overlaps* are symmetrical, meaning that if `function(x, y)` is true, `function(y, x)` will also be true, relations in which the order of the geometries are important such as *contains* and *within* are not. Notice that each geometry pair has a "DE-9IM" string such as FF2F11212, described in the next section.

In `sf`, functions testing for different types of topological relations are called 'binary predicates', as described in the vignette *Manipulating Simple Feature Geometries*, which can be viewed with the command `vignette("sf3")`, and in the help page `?geos_binary_pred`. To see how topological relations work in practice, let's create a simple reproducible example, building on the relations illustrated

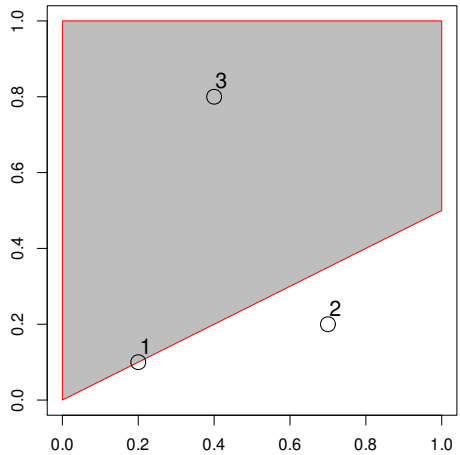

FIGURE 4.3 Points, line and polygon objects arranged to illustrate topological relations.

in Figure 4.2 and consolidating knowledge of how vector geometries are represented from a previous chapter (Section 2.2.4). Note that to create tabular data representing coordinates (x and y) of the polygon vertices, we use the base R function `cbind()` to create a matrix representing coordinates points, a `POLYGON`, and finally an `sfc` object, as described in Chapter 2:

```
polygon_matrix = cbind(
  x = c(0, 0, 1, 1,   0),
  y = c(0, 1, 1, 0.5, 0)
)
polygon_sfc = st_sfc(st_polygon(list(polygon_matrix)))
```

We will create additional geometries to demonstrate spatial relations with the following commands which, when plotted on top of the polygon created above, relate in space to one another, as shown in Figure 4.3. Note the use of the function `st_as_sf()` and the argument `coords` to efficiently convert from a data frame containing columns representing coordinates to an `sf` object containing points:

```
point_df = data.frame(
  x = c(0.2, 0.7, 0.4),
  y = c(0.1, 0.2, 0.8)
)
point_sf = st_as_sf(point_df, coords = c("x", "y"))
```

A simple query is: which of the points in `point_sf` intersect in some way with polygon `polygon_sfc`? The question can be answered by inspection (points 1 and 3 are touching and within the polygon, respectively). This question can be answered with the spatial predicate `st_intersects()` as follows:

```
st_intersects(point_sf, polygon_sfc)
#> Sparse geometry binary predicate... `intersects'
#>  1: 1
#>  2: (empty)
#>  3: 1
```

The result should match your intuition: positive (`1`) results are returned for the first and third point, and a negative result (represented by an empty vector) for the second are outside the polygon's border. What may be unexpected is that the result comes in the form of a list of vectors. This *sparse matrix* output only registers a relation if one exists, reducing the memory requirements of topological operations on multi-feature objects. As we saw in the previous section, a *dense matrix* consisting of `TRUE` or `FALSE` values is returned when `sparse = FALSE`.

```
st_intersects(point_sf, polygon_sfc, sparse = FALSE)
#>       [,1]
#> [1,]  TRUE
#> [2,] FALSE
#> [3,]  TRUE
```

In the above output each row represents a feature in the target (argument `x`) object, and each column represents a feature in the selecting object (`y`). In this case, there is only one feature in the `y` object `polygon_sfc` so the result, which can be used for subsetting as we saw in Section 4.2.1, has only one column.

`st_intersects()` returns `TRUE` even in cases where the features just touch: *intersects* is a 'catch-all' topological operation which identifies many types of spatial relation, as illustrated in Figure 4.2. More restrictive questions include which points lie within the polygon, and which features are on or contain a shared boundary with `y`? These can be answered as follows (results not shown):

```
st_within(point_sf, polygon_sfc)
st_touches(point_sf, polygon_sfc)
```

Note that although the first point *touches* the boundary polygon, it is not within it; the third point is within the polygon but does not touch any part of its border. The opposite of `st_intersects()` is `st_disjoint()`, which returns only objects that do not spatially relate in any way to the selecting object (note `[, 1]` converts the result into a vector).

```
st_disjoint(point_sf, polygon_sfc, sparse = FALSE)[, 1]
#> [1] FALSE  TRUE FALSE
```

The function `st_is_within_distance()` detects features that *almost touch* the selection object, which has an additional `dist` argument. It can be used to set how close target objects need to be before they are selected. The 'is within distance' binary spatial predicate is demonstrated in the code chunk below, the results of which show that every point is within 0.2 units of the polygon.

```
st_is_within_distance(point_sf, polygon_sfc, dist = 0.2, sparse = FALSE)[, 1]
#> [1] TRUE TRUE TRUE
```

Note that although point 2 is more than 0.2 units of distance from the nearest vertex of `polygon_sfc`, it is still selected when the distance is set to 0.2. This is because distance is measured to the nearest edge, in this case the part of the polygon that lies directly above point 2 in Figure 4.3. (You can verify the actual distance between point 2 and the polygon is 0.13 with the command `st_distance(point_sf, polygon_sfc)`.)

Functions for calculating topological relations use spatial indices to largely speed up spatial query performance. They achieve that using the Sort-Tile-Recursive (STR) algorithm. The `st_join` function, mentioned in the next section, also uses the spatial indexing. You can learn more at https://www.r-spatial.org/r/2017/06/22/spatial-index.html.

4.2.3 Distance relations

While the topological relations presented in the previous section are binary (a feature either intersects with another or does not) distance relations are continuous. The distance between two `sf` objects is calculated with `st_distance()`, which is also used behind the scenes in Section 4.2.6 for distance-based joins. This is illustrated in the code chunk below, which finds the distance between the highest point in New Zealand and the geographic centroid of the Canterbury region, created in Section 4.2.1:

```
nz_highest = nz_height |> slice_max(n = 1, order_by = elevation)
canterbury_centroid = st_centroid(canterbury)
st_distance(nz_highest, canterbury_centroid)
#> Units: [m]
#>        [,1]
#> [1,] 115540
```

There are two potentially surprising things about the result:

- It has `units`, telling us the distance is 100,000 meters, not 100,000 inches, or any other measure of distance
- It is returned as a matrix, even though the result only contains a single value

This second feature hints at another useful feature of `st_distance()`, its ability to return *distance matrices* between all combinations of features in objects `x` and `y`. This is illustrated in the command below, which finds the distances between the first three features in `nz_height` and the Otago and Canterbury regions of New Zealand represented by the object `co`.

```
co = filter(nz, grepl("Canter|Otag", Name))
st_distance(nz_height[1:3, ], co)
#> Units: [m]
#>        [,1]  [,2]
#> [1,] 123537 15498
#> [2,]  94283     0
#> [3,]  93019     0
```

Note that the distance between the second and third features in `nz_height` and the second feature in `co` is zero. This demonstrates the fact that distances between points and polygons refer to the distance to *any part of the polygon*. The second and third points in `nz_height` are *in* Otago, which can be verified by plotting them (result not shown):

```
plot(st_geometry(co)[2])
plot(st_geometry(nz_height)[2:3], add = TRUE)
```

4.2.4 DE-9IM strings

Underlying the binary predicates demonstrated in the previous section is the Dimensionally Extended 9-Intersection Model (DE-9IM). As the cryptic name suggests, this is not an easy topic to understand, but it is worth knowing about because it underlies many spatial operations and enables the creation of custom spatial predicates. The model was originally labelled "DE + 9IM" by its inventors, referring to the "dimension of the intersections of boundaries, interiors, and exteriors of two features" (Clementini and Di Felice, 1995), but it is now referred to as DE-9IM (Shen et al., 2018). DE-9IM is applicable to two-dimensional objects (points, lines and polygons) in Euclidean space, meaning that the model (and software implementing it such as GEOS) assumes you are working with data in a projected coordinate reference system, described in Chapter 7.

To demonstrate how DE-9IM strings work, let's take a look at the various ways that the first geometry pair in Figure 4.2 relate. Figure 4.4 illustrates

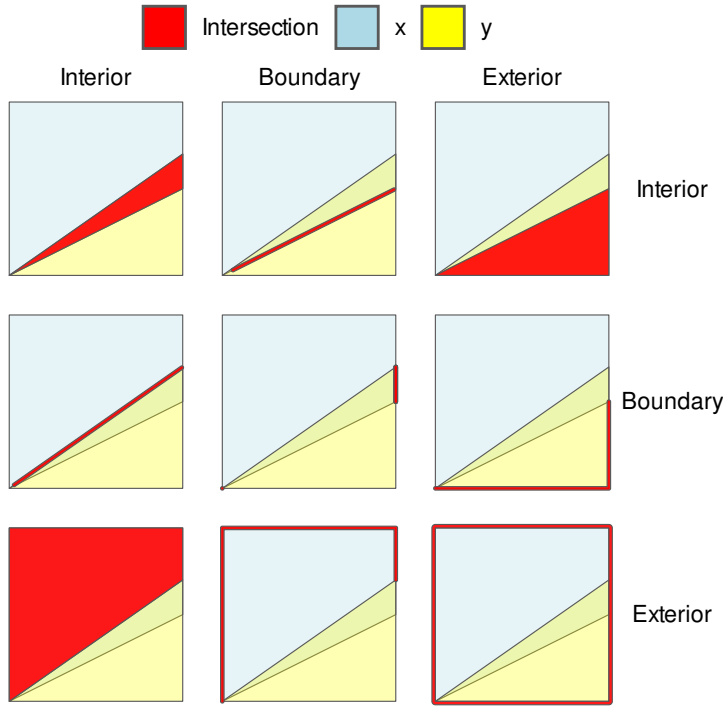

FIGURE 4.4 Illustration of how the Dimensionally Extended 9 Intersection Model (DE-9IM) works. Colors not in the legend represent the overlap between different components. The thick lines highlight two-dimensional intersections, e.g., between the boundary of object x and the interior of object y, shown in the middle top facet.

the 9-intersection model (9IM) which shows the intersections between every combination of each object's interior, boundary and exterior: when each component of the first object `x` is arranged as columns, and each component of `y` is arranged as rows, a facetted graphic is created with the intersections between each element highlighted.

DE-9IM strings are derived from the dimension of each type of relation. In this case, the red intersections in Figure 4.4 have dimensions of 0 (points), 1 (lines), and 2 (polygons), as shown in Table 4.1.

Flattening this matrix 'row-wise' (meaning concatenating the first row, then the second, then the third) results in the string `212111212`. Another example will serve to demonstrate the system: the relation shown in Figure 4.2 (the third polygon pair in the third column and 1st row) can be defined in the DE-9IM system as follows:

- The intersections between the *interior* of the larger object `x` and the interior, boundary and exterior of `y` have dimensions of 2, 1 and 2, respectively

TABLE 4.1 Relations between interiors, boundaries and exteriors of geometries x and y.

	Interior (x)	Boundary (x)	Exterior (x)
Interior (y)	2	1	2
Boundary (y)	1	1	1
Exterior (y)	2	1	2

- The intersections between the *boundary* of the larger object `x` and the interior, boundary and exterior of `y` have dimensions of F, F and 1, respectively, where 'F' means 'false', the objects are disjoint
- The intersections between the *exterior* of `x` and the interior, boundary and exterior of `y` have dimensions of F, F and 2, respectively: the exterior of the larger object does not touch the interior or boundary of `y`, but the exterior of the smaller and larger objects cover the same area

These three components, when concatenated, create the string `212`, `FF1`, and `FF2`. This is the same as the result obtained from the function `st_relate()` (see the source code of this chapter to see how other geometries in Figure 4.2 were created):

```
xy2sfc = function(x, y) st_sfc(st_polygon(list(cbind(x, y))))
x = xy2sfc(x = c(0, 0, 1, 1, 0), y = c(0, 1, 1, 0.5, 0))
y = xy2sfc(x = c(0.7, 0.7, 0.9, 0.7), y = c(0.8, 0.5, 0.5, 0.8))
st_relate(x, y)
#>      [,1]
#> [1,] "212FF1FF2"
```

Understanding DE-9IM strings allows new binary spatial predicates to be developed. The help page `?st_relate` contains function definitions for 'queen' and 'rook' relations in which polygons share a border or only a point, respectively. 'Queen' relations mean that 'boundary-boundary' relations (the cell in the second column and the second row in Table 4.1, or the 5th element of the DE-9IM string) must not be empty, corresponding to the pattern `F***T****`, while for 'rook' relations, the same element must be 1 (meaning a linear intersection) (see Figure 4.5). These are implemented as follows:

```
st_queen = function(x, y) st_relate(x, y, pattern = "F***T****")
st_rook = function(x, y) st_relate(x, y, pattern = "F***1****")
```

Building on the object `x` created previously, we can use the newly created functions to find out which elements in the grid are a 'queen' and 'rook' in relation to the middle square of the grid as follows:

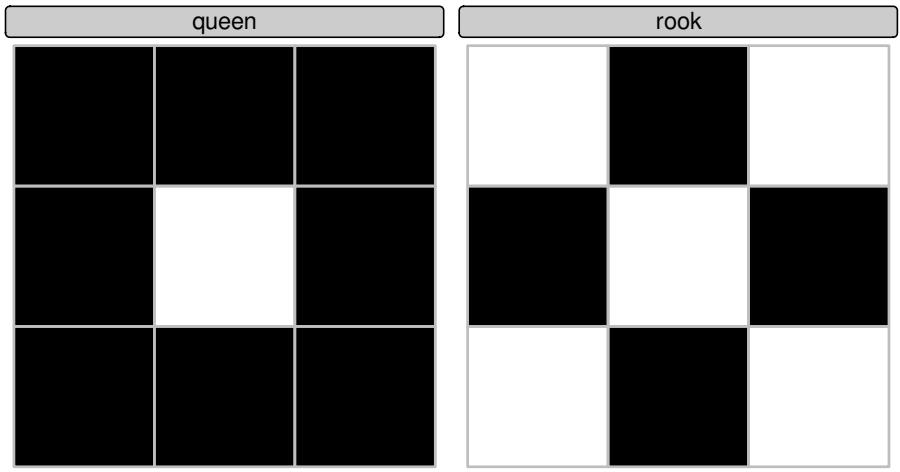

FIGURE 4.5 Demonstration of custom binary spatial predicates for finding queen (left) and rook (right) relations to the central square in a grid with 9 geometries.

```
grid = st_make_grid(x, n = 3)
grid_sf = st_sf(grid)
grid_sf$queens = lengths(st_queen(grid, grid[5])) > 0
plot(grid, col = grid_sf$queens)
grid_sf$rooks = lengths(st_rook(grid, grid[5])) > 0
plot(grid, col = grid_sf$rooks)
```

4.2.5 Spatial joining

Joining two non-spatial datasets relies on a shared 'key' variable, as described in Section 3.2.4. Spatial data joining applies the same concept, but instead relies on spatial relations, described in the previous section. As with attribute data, joining adds new columns to the target object (the argument `x` in joining functions), from a source object (`y`).

The process is illustrated by the following example: imagine you have ten points randomly distributed across the Earth's surface and you ask, for the points that are on land, which countries are they in? Implementing this idea in a reproducible example will build your geographic data-handling skills and will showhow spatial joins work. The starting point is to create points that are randomly scattered over the Earth's surface.

```
set.seed(2018) # set seed for reproducibility
(bb = st_bbox(world)) # the world's bounds
#>   xmin   ymin   xmax   ymax
#> -180.0  -89.9  180.0   83.6
random_df = data.frame(
  x = runif(n = 10, min = bb[1], max = bb[3]),
  y = runif(n = 10, min = bb[2], max = bb[4])
)
random_points = random_df |>
  st_as_sf(coords = c("x", "y"), crs = "EPSG:4326") # set coordinates and CRS
```

The scenario illustrated in Figure 4.6 shows that the `random_points` object (top left) lacks attribute data, while the `world` (top right) has attributes, including country names shown for a sample of countries in the legend. Spatial joins are implemented with `st_join()`, as illustrated in the code chunk below. The output is the `random_joined` object which is illustrated in Figure 4.6 (bottom left). Before creating the joined dataset, we use spatial subsetting to create `world_random`, which contains only countries that contain random points, to verify that number of country names returned in the joined dataset should be four (Figure 4.6, top right panel).

```
world_random = world[random_points, ]
nrow(world_random)
#> [1] 4
random_joined = st_join(random_points, world["name_long"])
```

By default, `st_join()` performs a left join, meaning that the result is an object containing all rows from `x` including rows with no match in `y` (see Section 3.2.4), but it can also do inner joins by setting the argument `left = FALSE`. Like spatial subsetting, the default topological operator used by `st_join()` is `st_intersects()`, which can be changed by setting the `join` argument (see `?st_join` for details). The example above demonstrates the addition of a column from a polygon layer to a point layer, but the approach works regardless of geometry types. In such cases, for example when `x` contains polygons, each of which matches multiple objects in `y`, spatial joins will result in duplicate features by creating a new row for each match in `y`.

4.2.6 Distance-based joins

Sometimes two geographic datasets do not intersect but still have a strong geographic relationship due to their proximity. The datasets `cycle_hire` and `cycle_hire_osm`, already attached in the **spData** package, provide a good example. Plotting them shows that they are often closely related, but they do

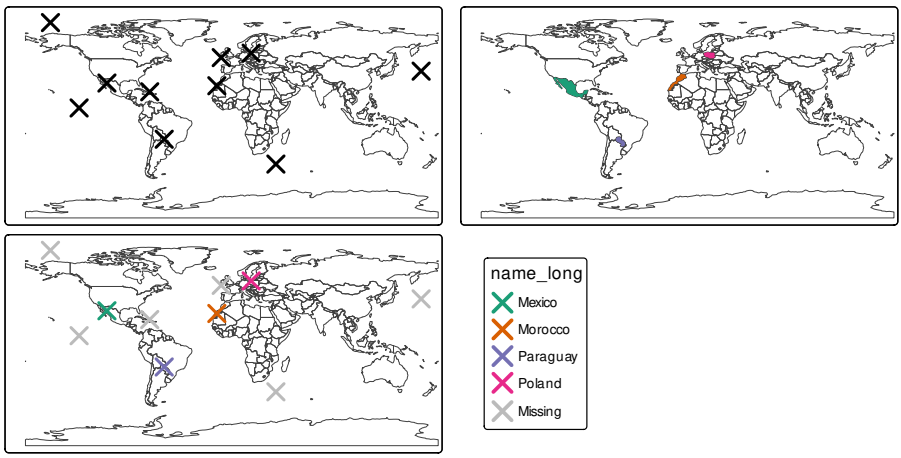

FIGURE 4.6 Illustration of a spatial join. A new attribute variable is added to random points (top left) from source world object (top right) resulting in the data represented in the final panel.

not touch, as shown in Figure 4.7, a base version of which is created with the following code below:

```
plot(st_geometry(cycle_hire), col = "blue")
plot(st_geometry(cycle_hire_osm), add = TRUE, pch = 3, col = "red")
```

We can check if any points are the same using `st_intersects()` as shown below:

```
any(st_intersects(cycle_hire, cycle_hire_osm, sparse = FALSE))
#> [1] FALSE
```

Imagine that we need to join the `capacity` variable in `cycle_hire_osm` onto the official 'target' data contained in `cycle_hire`. This is when a non-overlapping join is needed. The simplest method is to use the binary predicate `st_is_within_distance()`, as demonstrated below using a threshold distance of 20 m. One can set the threshold distance in metric units also for unprojected data (e.g., lon/lat CRSs such as WGS84), if the spherical geometry engine (S2) is enabled, as it is in **sf** by default (see Section 2.2.9).

```
sel = st_is_within_distance(cycle_hire, cycle_hire_osm,
                            dist = units::set_units(20, "m"))
summary(lengths(sel) > 0)
#>    Mode   FALSE    TRUE
#> logical     304     438
```

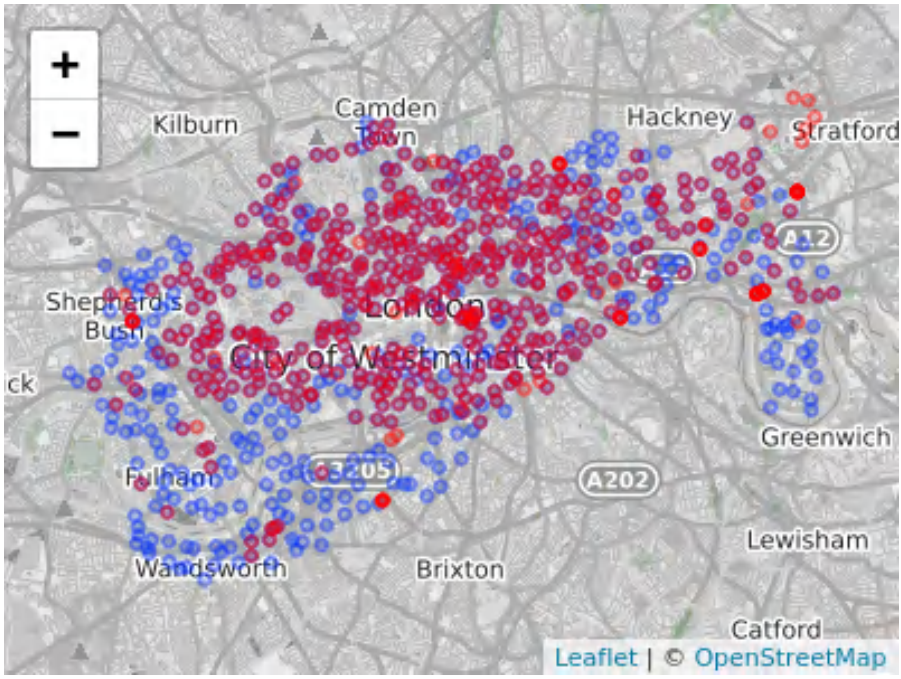

FIGURE 4.7 The spatial distribution of cycle hire points in London based on official data (blue) and OpenStreetMap data (red).

This shows that there are 438 points in the target object `cycle_hire` within the threshold distance of `cycle_hire_osm`. How to retrieve the *values* associated with the respective `cycle_hire_osm` points? The solution is again with `st_join()`, but with an additional `dist` argument (set to 20 m below):

```
z = st_join(cycle_hire, cycle_hire_osm, st_is_within_distance,
            dist = units::set_units(20, "m"))
nrow(cycle_hire)
#> [1] 742
nrow(z)
#> [1] 762
```

Note that the number of rows in the joined result is greater than the target. This is because some cycle hire stations in `cycle_hire` have multiple matches in `cycle_hire_osm`. To aggregate the values for the overlapping points and return the mean, we can use the aggregation methods learned in Chapter 3, resulting in an object with the same number of rows as the target.

```
z = z |>
  group_by(id) |>
  summarize(capacity = mean(capacity))
nrow(z) == nrow(cycle_hire)
#> [1] TRUE
```

The capacity of nearby stations can be verified by comparing a plot of the capacity of the source `cycle_hire_osm` data with the results in this new object (plots not shown):

```
plot(cycle_hire_osm["capacity"])
plot(z["capacity"])
```

The result of this join has used a spatial operation to change the attribute data associated with simple features; the geometry associated with each feature has remained unchanged.

4.2.7 Spatial aggregation

As with attribute data aggregation, spatial data aggregation *condenses* data: aggregated outputs have fewer rows than non-aggregated inputs. Statistical *aggregating functions*, such as mean average or sum, summarize multiple values of a variable, and return a single value per *grouping variable*. Section 3.2.3 demonstrated how `aggregate()` and `group_by() |> summarize()` condense data based on attribute variables, this section shows how the same functions work with spatial objects.

Returning to the example of New Zealand, imagine you want to find out the average height of high points in each region: it is the geometry of the source (`y` or `nz` in this case) that defines how values in the target object (`x` or `nz_height`) are grouped. This can be done in a single line of code with base R's `aggregate()` method.

```
nz_agg = aggregate(x = nz_height, by = nz, FUN = mean)
```

The result of the previous command is an `sf` object with the same geometry as the (spatial) aggregating object (`nz`), which you can verify with the command `identical(st_geometry(nz), st_geometry(nz_agg))`. The result of the previous operation is illustrated in Figure 4.8, which shows the average value of features in `nz_height` within each of New Zealand's 16 regions. The same result can also be generated by piping the output from `st_join()` into the 'tidy' functions `group_by()` and `summarize()` as follows:

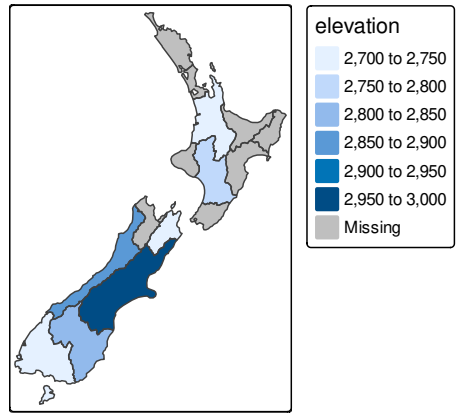

FIGURE 4.8 Average height of the top 101 high points across the regions of New Zealand.

```
nz_agg2 = st_join(x = nz, y = nz_height) |>
  group_by(Name) |>
  summarize(elevation = mean(elevation, na.rm = TRUE))
```

The resulting `nz_agg` objects have the same geometry as the aggregating object `nz` but with a new column summarizing the values of `x` in each region using the function `mean()`. Other functions could be used instead of `mean()` here, including `median()`, `sd()` and other functions that return a single value per group. Note: one difference between the `aggregate()` and `group_by() |> summarize()` approaches is that the former results in `NA` values for unmatching region names, while the latter preserves region names. The 'tidy' approach is thus more flexible in terms of aggregating functions and the column names of the results. Aggregating operations that also create new geometries are covered in Section 5.2.7.

4.2.8 Joining incongruent layers

Spatial congruence is an important concept related to spatial aggregation. An *aggregating object* (which we will refer to as `y`) is *congruent* with the target object (`x`) if the two objects have shared borders. Often this is the case for administrative boundary data, whereby larger units — such as Middle Layer Super Output Areas (MSOAs) in the UK or districts in many other European countries — are composed of many smaller units.

Incongruent aggregating objects, by contrast, do not share common borders with the target (Qiu et al., 2012). This is problematic for spatial aggregation (and other spatial operations) illustrated in Figure 4.9: aggregating the cen-

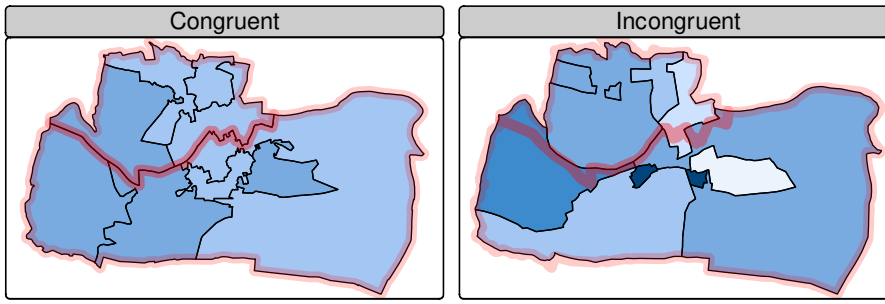

FIGURE 4.9 Congruent (left) and incongruent (right) areal units with respect to larger aggregating zones (translucent red borders).

troid of each sub-zone will not return accurate results. Areal interpolation overcomes this issue by transferring values from one set of areal units to another, using a range of algorithms including simple area weighted approaches and more sophisticated approaches such as 'pycnophylactic' methods (Tobler, 1979).

The **spData** package contains a dataset named `incongruent` (colored polygons with black borders in the right panel of Figure 4.9) and a dataset named `aggregating_zones` (the two polygons with the translucent blue border in the right panel of Figure 4.9). Let us assume that the `value` column of `incongruent` refers to the total regional income in million Euros. How can we transfer the values of the underlying nine spatial polygons into the two polygons of `aggregating_zones`?

The simplest useful method for this is *area weighted* spatial interpolation, which transfers values from the `incongruent` object to a new column in `aggregating_zones` in proportion with the area of overlap: the larger the spatial intersection between input and output features, the larger the corresponding value. This is implemented in `st_interpolate_aw()`, as demonstrated in the code chunk below.

```
iv = incongruent["value"] # keep only the values to be transferred
agg_aw = st_interpolate_aw(iv, aggregating_zones, extensive = TRUE)
#> Warning in st_interpolate_aw.sf(iv, aggregating_zones, extensive = TRUE):
#> st_interpolate_aw assumes attributes are constant or uniform over areas of x
agg_aw$value
#> [1] 19.6 25.7
```

In our case it is meaningful to sum up the values of the intersections falling into the aggregating zones since total income is a so-called spatially extensive variable (which increases with area), assuming income is evenly distributed across the smaller zones (hence the warning message above). This would be

different for spatially intensive variables such as *average* income or percentages, which do not increase as the area increases. `st_interpolate_aw()` works equally with spatially intensive variables: set the `extensive` parameter to `FALSE` and it will use an average rather than a sum function when doing the aggregation.

4.3 Spatial operations on raster data

This section builds on Section 3.3, which highlights various basic methods for manipulating raster datasets, to demonstrate more advanced and explicitly spatial raster operations, and it uses the objects `elev` and `grain` manually created in Section 3.3. For the reader's convenience, these datasets can be also found in the **spData** package.

```
elev = rast(system.file("raster/elev.tif", package = "spData"))
grain = rast(system.file("raster/grain.tif", package = "spData"))
```

4.3.1 Spatial subsetting

The previous chapter (Section 3.3) demonstrated how to retrieve values associated with specific cell IDs or row and column combinations. Raster objects can also be extracted by location (coordinates) and other spatial objects. To use coordinates for subsetting, one can 'translate' the coordinates into a cell ID with the **terra** function `cellFromXY()`. An alternative is to use `terra::extract()` (be careful, there is also a function called `extract()` in the **tidyverse**) to extract values. Both methods are demonstrated below to find the value of the cell that covers a point located at coordinates of 0.1, 0.1.

```
id = cellFromXY(elev, xy = matrix(c(0.1, 0.1), ncol = 2))
elev[id]
# the same as
terra::extract(elev, matrix(c(0.1, 0.1), ncol = 2))
```

Raster objects can also be subset with another raster object, as demonstrated in the code chunk below:

```
clip = rast(xmin = 0.9, xmax = 1.8, ymin = -0.45, ymax = 0.45,
            resolution = 0.3, vals = rep(1, 9))
elev[clip]
# we can also use extract
# terra::extract(elev, ext(clip))
```

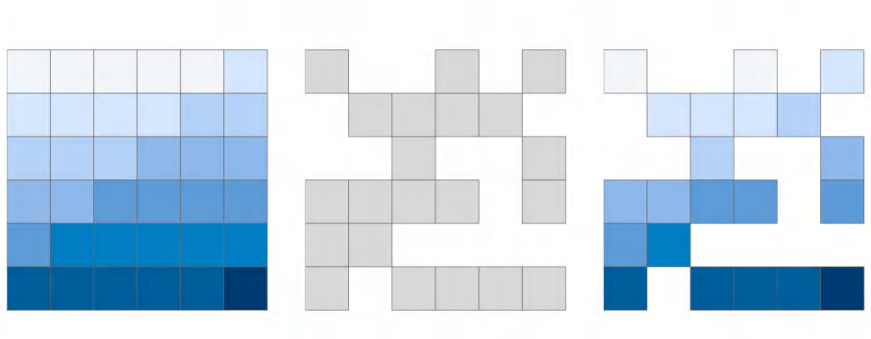

FIGURE 4.10 Original raster (left), raster mask (middle), and output of masking a raster (right).

This amounts to retrieving the values of the first raster object (in this case `elev`) that fall within the extent of a second raster (here: `clip`), as illustrated in Figure 4.10.

The example above returned the values of specific cells, but in many cases spatial outputs from subsetting operations on raster datasets are needed. This can be done by setting the `drop` argument of the `[` operator to `FALSE`. The code below returns the first two cells of `elev`, i.e., the first two cells of the top row, as a raster object (only the first 2 lines of the output is shown):

```
elev[1:2, drop = FALSE]    # spatial subsetting with cell IDs
#> class       : SpatRaster
#> dimensions  : 1, 2, 1  (nrow, ncol, nlyr)
#> ...
```

Another common use case of spatial subsetting is when a raster with `logical` (or `NA`) values is used to mask another raster with the same extent and resolution, as illustrated in Figure 4.10. In this case, the `[` and `mask()` functions can be used (results not shown).

```
# create raster mask
rmask = elev
values(rmask) = sample(c(NA, TRUE), 36, replace = TRUE)
```

In the code chunk above, we have created a mask object called `rmask` with values randomly assigned to `NA` and `TRUE`. Next, we want to keep those values of `elev` which are `TRUE` in `rmask`. In other words, we want to mask `elev` with `rmask`.

```
# spatial subsetting
elev[rmask, drop = FALSE]           # with [ operator
# we can also use mask
# mask(elev, rmask)
```

The above approach can be also used to replace some values (e.g., expected to be wrong) with NA.

```
elev[elev < 20] = NA
```

These operations are in fact Boolean local operations since we compare cell-wise two rasters. The next subsection explores these and related operations in more detail.

4.3.2 Map algebra

The term 'map algebra' was coined in the late 1970s to describe a "set of conventions, capabilities, and techniques" for the analysis of geographic raster *and* (although less prominently) vector data (Tomlin, 1994). In this context, we define map algebra more narrowly, as operations that modify or summarize raster cell values, with reference to surrounding cells, zones, or statistical functions that apply to every cell.

Map algebra operations tend to be fast, because raster datasets only implicitly store coordinates, hence the old adage "raster is faster but vector is corrector". The location of cells in raster datasets can be calculated by using its matrix position and the resolution and origin of the dataset (stored in the header). For the processing, however, the geographic position of a cell is barely relevant as long as we make sure that the cell position is still the same after the processing. Additionally, if two or more raster datasets share the same extent, projection and resolution, one could treat them as matrices for the processing.

This is the way that map algebra works with the **terra** package. First, the headers of the raster datasets are queried and (in cases where map algebra operations work on more than one dataset) checked to ensure the datasets are compatible. Second, map algebra retains the so-called one-to-one locational correspondence, meaning that cells cannot move. This differs from matrix algebra, in which values change position, for example when multiplying or dividing matrices.

Map algebra (or cartographic modeling with raster data) divides raster operations into four sub-classes (Tomlin, 1990), with each working on one or several grids simultaneously:

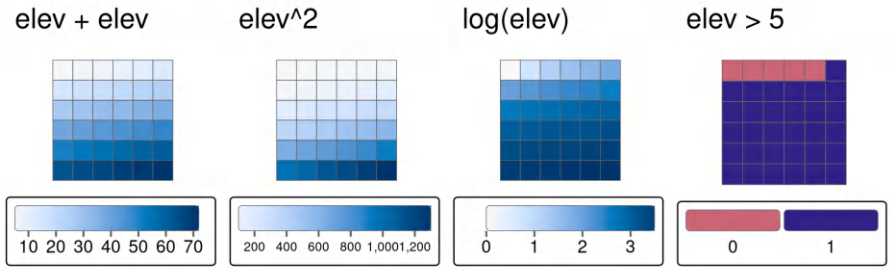

FIGURE 4.11 Examples of different local operations of the elev raster object: adding two rasters, squaring, applying logarithmic transformation, and performing a logical operation.

1. *Local* or per-cell operations
2. *Focal* or neighborhood operations. Most often the output cell value is the result of a 3 x 3 input cell block
3. *Zonal* operations are similar to focal operations, but the surrounding pixel grid on which new values are computed can have irregular sizes and shapes
4. *Global* or per-raster operations. That means the output cell derives its value potentially from one or several entire rasters

This typology classifies map algebra operations by the number of cells used for each pixel processing step and the type of the output. For the sake of completeness, we should mention that raster operations can also be classified by discipline such as terrain, hydrological analysis, or image classification. The following sections explain how each type of map algebra operations can be used, with reference to worked examples.

4.3.3 Local operations

Local operations comprise all cell-by-cell operations in one or several layers. This includes adding or subtracting values from a raster, squaring and multiplying rasters. Raster algebra also allows logical operations such as finding all raster cells that are greater than a specific value (5 in our example below). The **terra** package supports all these operations and more, as demonstrated below (Figure 4.11):

```
elev + elev
elev^2
log(elev)
elev > 5
```

Another good example of local operations is the classification of intervals of numeric values into groups such as grouping a digital elevation model into low (class 1), middle (class 2) and high elevations (class 3). Using the `classify()` command, we need first to construct a reclassification matrix, where the first column corresponds to the lower and the second column to the upper end of the class. The third column represents the new value for the specified ranges in columns one and two.

```
rcl = matrix(c(0, 12, 1, 12, 24, 2, 24, 36, 3), ncol = 3, byrow = TRUE)
rcl
#>      [,1] [,2] [,3]
#> [1,]    0   12    1
#> [2,]   12   24    2
#> [3,]   24   36    3
```

Here, we assign the raster values in the ranges 0–12, 12–24 and 24–36 are *reclassified* to take values 1, 2 and 3, respectively.

```
recl = classify(elev, rcl = rcl)
```

The `classify()` function can be also used when we want to reduce the number of classes in our categorical rasters. We will perform several additional reclassifications in Chapter 14.

Apart from applying arithmetic operators directly, one can also use the `app()`, `tapp()` and `lapp()` functions. They are more efficient, hence, they are preferable in the presence of large raster datasets. Additionally, they allow you to save an output file directly. The `app()` function applies a function to each cell of a raster and is used to summarize (e.g., calculating the sum) the values of multiple layers into one layer. `tapp()` is an extension of `app()`, allowing us to select a subset of layers (see the `index` argument) for which we want to perform a certain operation. Finally, the `lapp()` function allows us to apply a function to each cell using layers as arguments – an application of `lapp()` is presented below.

The calculation of the normalized difference vegetation index (NDVI) is a well-known local (pixel-by-pixel) raster operation. It returns a raster with values between -1 and 1; positive values indicate the presence of living plants (mostly > 0.2). NDVI is calculated from red and near-infrared (NIR) bands of remotely sensed imagery, typically from satellite systems such as Landsat or Sentinel. Vegetation absorbs light heavily in the visible light spectrum, and especially in the red channel, while reflecting NIR light. Here's the NDVI formula:

$$NDVI = \frac{\text{NIR} - \text{Red}}{\text{NIR} + \text{Red}}$$

Let's calculate NDVI for the multi-spectral satellite file of Zion National Park.

```
multi_raster_file = system.file("raster/landsat.tif", package = "spDataLarge")
multi_rast = rast(multi_raster_file)
```

Our raster object has four satellite bands from the Landsat 8 satellite: blue, green, red, and NIR. Importantly, Landsat level-2 products are stored as integers to save disk space, and thus we need to convert them to floating-point numbers before doing any calculations. For that purpose, we need to apply a scaling factor (0.0000275) and add an offset (-0.2) to the original values.[1]

```
multi_rast = (multi_rast * 0.0000275) - 0.2
```

The proper values now should be in a range between 0 and 1. This is not the case here, probably due to the presence of clouds and other atmospheric effects, which are stored as negative values. We will replace these negative values with 0 as follows.

```
multi_rast[multi_rast < 0] = 0
```

The next step should be to implement the NDVI formula into an R function:

```
ndvi_fun = function(nir, red){
  (nir - red) / (nir + red)
}
```

This function accepts two numerical arguments, `nir` and `red`, and returns a numerical vector with NDVI values. It can be used as the `fun` argument of `lapp()`. We just need to remember that our function expects two bands (not four from the original raster), and they need to be in the NIR, red order. That is why we subset the input raster with `multi_rast[[c(4, 3)]]` before doing any calculations.

```
ndvi_rast = lapp(multi_rast[[c(4, 3)]], fun = ndvi_fun)
```

The result, shown on the right panel in Figure 4.12, can be compared to the RGB image of the same area (left panel of the same figure). It allows us to see that the largest NDVI values are connected to northern areas of dense forest, while the lowest values are related to the lake in the north and snowy mountain ridges.

[1] You can read more about it at https://www.usgs.gov/faqs/how-do-i-use-a-scale-factor-landsat-level-2-science-products.

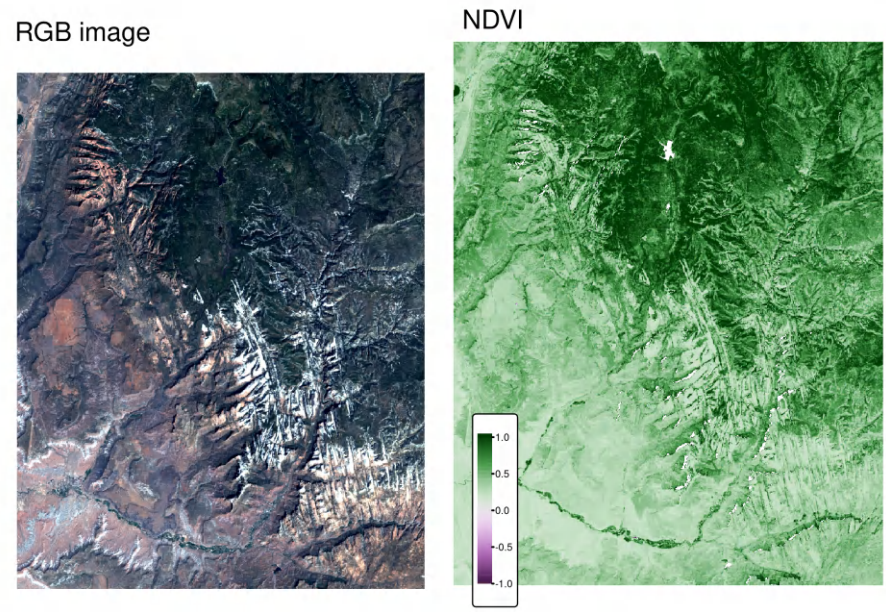

FIGURE 4.12 RGB image (left) and NDVI values (right) calculated for the example satellite file of Zion National Park

Predictive mapping is another interesting application of local raster operations. The response variable corresponds to measured or observed points in space, for example, species richness, the presence of landslides, tree disease or crop yield. Consequently, we can easily retrieve space- or airborne predictor variables from various rasters (elevation, pH, precipitation, temperature, land cover, soil class, etc.). Subsequently, we model our response as a function of our predictors using `lm()`, `glm()`, `gam()` or a machine-learning technique. Spatial predictions on raster objects can therefore be made by applying estimated coefficients to the predictor raster values, and summing the output raster values (see Chapter 15).

4.3.4 Focal operations

While local functions operate on one cell, though possibly from multiple layers, **focal** operations take into account a central (focal) cell and its neighbors. The neighborhood (also named kernel, filter or moving window) under consideration is typically of size 3-by-3 cells (that is the central cell and its eight surrounding neighbors), but it can take on any other size or (not necessarily rectangular) shape as defined by the user. A focal operation applies an aggregation function to all cells within the specified neighborhood, uses the

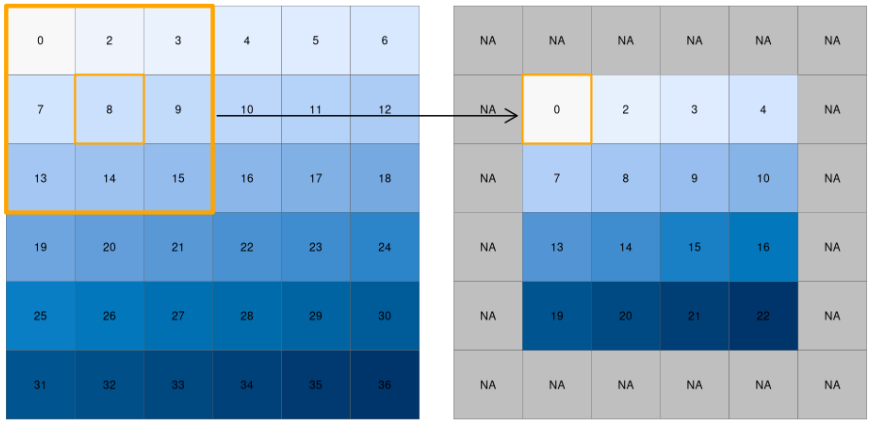

FIGURE 4.13 Input raster (left) and resulting output raster (right) due to a focal operation, finding the minimum value in 3-by-3 moving windows.

corresponding output as the new value for the central cell, and moves on to the next central cell (Figure 4.13). Other names for this operation are spatial filtering and convolution (Burrough et al., 2015).

In R, we can use the `focal()` function to perform spatial filtering. We define the shape of the moving window with a `matrix` whose values correspond to weights (see `w` parameter in the code chunk below). Secondly, the `fun` parameter lets us specify the function we wish to apply to this neighborhood. Here, we choose the minimum, but any other summary function, including `sum()`, `mean()`, or `var()` can be used.

```
r_focal = focal(elev, w = matrix(1, nrow = 3, ncol = 3), fun = min)
```

The `min()` function has an additional argument to determine whether to remove NAs in the process (`na.rm = TRUE`) or not (`na.rm = FALSE`, the default).

We can quickly check if the output meets our expectations. In our example, the minimum value has to be always the upper left corner of the moving window (remember, we have created the input raster by row-wise incrementing the cell values by one starting at the upper left corner). In this example, the weighting matrix consists only of 1s, meaning each cell has the same weight on the output, but this can be changed.

Focal functions or filters play a dominant role in image processing. Low-pass or smoothing filters use the mean function to remove extremes. In the case

of categorical data, we can replace the mean with the mode, which is the most common value. By contrast, high-pass filters accentuate features. The line detection Laplace and Sobel filters might serve as an example here. Check the `focal()` help page for how to use them in R (this will also be used in the exercises at the end of this chapter).

Terrain processing, the calculation of topographic characteristics such as slope, aspect and flow directions, relies on focal functions. `terrain()` can be used to calculate these metrics, although some terrain algorithms, including the Zevenbergen and Thorne method to compute slope, are not implemented in this **terra** function. Many other algorithms — including curvatures, contributing areas and wetness indices — are implemented in open source desktop geographic information system (GIS) software. Chapter 10 shows how to access such GIS functionality from within R.

4.3.5 Zonal operations

Just like focal operations, *zonal* operations apply an aggregation function to multiple raster cells. However, a second raster, usually with categorical values, defines the *zonal filters* (or 'zones') in the case of zonal operations, as opposed to a neighborhood window in the case of focal operations presented in the previous section. Consequently, raster cells defining the zonal filter do not necessarily have to be neighbors. Our grain-size raster is a good example, as illustrated in the right panel of Figure 3.2: different grain sizes are spread irregularly throughout the raster. Finally, the result of a zonal operation is a summary table grouped by zone which is why this operation is also known as *zonal statistics* in the GIS world. This is in contrast to focal operations which return a raster object by default.

The following code chunk uses the `zonal()` function to calculate the mean elevation associated with each grain-size class.

```
z = zonal(elev, grain, fun = "mean")
z
#>   grain elev
#> 1  clay 14.8
#> 2  silt 21.2
#> 3  sand 18.7
```

This returns the statistics for each category, here the mean altitude for each grain-size class. Note that it is also possible to get a raster with calculated statistics for each zone by setting the `as.raster` argument to `TRUE`.

4.3.6 Global operations and distances

Global operations are a special case of zonal operations with the entire raster dataset representing a single zone. The most common global operations are descriptive statistics for the entire raster dataset such as the minimum or maximum – we already discussed those in Section 3.3.2.

Aside from that, global operations are also useful for the computation of distance and weight rasters. In the first case, one can calculate the distance from each cell to a specific target cell. For example, one might want to compute the distance to the nearest coast (see also `terra::distance()`). We might also want to consider topography, for example to avoid the crossing mountain ranges on the way to the coast. This can be done by weighting distance by elevation so that each additional altitudinal meter 'prolongs' the Euclidean distance (in Exercises E8 and E9 at the end of this chapter you will do exactly that). Visibility and viewshed computations also belong to the family of global operations (in the Exercises of Chapter 10, you will compute a viewshed raster).

4.3.7 Map algebra counterparts in vector processing

Many map algebra operations have a counterpart in vector processing (Liu and Mason, 2009). Computing a distance raster (global operation) while only considering a maximum distance (logical focal operation) is the equivalent to a vector buffer operation (Section 5.2.5). Reclassifying raster data (either local or zonal function depending on the input) is equivalent to dissolving vector data (Section 4.2.5). Overlaying two rasters (local operation), where one contains `NULL` or `NA` values representing a mask, is similar to vector clipping (Section 5.2.5). Quite similar to spatial clipping is intersecting two layers (Section 4.2.1). The difference is that these two layers (vector or raster) simply share an overlapping area (see Figure 5.8 for an example). However, be careful with the wording. Sometimes the same words have slightly different meanings for raster and vector data models. While aggregating polygon geometries means dissolving boundaries, for raster data geometries it means increasing cell sizes and thereby reducing spatial resolution. Zonal operations dissolve the cells of one raster in accordance with the zones (categories) of another raster dataset using an aggregating function.

4.3.8 Merging rasters

Suppose we would like to compute the NDVI (see Section 4.3.3), and additionally we want to compute terrain attributes from elevation data for observations within a study area. Such computations rely on remotely sensed information. The corresponding imagery is often divided into scenes covering a specific spatial extent, and frequently, a study area covers more than one scene. Then, we would need to merge the scenes covered by our study area. In the easiest case, we can just merge these scenes, that is put them side by side. This is

possible, for example, with digital elevation data. In the following code chunk, we first download the Shuttle Radar Topography Mission (SRTM) elevation data for Austria and Switzerland (for the country codes, see the **geodata** function `country_codes()`). In a second step, we merge the two rasters into one.

```
aut = geodata::elevation_30s(country = "AUT", path = tempdir())
ch = geodata::elevation_30s(country = "CHE", path = tempdir())
aut_ch = merge(aut, ch)
```

terra's `merge()` command combines two images, and in case they overlap, it uses the value of the first raster.

The merging approach is of little use when the overlapping values do not correspond to each other. This is frequently the case when you want to combine spectral imagery from scenes that were taken on different dates. The `merge()` command will still work, but you will see a clear border in the resulting image. On the other hand, the `mosaic()` command lets you define a function for the overlapping area. For instance, we could compute the mean value — this might smooth the clear border in the merged result, but it will most likely not make it disappear. For a more detailed introduction to remote sensing with R, see Wegmann et al. (2016).

4.4 Exercises

```
library(sf)
library(dplyr)
library(spData)
```

E1. It was established in Section 4.2 that Canterbury was the region of New Zealand containing most of the 101 highest points in the country. How many of these high points does the Canterbury region contain?

Bonus: plot the result using the `plot()` function to show all of New Zealand, `canterbury` region highlighted in yellow, high points in Canterbury represented by red crosses (hint: `pch = 7`) and high points in other parts of New Zealand represented by blue circles. See the help page `?points` for details with an illustration of different `pch` values.

E2. Which region has the second highest number of `nz_height` points, and how many does it have?

E3. Generalizing the question to all regions: how many of New Zealand's 16 regions contain points which belong to the top 101 highest points in the country? Which regions?

- Bonus: create a table listing these regions in order of the number of points and their name.

E4. Test your knowledge of spatial predicates by finding out and plotting how US states relate to each other and other spatial objects.

The starting point of this exercise is to create an object representing Colorado state in the USA. Do this with the command `colorado = us_states[us_states$NAME == "Colorado",]` (base R) or with the `filter()` function (tidyverse) and plot the resulting object in the context of US states.

- Create a new object representing all the states that geographically intersect with Colorado and plot the result (hint: the most concise way to do this is with the subsetting method `[`).
- Create another object representing all the objects that touch (have a shared boundary with) Colorado and plot the result (hint: remember you can use the argument `op = st_intersects` and other spatial relations during spatial subsetting operations in base R).
- Bonus: create a straight line from the centroid of the District of Columbia near the East coast to the centroid of California near the West coast of the USA (hint: functions `st_centroid()`, `st_union()` and `st_cast()` described in Chapter 5 may help) and identify which states this long East-West line crosses.

E5. Use `dem = rast(system.file("raster/dem.tif", package = "spDataLarge"))`, and reclassify the elevation in three classes: low (<300), medium and high (>500). Secondly, read the NDVI raster (`ndvi = rast(system.file("raster/ndvi.tif", package = "spDataLarge"))`) and compute the mean NDVI and the mean elevation for each altitudinal class.

E6. Apply a line detection filter to `rast(system.file("ex/logo.tif", package = "terra"))`. Plot the result. Hint: Read `?terra::focal()`.

E7. Calculate the Normalized Difference Water Index (NDWI; `(green - nir)/(green + nir)`) of a Landsat image. Use the Landsat image provided by the **spDataLarge** package (`system.file("raster/landsat.tif", package = "spDataLarge")`). Also, calculate a correlation between NDVI and NDWI for this area (hint: you can use the `layerCor()` function).

E8. A StackOverflow post (stackoverflow.com/questions/35555709) shows how to compute distances to the nearest coastline using `raster::distance()`. Try to do something similar but with `terra::distance()`: retrieve a digital elevation model of Spain, and compute a raster which represents distances to the coast across the country (hint: use `geodata::elevation_30s()`). Convert the resulting distances from meters to kilometers. Note: it may be wise to increase the

cell size of the input raster to reduce compute time during this operation (`aggregate()`).

E9. Try to modify the approach used in the above exercise by weighting the distance raster with the elevation raster; every 100 altitudinal meters should increase the distance to the coast by 10 km. Next, compute and visualize the difference between the raster created using the Euclidean distance (E7) and the raster weighted by elevation.

5

Geometry operations

Prerequisites

- This chapter uses the same packages as Chapter 4 but with the addition of **spDataLarge**, which was installed in Chapter 2:

```
library(sf)
library(terra)
library(dplyr)
library(spData)
library(spDataLarge)
```

5.1 Introduction

So far the book has explained the structure of geographic datasets (Chapter 2), and how to manipulate them based on their non-geographic attributes (Chapter 3) and spatial relations (Chapter 4). This chapter focuses on manipulating the geographic elements of spatial objects, for example by creating buffers, simplifying and converting vector geometries, and aggregating and resampling raster data. After reading it — and attempting the Exercises at the end — you should understand and have control over the geometry column in `sf` objects and the extent and geographic location of pixels represented in rasters in relation to other geographic objects.

Section 5.2 covers transforming vector geometries with 'unary' and 'binary' operations. Unary operations work on a single geometry in isolation, including simplification (of lines and polygons), the creation of buffers and centroids, and shifting/scaling/rotating single geometries using 'affine transformations' (Sections 5.2.1 to 5.2.4). Binary transformations modify one geometry based on the shape of another, including clipping and geometry unions, covered in Sections 5.2.5 to 5.2.7. Type transformations (from a polygon to a line, for example) are demonstrated in Section 5.2.8.

Section 5.3 covers geometric transformations on raster objects. This involves changing the size and number of the underlying pixels, and assigning them new values. It teaches how to change the resolution (also called raster aggregation and disaggregation), the extent and the origin of a raster. These operations are especially useful if one would like to align raster datasets from diverse sources. Aligned raster objects share a one-to-one correspondence between pixels, allowing them to be processed using map algebra operations, described in Section 4.3.2.

The interaction between raster and vector objects is covered in Chapter 6. It presents how raster values can be 'masked' and 'extracted' by vector geometries. Importantly it also shows how to 'polygonize' rasters and 'rasterize' vector datasets, making the two data models more interchangeable.

5.2 Geometric operations on vector data

This section is about operations that in some way change the geometry of vector (`sf`) objects. It is more advanced than the spatial data operations presented in the previous chapter (Section 4.2), because here we drill down into the geometry: the functions discussed in this section work on objects of class `sfc` in addition to objects of class `sf`.

5.2.1 Simplification

Simplification is a process for generalization of vector objects (lines and polygons) usually for use in smaller scale maps. Another reason for simplifying objects is to reduce the amount of memory, disk space and network bandwidth they consume: it may be wise to simplify complex geometries before publishing them as interactive maps. The **sf** package provides `st_simplify()`, which uses the Douglas-Peucker algorithm to reduce the vertex count. `st_simplify()` uses the `dTolerance` to control the level of generalization in map units (see Douglas and Peucker, 1973, for details). Figure 5.1 illustrates simplification of a `LINESTRING` geometry representing the River Seine and tributaries. The simplified geometry was created by the following command:

```
seine_simp = st_simplify(seine, dTolerance = 2000)  # 2000 m
```

The resulting `seine_simp` object is a copy of the original `seine` but with fewer vertices. This is apparent, with the result being visually simpler (Figure 5.1, right) and consuming less memory than the original object, as verified below:

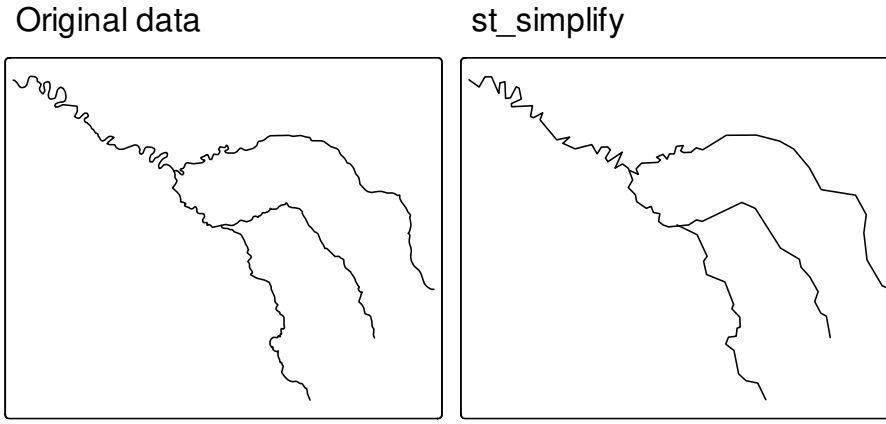

FIGURE 5.1 Comparison of the original and simplified geometry of the seine object.

```
object.size(seine)
#> 18096 bytes
object.size(seine_simp)
#> 9112 bytes
```

Simplification is also applicable for polygons. This is illustrated using `us_states`, representing the contiguous United States.

```
us_states_simp1 = st_simplify(us_states, dTolerance = 100000)  # 100 km
```

A limitation with `st_simplify()` is that it simplifies objects on a per-geometry basis. This means the 'topology' is lost, resulting in overlapping and 'holey' areal units illustrated in Figure 5.2 (right top panel). `ms_simplify()` from **rmapshaper** provides an alternative. By default it uses the Visvalingam algorithm, which overcomes some limitations of the Douglas-Peucker algorithm (Visvalingam and Whyatt, 1993). The following code chunk uses this function to simplify `us_states`. The result has only 1% of the vertices of the input (set using the argument `keep`), but its number of objects remains intact because we set `keep_shapes = TRUE`:[1]

```
# proportion of points to retain (0-1; default 0.05)
us_states_simp2 = rmapshaper::ms_simplify(us_states, keep = 0.01,
                                          keep_shapes = TRUE)
```

[1]Simplification of multipolygon objects can remove small internal polygons, even if the `keep_shapes` argument is set to TRUE. To prevent this, you need to set `explode = TRUE`. This option converts all multipolygons into separate polygons before its simplification.

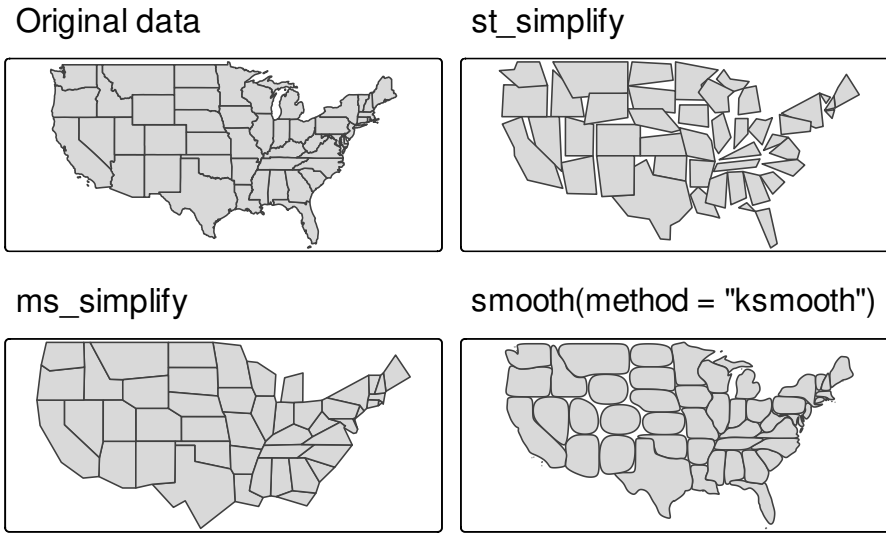

FIGURE 5.2 Polygon simplification in action, comparing the original geometry of the contiguous United States with simplified versions, generated with functions from sf (top-right), rmapshaper (bottom-left), and smoothr (bottom-right) packages.

An alternative process to simplification is smoothing the boundaries of polygon and linestring geometries, which is implemented in the **smoothr** package. Smoothing interpolates the edges of geometries and does not necessarily lead to fewer vertices, but can be especially useful when working with geometries that arise from spatially vectorizing a raster (a topic covered in Chapter 6). **smoothr** implements three techniques for smoothing: a Gaussian kernel regression, Chaikin's corner cutting algorithm, and spline interpolation, which are all described in the package vignette and website. Note that similar to `st_simplify()`, the smoothing algorithms don't preserve 'topology'. The workhorse function of **smoothr** is `smooth()`, where the `method` argument specifies what smoothing technique to use. Below is an example of using Gaussian kernel regression to smooth the borders of US states by using `method=ksmooth`. The `smoothness` argument controls the bandwidth of the Gaussian that is used to smooth the geometry and has a default value of 1.

```
us_states_simp3 = smoothr::smooth(us_states, method = "ksmooth", smoothness = 6)
```

Finally, the visual comparison of the original dataset with the simplified and smoothed versions is shown in Figure 5.2. Differences can be observed between the outputs of the Douglas-Peucker (`st_simplify`), Visvalingam (`ms_simplify`), and Gaussian kernel regression (`smooth(method=ksmooth)` algorithms.

5.2.2 Centroids

Centroid operations identify the center of geographic objects. Like statistical measures of central tendency (including mean and median definitions of 'average'), there are many ways to define the geographic center of an object. All of them create single point representations of more complex vector objects.

The most commonly used centroid operation is the *geographic centroid*. This type of centroid operation (often referred to as 'the centroid') represents the center of mass in a spatial object (think of balancing a plate on your finger). Geographic centroids have many uses, for example to create a simple point representation of complex geometries, or to estimate distances between polygons. They can be calculated with the **sf** function `st_centroid()` as demonstrated in the code below, which generates the geographic centroids of regions in New Zealand and tributaries to the River Seine, illustrated with black points in Figure 5.3.

```
nz_centroid = st_centroid(nz)
seine_centroid = st_centroid(seine)
```

Sometimes the geographic centroid falls outside the boundaries of their parent objects (think of a doughnut). In such cases *point on surface* operations can be used to guarantee the point will be in the parent object (e.g., for labeling irregular multipolygon objects such as island states), as illustrated by the red points in Figure 5.3. Notice that these red points always lie on their parent objects. They were created with `st_point_on_surface()` as follows:[2]

```
nz_pos = st_point_on_surface(nz)
seine_pos = st_point_on_surface(seine)
```

Other types of centroids exist, including the *Chebyshev center* and the *visual center*. We will not explore these here, but it is possible to calculate them using R, as we'll see in Chapter 11.

5.2.3 Buffers

Buffers are polygons representing the area within a given distance of a geometric feature: regardless of whether the input is a point, line or polygon, the output is a polygon. Unlike simplification (which is often used for visualization and reducing file size), buffering tends to be used for geographic data analysis. How many points are within a given distance of this line? Which demographic groups are within travel distance of this new shop? These kinds

[2] A description of how `st_point_on_surface()` works is provided at https://gis.stackexchange.com/a/76563/20955.

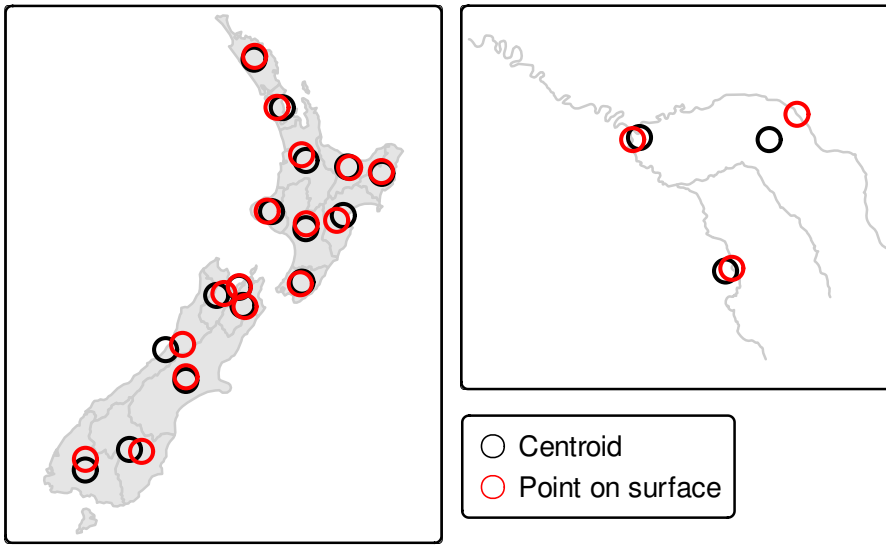

FIGURE 5.3 Centroids (black points) and 'points on surface' (red points) of New Zealand's regions (left) and the Seine (right) datasets.

of questions can be answered and visualized by creating buffers around the geographic entities of interest.

Figure 5.4 illustrates buffers of different sizes (5 and 50 km) surrounding the River Seine and tributaries. These buffers were created with commands below, which show that the command `st_buffer()` requires at least two arguments: an input geometry and a distance, provided in the units of the CRS (in this case meters).

```
seine_buff_5km = st_buffer(seine, dist = 5000)
seine_buff_50km = st_buffer(seine, dist = 50000)
```

The `st_buffer()` has a few additional arguments. The most important ones are:

- `nQuadSegs` (when the GEOS engine is used), which means 'number of segments per quadrant' and is set by default to 30 (meaning circles created by buffers are composed of $4 \times 30 = 120$ lines). Unusual cases where it may be useful include when the memory consumed by the output of a buffer operation is a major concern (in which case it should be reduced) or when very high precision is needed (in which case it should be increased)

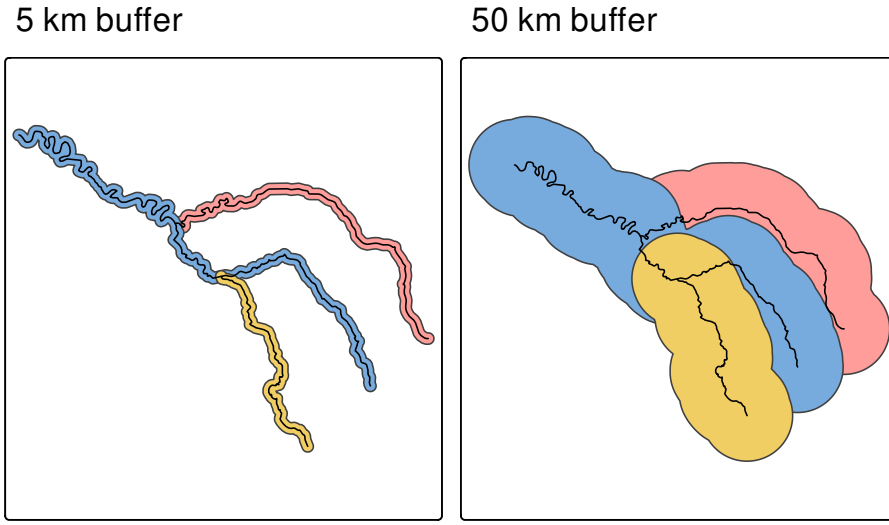

FIGURE 5.4 Buffers around the Seine dataset of 5 km (left) and 50 km (right). Note the colors, which reflect the fact that one buffer is created per geometry feature.

- `max_cells` (when the S2 engine is used), the larger the value, the more smooth the buffer will be, but the calculations will take longer
- `endCapStyle` and `joinStyle` (when the GEOS engine is used), which control the appearance of the buffer's edges
- `singleSide` (when the GEOS engine is used), which controls whether the buffer is created on one or both sides of the input geometry

5.2.4 Affine transformations

Affine transformation is any transformation that preserves lines and parallelism. However, angles or length are not necessarily preserved. Affine transformations include, among others, shifting (translation), scaling and rotation. Additionally, it is possible to use any combination of these. Affine transformations are an essential part of geocomputation. For example, shifting is needed for labels placement, scaling is used in non-contiguous area cartograms (see Section 9.6), and many affine transformations are applied when reprojecting or improving the geometry that was created based on a distorted or wrongly projected map. The **sf** package implements affine transformation for objects of classes `sfg` and `sfc`.

```
nz_sfc = st_geometry(nz)
```

Shifting moves every point by the same distance in map units. It could be done by adding a numerical vector to a vector object. For example, the code below shifts all y-coordinates by 100,000 meters to the north, but leaves the x-coordinates untouched (Figure 5.5, left panel).

```
nz_shift = nz_sfc + c(0, 100000)
```

Scaling enlarges or shrinks objects by a factor. It can be applied either globally or locally. Global scaling increases or decreases all coordinates values in relation to the origin coordinates, while keeping all geometries topological relations intact. It can be done by subtraction or multiplication of a `sfg` or `sfc` object.

Local scaling treats geometries independently and requires points around which geometries are going to be scaled, e.g., centroids. In the example below, each geometry is shrunk by a factor of two around the centroids (Figure 5.5, middle panel). To achieve that, each object is firstly shifted in a way that its center has coordinates of `0, 0` (`(nz_sfc - nz_centroid_sfc)`). Next, the sizes of the geometries are reduced by half (`* 0.5`). Finally, each object's centroid is moved back to the input data coordinates (`+ nz_centroid_sfc`).

```
nz_centroid_sfc = st_centroid(nz_sfc)
nz_scale = (nz_sfc - nz_centroid_sfc) * 0.5 + nz_centroid_sfc
```

Rotation of two-dimensional coordinates requires a rotation matrix:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

It rotates points in a clockwise direction. The rotation matrix can be implemented in R as:

```
rotation = function(a){
  r = a * pi / 180 #degrees to radians
  matrix(c(cos(r), sin(r), -sin(r), cos(r)), nrow = 2, ncol = 2)
}
```

The `rotation` function accepts one argument `a` — a rotation angle in degrees. Rotation could be done around selected points, such as centroids (Figure 5.5, right panel). See `vignette("sf3")` for more examples.

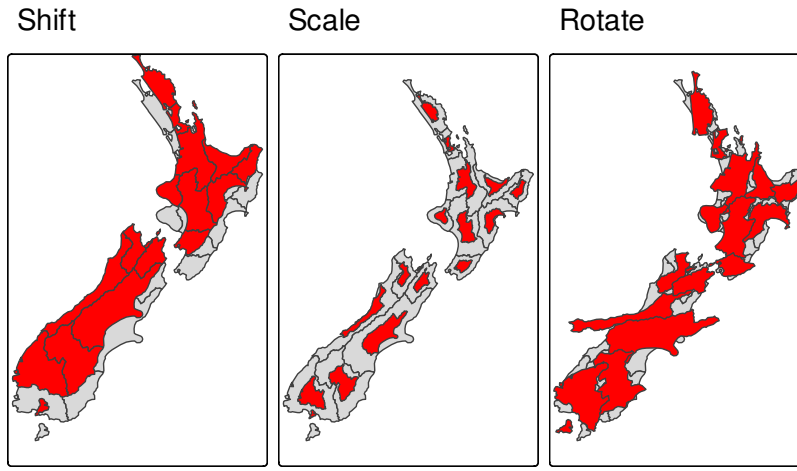

FIGURE 5.5 Affine transformations: shift, scale and rotate.

```
nz_rotate = (nz_sfc - nz_centroid_sfc) * rotation(30) + nz_centroid_sfc
```

Finally, the newly created geometries can replace the old ones with the `st_set_geometry()` function:

```
nz_scale_sf = st_set_geometry(nz, nz_scale)
```

5.2.5 Clipping

Spatial clipping is a form of spatial subsetting that involves changes to the `geometry` columns of at least some of the affected features.

Clipping can only apply to features more complex than points: lines, polygons and their 'multi' equivalents. To illustrate the concept, we will start with a simple example: two overlapping circles with a center point one unit away from each other and a radius of one (Figure 5.6).

```
b = st_sfc(st_point(c(0, 1)), st_point(c(1, 1))) # create 2 points
b = st_buffer(b, dist = 1) # convert points to circles
plot(b, border = "gray")
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y"), cex = 3) # add text
```

Imagine you want to select not one circle or the other, but the space covered by both `x` *and* `y`. This can be done using the function `st_intersection()`, illustrated using objects named `x` and `y` which represent the left- and right-hand circles (Figure 5.7).

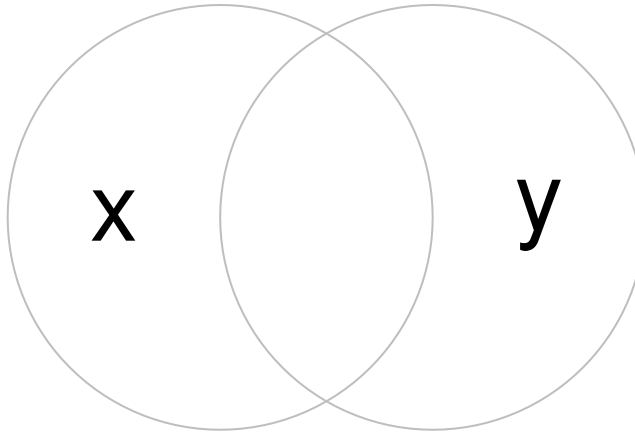

FIGURE 5.6 Overlapping circles.

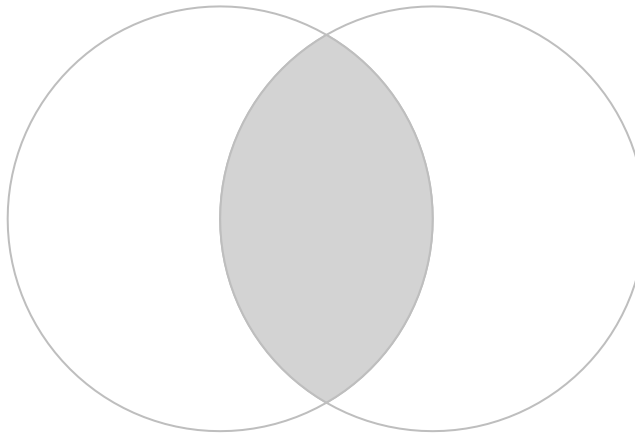

FIGURE 5.7 Overlapping circles with a gray color indicating intersection between them.

```
x = b[1]
y = b[2]
x_and_y = st_intersection(x, y)
plot(b, border = "gray")
plot(x_and_y, col = "lightgray", border = "gray", add = TRUE) # intersecting area
```

The subsequent code chunk demonstrates how this works for all combinations of the 'Venn' diagram representing `x` and `y`, inspired by Figure 5.1 of the book *R for Data Science* (Grolemund and Wickham, 2016).

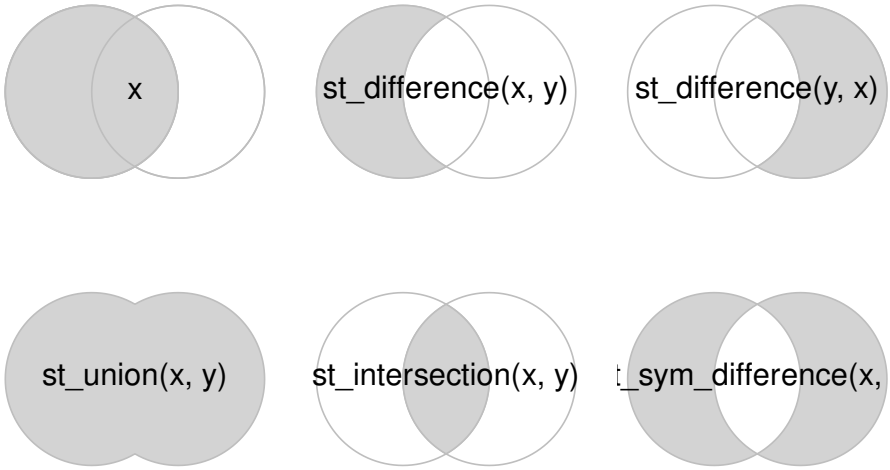

FIGURE 5.8 Spatial equivalents of logical operators.

5.2.6 Subsetting and clipping

Clipping objects can change their geometry, but it can also subset objects, returning only features that intersect (or partly intersect) with a clipping/subsetting object. To illustrate this point, we will subset points that cover the bounding box of the circles `x` and `y` in Figure 5.8. Some points will be inside just one circle, some will be inside both and some will be inside neither. `st_sample()` is used below to generate a *simple random* distribution of points within the extent of circles `x` and `y`, resulting in output illustrated in Figure 5.9, raising the question: how to subset the points to only return the point that intersects with *both* `x` and `y`?

```
bb = st_bbox(st_union(x, y))
box = st_as_sfc(bb)
set.seed(2024)
p = st_sample(x = box, size = 10)
p_xy1 = p[x_and_y]
plot(box, border = "gray", lty = 2)
plot(x, add = TRUE, border = "gray")
plot(y, add = TRUE, border = "gray")
plot(p, add = TRUE, cex = 3.5)
plot(p_xy1, cex = 5, col = "red", add = TRUE)
text(x = c(-0.5, 1.5), y = 1, labels = c("x", "y"), cex = 3)
```

```
bb = st_bbox(st_union(x, y))
box = st_as_sfc(bb)
```

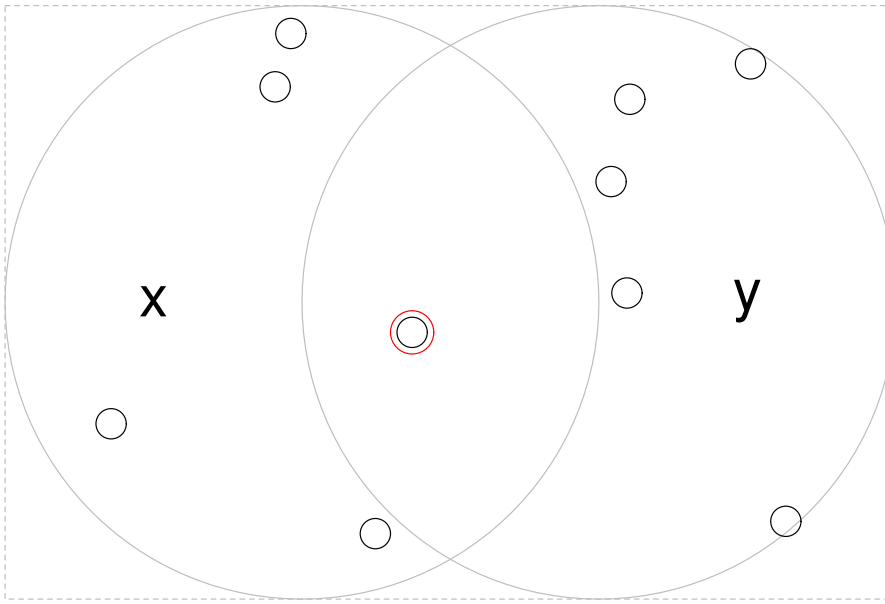

FIGURE 5.9 Randomly distributed points within the bounding box enclosing circles x and y. The point that intersects with both objects x and y is highlighted.

```
set.seed(2024)
p = st_sample(x = box, size = 10)
x_and_y = st_intersection(x, y)
```

The code chunk below demonstrates three ways to achieve the same result. We can use the intersection of `x` and `y` (represented by `x_and_y` in the previous code chunk) as a subsetting object directly, as shown in the first line in the code chunk below. We can also find the *intersection* between the input points represented by `p` and the subsetting/clipping object `x_and_y`, as demonstrated in the second line in the code chunk below. This second approach will return features that partly intersect with `x_and_y` but with modified geometries for spatially extensive features that cross the border of the subsetting object. The third approach is to create a subsetting object using the binary spatial predicate `st_intersects()`, introduced in the previous chapter. The results are identical (except superficial differences in attribute names), but the implementation differs substantially:

```
# way #1
p_xy1 = p[x_and_y]
```

```
# way #2
p_xy2 = st_intersection(p, x_and_y)
# way #3
sel_p_xy = st_intersects(p, x, sparse = FALSE)[, 1] &
  st_intersects(p, y, sparse = FALSE)[, 1]
p_xy3 = p[sel_p_xy]
```

Although the example above is rather contrived and provided for educational rather than applied purposes, and we encourage the reader to reproduce the results to deepen understanding for handling geographic vector objects in R, it raises an important question: which implementation to use? Generally, more concise implementations should be favored, meaning the first approach above. We will return to the question of choosing between different implementations of the same technique or algorithm in Chapter 11.

5.2.7 Geometry unions

As we saw in Section 3.2.3, spatial aggregation can silently dissolve the geometries of touching polygons in the same group. This is demonstrated in the code chunk below in which 48 US states and the District of Columbia (`us_states`) are aggregated into four regions using base and **dplyr** functions (see results in Figure 5.10):

```
regions = aggregate(x = us_states[, "total_pop_15"], by = list(us_states$REGION),
                    FUN = sum, na.rm = TRUE)
regions2 = us_states |>
  group_by(REGION) |>
  summarize(pop = sum(total_pop_15, na.rm = TRUE))
```

What is going on in terms of the geometries? Behind the scenes, both `aggregate()` and `summarize()` combine the geometries and dissolve the boundaries between them using `st_union()`. This is demonstrated in the code chunk below which creates a united western US:

```
us_west = us_states[us_states$REGION == "West", ]
us_west_union = st_union(us_west)
```

The function can take two geometries and unite them, as demonstrated in the code chunk below which creates a united western block incorporating Texas (challenge: reproduce and plot the result):

```
texas = us_states[us_states$NAME == "Texas", ]
texas_union = st_union(us_west_union, texas)
```

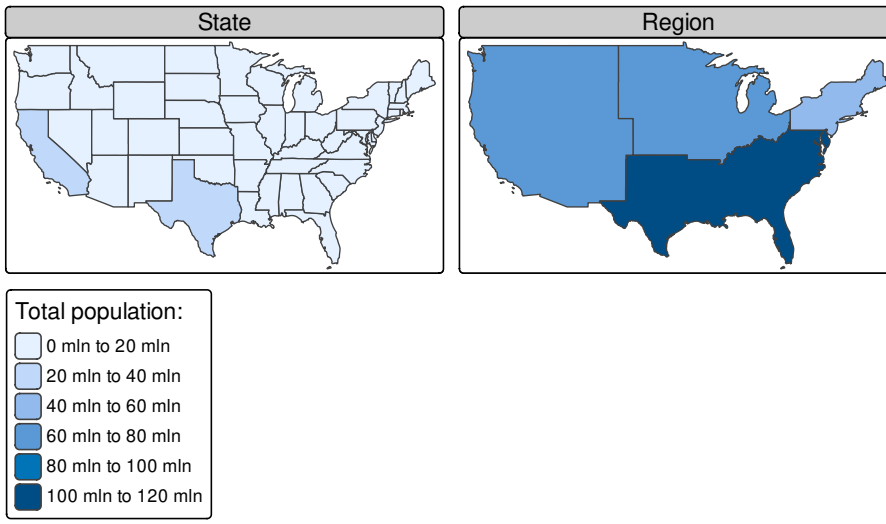

FIGURE 5.10 Spatial aggregation on contiguous polygons, illustrated by aggregating the population of US states into regions, with population represented by color. Note the operation automatically dissolves boundaries between states.

5.2.8 Type transformations

Geometry casting is a powerful operation that enables transformation of the geometry type. It is implemented in the `st_cast()` function from the **sf** package. Importantly, `st_cast()` behaves differently on single simple feature geometry (`sfg`) objects, simple feature geometry column (`sfc`) and simple features objects.

Let's create a multipoint to illustrate how geometry casting works on simple feature geometry (`sfg`) objects:

```
multipoint = st_multipoint(matrix(c(1, 3, 5, 1, 3, 1), ncol = 2))
```

In this case, `st_cast()` can be useful to transform the new object into a linestring or a polygon (Figure 5.11).

```
linestring = st_cast(multipoint, "LINESTRING")
polyg = st_cast(multipoint, "POLYGON")
```

Conversion from multipoint to linestring is a common operation that creates a line object from ordered point observations, such as GPS measurements or geotagged media. This, in turn, allows us to perform spatial operations such as the calculation of the length of the path traveled. Conversion from multipoint

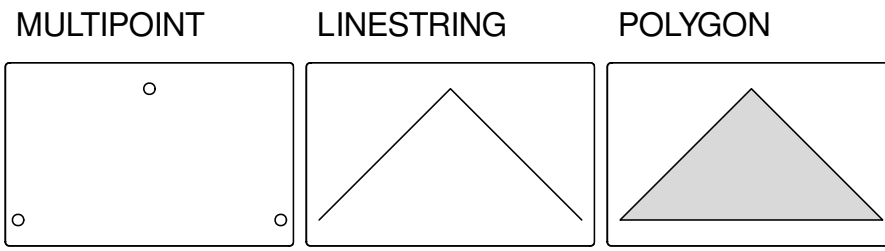

FIGURE 5.11 Examples of a linestring and a polygon casted from a multipoint geometry.

or linestring to polygon is often used to calculate an area, for example from the set of GPS measurements taken around a lake or from the corners of a building lot.

The transformation process can be also reversed using `st_cast()`:

```
multipoint_2 = st_cast(linestring, "MULTIPOINT")
multipoint_3 = st_cast(polyg, "MULTIPOINT")
all.equal(multipoint, multipoint_2)
#> [1] TRUE
all.equal(multipoint, multipoint_3)
#> [1] TRUE
```

For single simple feature geometries (`sfg`), `st_cast()` also provides geometry casting from non-multi-types to multi-types (e.g., `POINT` to `MULTIPOINT`) and from multi-types to non-multi-types. However, when casting from multi-types to non-multi-types, only the first element of the old object would remain in the output object.

Geometry casting of simple features geometry column (`sfc`) and simple features objects works the same as for `sfg` in most of the cases. One important difference is the conversion between multi-types to non-multi-types. As a result of this process, multi-objects of `sfc` or `sf` are split into many non-multi-objects.

Let's say we have the following `sf` objects:

- `POI` - POINT type (with one point by definition)
- `MPOI` - MULTIPOINT type with four points
- `LIN` - LINESTRING type with one linestring containing five points
- `MLIN` - MULTILINESTRING type with two linestrings (one with five points and one with two points)
- `POL` - POLYGON type with one polygon (created using five points)

TABLE 5.1 Geometry casting on simple feature geometries (see Section 2.1) with input type by row and output type by column.

	POI	MPOI	LIN	MLIN	POL	MPOL	GC
POI(1)	1	1	1	NA	NA	NA	NA
MPOI(1)	4	1	1	1	1	NA	NA
LIN(1)	5	1	1	1	1	NA	NA
MLIN(1)	7	2	2	1	NA	NA	NA
POL(1)	5	1	1	1	1	1	NA
MPOL(1)	10	1	NA	1	2	1	1
GC(1)	9	1	NA	NA	NA	NA	1

Note: Values in parentheses represent the number of features; NA means the operation is not available

- `MPOL` - MULTIPOLYGON type consisting of two polygons (both consisting of five points)
- `GC` - GEOMETRYCOLLECTION type with two geometries, a MULTIPOINT (four points) and a LINESTRING (five points)

Table 5.1 shows possible geometry type transformations on the simple feature objects listed above. Single simple feature geometries (represented by the first column in the table) can be transformed into multiple geometry types, represented by the columns in Table 5.1. Some transformations are not possible: you cannot convert a single point into a multilinestring or a polygon, for example, explaining why the cells `[1, 4:5]` in the table contain NA. Some transformations split single features input into multiple sub-features, 'expanding' `sf` objects (adding new rows with duplicate attribute values). When a multipoint geometry consisting of five pairs of coordinates is tranformed into a 'POINT' geometry, for example, the output will contain five features.

Let's try to apply geometry type transformations on a new object, `multilinestring_sf`, as an example (on the left in Figure 5.12):

```
multilinestring_list = list(matrix(c(1, 4, 5, 3), ncol = 2),
                            matrix(c(4, 4, 4, 1), ncol = 2),
                            matrix(c(2, 4, 2, 2), ncol = 2))
multilinestring = st_multilinestring(multilinestring_list)
multilinestring_sf = st_sf(geom = st_sfc(multilinestring))
multilinestring_sf
#> Simple feature collection with 1 feature and 0 fields
#> Geometry type: MULTILINESTRING
#> Dimension:     XY
#> Bounding box:  xmin: 1 ymin: 1 xmax: 4 ymax: 5
#> CRS:           NA
```

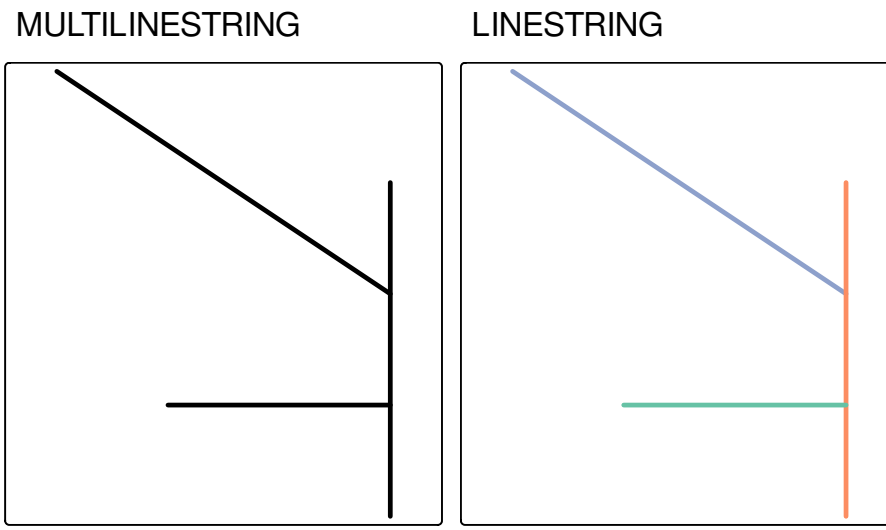

FIGURE 5.12 Examples of type casting between multilinestring (left) and linestring (right).

```
#>                              geom
#> 1 MULTILINESTRING ((1 5, 4 3)...
```

You can imagine it as a road or river network. The new object has only one row that defines all the lines. This restricts the number of operations that can be done, for example it prevents adding names to each line segment or calculating lengths of single lines. The `st_cast()` function can be used in this situation, as it separates one mutlilinestring into three linestrings.

```
linestring_sf2 = st_cast(multilinestring_sf, "LINESTRING")
linestring_sf2
#> Simple feature collection with 3 features and 0 fields
#> Geometry type: LINESTRING
#> Dimension:     XY
#> Bounding box:  xmin: 1 ymin: 1 xmax: 4 ymax: 5
#> CRS:           NA
#>                       geom
#> 1 LINESTRING (1 5, 4 3)
#> 2 LINESTRING (4 4, 4 1)
#> 3 LINESTRING (2 2, 4 2)
```

The newly created object allows for attributes creation (see more in Section 3.2.5) and length measurements:

```
linestring_sf2$name = c("Riddle Rd", "Marshall Ave", "Foulke St")
linestring_sf2$length = st_length(linestring_sf2)
linestring_sf2
#> Simple feature collection with 3 features and 2 fields
#> Geometry type: LINESTRING
#> Dimension:     XY
#> Bounding box:  xmin: 1 ymin: 1 xmax: 4 ymax: 5
#> CRS:           NA
#>                    geom         name length
#> 1 LINESTRING (1 5, 4 3)    Riddle Rd   3.61
#> 2 LINESTRING (4 4, 4 1) Marshall Ave   3.00
#> 3 LINESTRING (2 2, 4 2)    Foulke St   2.00
```

5.3 Geometric operations on raster data

Geometric raster operations include the shifting, flipping, mirroring, scaling, rotation or warping of images. These operations are necessary for a variety of applications including georeferencing, used to allow images to be overlaid on an accurate map with a known CRS (Liu and Mason, 2009). A variety of georeferencing techniques exist, including:

- Georectification based on known ground control points
- Orthorectification, which also accounts for local topography
- Image registration is used to combine images of the same thing, but shot from different sensors by aligning one image with another (in terms of coordinate system and resolution)

R is rather unsuitable for the first two points since these often require manual intervention which is why they are usually done with the help of dedicated GIS software (see also Chapter 10). On the other hand, aligning several images is possible in R and this section shows among others how to do so. This often includes changing the extent, the resolution and the origin of an image. A matching projection is of course also required but is already covered in Section 7.8.

In any case, there are other reasons to perform a geometric operation on a single raster image. For instance, in Chapter 14 we define metropolitan areas in Germany as 20 km^2 pixels with more than 500,000 inhabitants. The original inhabitant raster, however, has a resolution of 1 km^2 which is why we will decrease (aggregate) the resolution by a factor of 20 (see Section 14.5). Another reason for aggregating a raster is simply to decrease run-time or save disk space. Of course, this approach is only recommended if the task at hand allows a coarser resolution of raster data.

5.3.1 Geometric intersections

In Section 4.3.1 we have shown how to extract values from a raster overlaid by other spatial objects. To retrieve a spatial output, we can use almost the same subsetting syntax. The only difference is that we have to make clear that we would like to keep the matrix structure by setting the `drop` argument to `FALSE`. This will return a raster object containing the cells whose midpoints overlap with `clip`.

```
elev = rast(system.file("raster/elev.tif", package = "spData"))
clip = rast(xmin = 0.9, xmax = 1.8, ymin = -0.45, ymax = 0.45,
            resolution = 0.3, vals = rep(1, 9))
elev[clip, drop = FALSE]
#> class       : SpatRaster
#> dimensions  : 2, 1, 1  (nrow, ncol, nlyr)
#> resolution  : 0.5, 0.5  (x, y)
#> extent      : 1, 1.5, -0.5, 0.5  (xmin, xmax, ymin, ymax)
#> coord. ref. : lon/lat WGS 84 (EPSG:4326)
#> source(s)   : memory
#> varname     : elev
#> name        : elev
#> min value   :   18
#> max value   :   24
```

For the same operation, we can also use the `intersect()` and `crop()` command.

5.3.2 Extent and origin

When merging or performing map algebra on rasters, their resolution, projection, origin and/or extent have to match. Otherwise, how should we add the values of one raster with a resolution of 0.2 decimal degrees to a second raster with a resolution of 1 decimal degree? The same problem arises when we would like to merge satellite imagery from different sensors with different projections and resolutions. We can deal with such mismatches by aligning the rasters.

In the simplest case, two images only differ with regard to their extent. The following code adds one row and two columns to each side of the raster while setting all new values to `NA` (Figure 5.13).

```
elev = rast(system.file("raster/elev.tif", package = "spData"))
elev_2 = extend(elev, c(1, 2))
```

Performing an algebraic operation on two objects with differing extents in R, the **terra** package returns an error.

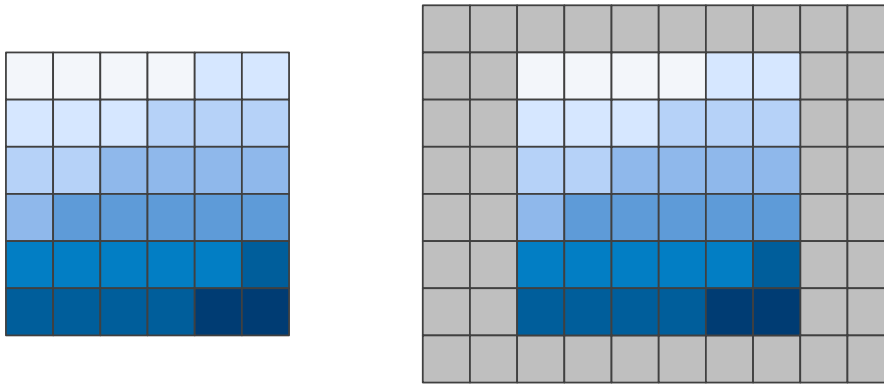

FIGURE 5.13 Original raster (left) and the same raster (right) extended by one row on the top and bottom and two columns on the left and right.

```
elev_3 = elev + elev_2
#> Error: [+] extents do not match
```

However, we can align the extent of two rasters with `extend()`. Instead of telling the function how many rows or columns should be added (as done before), we allow it to figure it out by using another raster object. Here, we extend the `elev` object to the extent of `elev_2`. The values of the newly added rows and columns are set to `NA`.

```
elev_4 = extend(elev, elev_2)
```

The origin of a raster is the cell corner closest to the coordinates (0, 0). The `origin()` function returns the coordinates of the origin. In the example below a cell corner exists with coordinates (0, 0), but that is not necessarily the case.

```
origin(elev_4)
#> [1] 0 0
```

If two rasters have different origins, their cells do not overlap completely which would make map algebra impossible. To change the origin, use `origin()`.[3] Figure 5.14 reveals the effect of changing the origin in this way.

```
# change the origin
origin(elev_4) = c(0.25, 0.25)
```

[3]If the origins of two raster datasets are just marginally apart, it sometimes is sufficient to simply increase the `tolerance` argument of `terra::terraOptions()`.

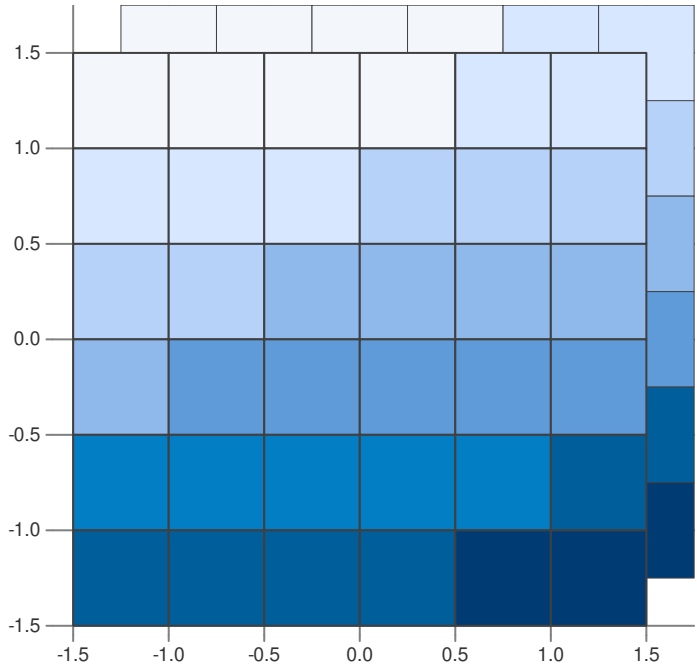

FIGURE 5.14 Rasters with identical values but different origins.

Note that changing the resolution (next section) frequently also changes the origin.

5.3.3 Aggregation and disaggregation

Raster datasets can also differ with regard to their resolution. To match resolutions, one can either decrease (`aggregate()`) or increase (`disagg()`) the resolution of one raster.[4]

As an example, we here change the spatial resolution of `dem` (found in the **spDataLarge** package) by a factor of 5 (Figure 5.15). Additionally, the output cell value is going to correspond to the mean of the input cells (note that one could use other functions as well, such as `median()`, `sum()`, etc.):

```
dem = rast(system.file("raster/dem.tif", package = "spDataLarge"))
dem_agg = aggregate(dem, fact = 5, fun = mean)
```

[4]Here we refer to spatial resolution. In remote sensing the spectral (spectral bands), temporal (observations through time of the same area) and radiometric (color depth) resolution are also important. Check out the `tapp()` example in the documentation for getting an idea on how to do temporal raster aggregation.

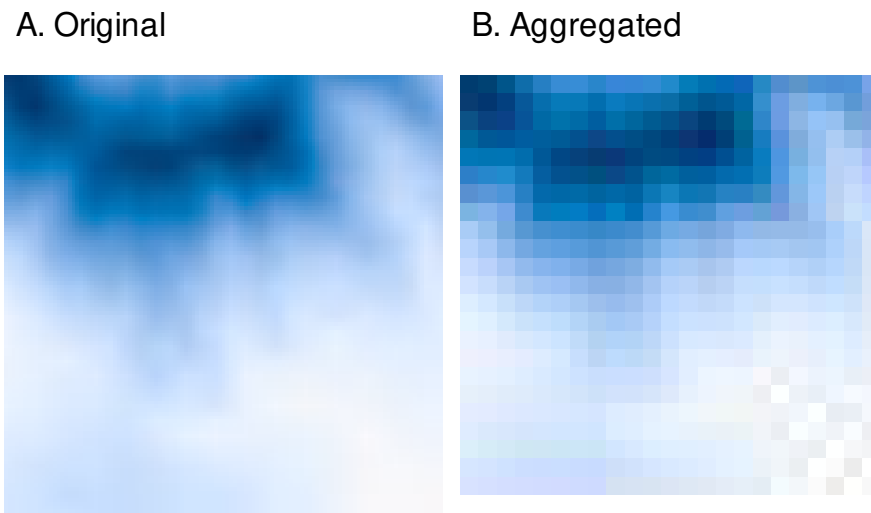

FIGURE 5.15 Original raster (left) and aggregated raster (right).

TABLE 5.2 Properties of the original and aggregated raster.

Object	Resolution	Dimensions	Extent
dem	(30.85, 30.85)	117 * 117	794599.1, 798208.6, 8931775, 8935384
dem_agg	(154.25, 154.25)	24 * 24	794599.1, 798301.1, 8931682, 8935384

Table 5.2 compares the properties of the original and aggregated raster. Notice that "decreasing" the resolution with `aggregate()` increases the resolution from $(30.85, 30.85)$ to $(154.25, 154.25)$. This is done by decreasing the number of rows (`nrow`) and columns (`ncol`) (see Section 2.3). The extent was slightly adjusted to accommodate the new grid size.

The `disagg()` function increases the resolution of raster objects. It comes with two methods on how to compute the values of the newly created cells: the default method (`method = "near"`) simply gives all output cells the value of the input cell, and hence duplicates values, which translates into a 'blocky' output. The `bilinear` method uses the four nearest pixel centers of the input image (orange colored points in Figure 5.16) to compute an average weighted by distance (arrows in Figure 5.16). The value of the output cell is represented by a square in the upper left corner in Figure 5.16.

```
dem_disagg = disagg(dem_agg, fact = 5, method = "bilinear")
identical(dem, dem_disagg)
#> [1] FALSE
```

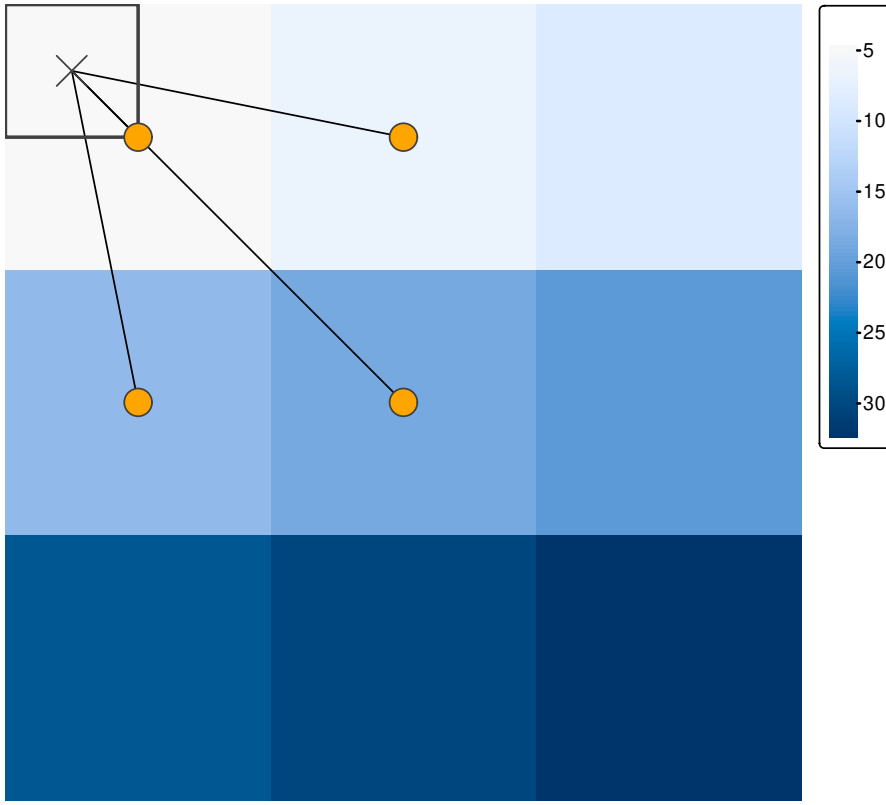

FIGURE 5.16 The distance-weighted average of the four closest input cells determine the output when using the bilinear method for disaggregation.

Comparing the values of `dem` and `dem_disagg` tells us that they are not identical (you can also use `compareGeom()` or `all.equal()`). However, this was hardly to be expected, since disaggregating is a simple interpolation technique. It is important to keep in mind that disaggregating results in a finer resolution; the corresponding values, however, are only as accurate as their lower resolution source.

5.3.4 Resampling

The above methods of aggregation and disaggregation are only suitable when we want to change the resolution of our raster by the aggregation/disaggregation factor. However, what to do when we have two or more rasters with different resolutions and origins? This is the role of resampling – a process of computing values for new pixel locations. In short, this process

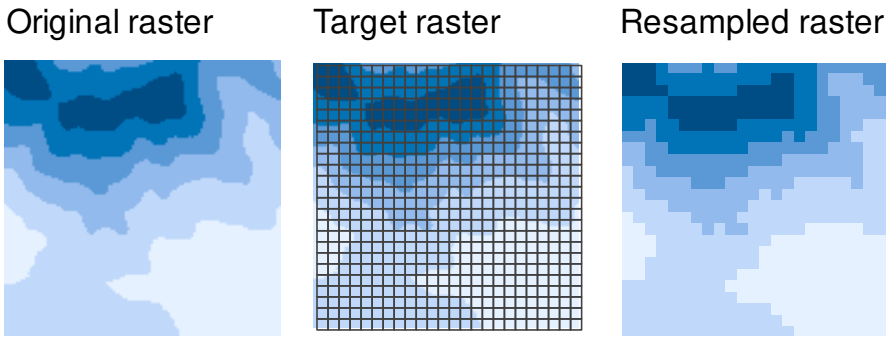

FIGURE 5.17 Resampling of an original (input) raster into a target raster with custom resolution and origin.

takes the values of our original raster and recalculates new values for a target raster with custom resolution and origin (Figure 5.17).

There are several methods for estimating values for a raster with different resolutions/origins, as shown in Figure 5.18. The main resampling methods include:

- Nearest neighbor: assigns the value of the nearest cell of the original raster to the cell of the target one. This is a fast simple technique that is usually suitable for resampling categorical rasters.
- Bilinear interpolation: assigns a weighted average of the four nearest cells from the original raster to the cell of the target one (Figure 5.16). This is the fastest method that is appropriate for continuous rasters.
- Cubic interpolation: uses values of the 16 nearest cells of the original raster to determine the output cell value, applying third-order polynomial functions. Used for continuous rasters and results in a smoother surface compared to bilinear interpolation but is computationally more demanding.
- Cubic spline interpolation: also uses values of the 16 nearest cells of the original raster to determine the output cell value, but applies cubic splines (piece-wise third-order polynomial functions). Used for continuous rasters.
- Lanczos windowed sinc resampling: uses values of the 36 nearest cells of the original raster to determine the output cell value. Used for continuous rasters.[5]

The above explanation highlights that only *nearest neighbor* resampling is suitable for categorical rasters, while all methods can be used (with different outcomes) for continuous rasters. Please note also that the methods gain both in complexity and processing time from top to bottom. Moreover, resampling can be done using statistics (e.g., minimum or mode) of all contributing cells.

[5]More detailed explanation of this method can be found at https://gis.stackexchange.com/a/14361/20955.

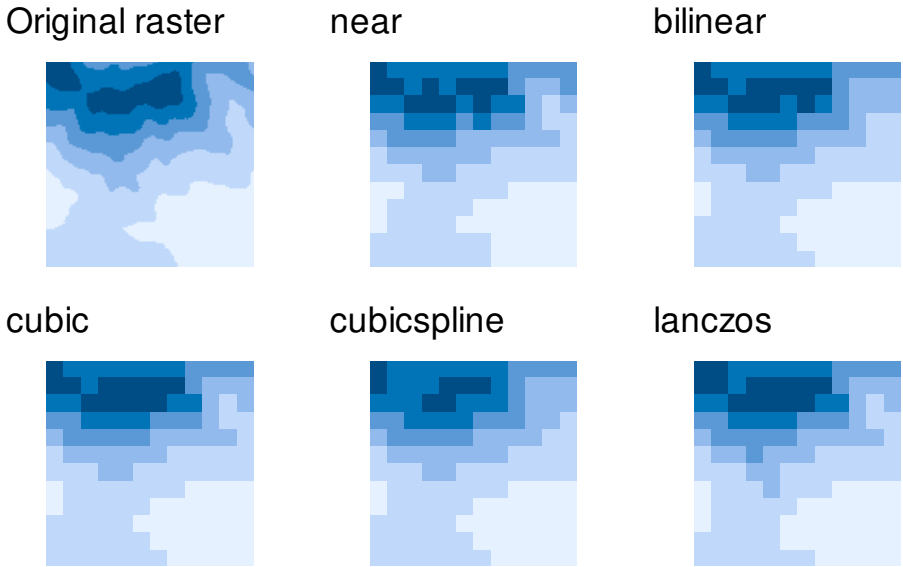

FIGURE 5.18 Visual comparison of the original raster and five different resampling methods.

To apply resampling, the **terra** package provides a `resample()` function. It accepts an input raster (`x`), a raster with target spatial properties (`y`), and a resampling method (`method`).

We need a raster with target spatial properties to see how the `resample()` function works. For this example, we create `target_rast`, but you would often use an already existing raster object.

```
target_rast = rast(xmin = 794650, xmax = 798250,
                   ymin = 8931750, ymax = 8935350,
                   resolution = 300, crs = "EPSG:32717")
```

Next, we need to provide our two raster objects as the first two arguments and one of the resampling methods described above.

```
dem_resampl = resample(dem, y = target_rast, method = "bilinear")
```

Figure 5.18 shows a comparison of different resampling methods on the `dem` object.

The `resample()` function also has some additional resampling methods, including `sum`, `min`, `q1`, `med`, `q3`, `max`, `average`, `mode`, and `rms`. All of them calculate a given statistic based on the values of all non-NA contributing grid cells. For

example, `sum` is useful when each raster cell represents a spatially extensive variable (e.g., number of people). As an effect of using `sum`, the resampled raster should have the same total number of people as the original one.

As you will see in Section 7.8, raster reprojection is a special case of resampling when our target raster has a different CRS than the original raster.

Most geometry operations in **terra** are user-friendly, rather fast, and work on large raster objects. However, there could be some cases when **terra** is not the most performant either for extensive rasters or many raster files, and some alternatives should be considered.

The most established alternatives come with the GDAL library. It contains several utility functions, including:

- `gdalinfo` - lists various information about a raster file, including its resolution, CRS, bounding box, and more
- `gdal_translate` - converts raster data between different file formats
- `gdal_rasterize` - converts vector data into raster files
- `gdalwarp` - allows for raster mosaicing, resampling, cropping, and reprojecting

All of the above functions are written in C++, but can be called in R using `sf::gdal_utils()`, the **gdalUtilities** package, or via system commands (see Section 10.6). Importantly, all of these functions expect a raster file path as an input and often return their output as a raster file (for example, `gdalUtilities::gdal_translate("my_file.tif", "new_file.tif", t_srs = "EPSG:4326")`). This is very different from the usual **terra** approach, which expects `SpatRaster` objects as inputs.

5.4 Exercises

E1. Generate and plot simplified versions of the `nz` dataset. Experiment with different values of `keep` (ranging from 0.5 to 0.00005) for `ms_simplify()` and `dTolerance` (from 100 to 100,000) `st_simplify()`.

- At what value does the form of the result start to break down for each method, making New Zealand unrecognizable?
- Advanced: What is different about the geometry type of the results from `st_simplify()` compared with the geometry type of `ms_simplify()`? What problems does this create and how can this be resolved?

E2. In the first exercise in Chapter Spatial Data Operations it was established that Canterbury region had 70 of the 101 highest points in New Zealand. Using `st_buffer()`, how many points in `nz_height` are within 100 km of Canterbury?

E3. Find the geographic centroid of New Zealand. How far is it from the geographic centroid of Canterbury?

E4. Most world maps have a north-up orientation. A world map with a south-up orientation could be created by a reflection (one of the affine transformations not mentioned in this chapter) of the `world` object's geometry. Write code to do so. Hint: you can to use the `rotation()` function from this chapter for this transformation. Bonus: create an upside-down map of your country.

E5. Run the code in Section 5.2.6. With reference to the objects created in that section, subset the point in `p` that is contained within `x` *and* `y`.

- Using base subsetting operators.
- Using an intermediary object created with `st_intersection()`.

E6. Calculate the length of the boundary lines of US states in meters. Which state has the longest border and which has the shortest? Hint: The `st_length` function computes the length of a `LINESTRING` or `MULTILINESTRING` geometry.

E7. Read the srtm.tif file into R (`srtm = rast(system.file("raster/srtm.tif", package = "spDataLarge"))`). This raster has a resolution of 0.00083 * 0.00083 degrees. Change its resolution to 0.01 * 0.01 degrees using all of the methods available in the **terra** package. Visualize the results. Can you notice any differences between the results of these resampling methods?

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

6

Raster-vector interactions

Prerequisites

- This chapter requires the following packages:

```
library(sf)
library(terra)
library(dplyr)
```

6.1 Introduction

This chapter focuses on interactions between raster and vector geographic data models, introduced in Chapter 2. It includes several main techniques: raster cropping and masking using vector objects (Section 6.2), extracting raster values using different types of vector data (Section 6.3), and raster-vector conversion (Sections 6.4 and 6.5). The above concepts are demonstrated using data from previous chapters to understand their potential real-world applications.

6.2 Raster cropping

Many geographic data projects involve integrating data from many different sources, such as remote sensing images (rasters) and administrative boundaries (vectors). Often the extent of input raster datasets is larger than the area of interest. In this case, raster **cropping** and **masking** are useful for unifying the spatial extent of input data. Both operations reduce object memory use and associated computational resources for subsequent analysis steps and may be

a necessary preprocessing step when creating attractive maps involving raster data.

We will use two objects to illustrate raster cropping:

- A `SpatRaster` object `srtm` representing elevation (meters above sea level) in southwestern Utah
- A vector (`sf`) object `zion` representing Zion National Park

Both target and cropping objects must have the same projection. The following code chunk therefore not only reads the datasets from the **spDataLarge** package installed in Chapter 2, but it also 'reprojects' `zion` (a topic covered in Chapter 7):

```
srtm = rast(system.file("raster/srtm.tif", package = "spDataLarge"))
zion = read_sf(system.file("vector/zion.gpkg", package = "spDataLarge"))
zion = st_transform(zion, st_crs(srtm))
```

We use `crop()` from the **terra** package to crop the `srtm` raster. The function reduces the rectangular extent of the object passed to its first argument based on the extent of the object passed to its second argument. This functionality is demonstrated in the command below, which generates Figure 6.1(B).

```
srtm_cropped = crop(srtm, zion)
```

Related to `crop()` is the **terra** function `mask()`, which sets values outside of the bounds of the object passed to its second argument to `NA`. The following command therefore masks every cell outside of Zion National Park boundaries (Figure 6.1(C)).

```
srtm_masked = mask(srtm, zion)
```

Importantly, we want to use both `crop()` and `mask()` together in most cases. This combination of functions would (a) limit the raster's extent to our area of interest and then (b) replace all of the values outside of the area to NA.[1]

```
srtm_cropped = crop(srtm, zion)
srtm_final = mask(srtm_cropped, zion)
```

Changing the settings of `mask()` yields different results. Setting `inverse = TRUE` will mask everything *inside* the bounds of the park (see `?mask` for details)

[1] These two operations can be combined into a single step with `terra::crop(srtm, zion, mask = TRUE)`, but we prefer to keep them separate for clarity.

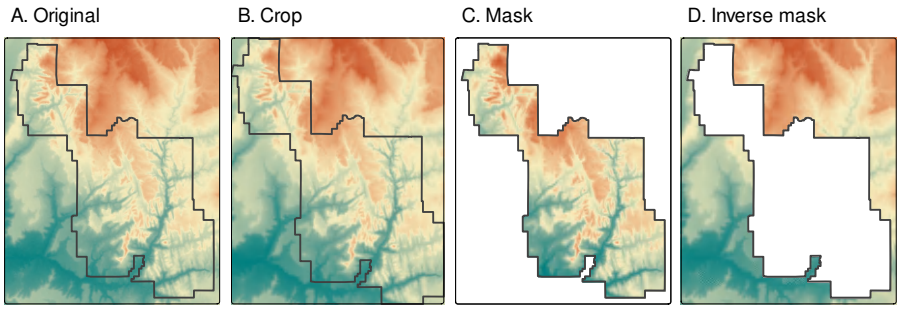

FIGURE 6.1 Raster cropping and raster masking.

(Figure 6.1(D)), while setting `updatevalue = 0` will set all pixels outside the national park to 0.

```
srtm_inv_masked = mask(srtm, zion, inverse = TRUE)
```

6.3 Raster extraction

Raster extraction is the process of identifying and returning the values associated with a 'target' raster at specific locations, based on a (typically vector) geographic 'selector' object. The results depend on the type of selector used (points, lines or polygons) and arguments passed to the `terra::extract()` function. The reverse of raster extraction — assigning raster cell values based on vector objects — is rasterization, described in Section 6.4.

The basic example is of extracting the value of a raster cell at specific **points**. For this purpose, we will use `zion_points`, which contain a sample of 30 locations within Zion National Park (Figure 6.2). The following command extracts elevation values from `srtm` and creates a data frame with points' IDs (one value per vector's row) and related `srtm` values for each point. Now, we can add the resulting object to our `zion_points` dataset with the `cbind()` function:

```
data("zion_points", package = "spDataLarge")
elevation = terra::extract(srtm, zion_points)
zion_points = cbind(zion_points, elevation)
```

Raster extraction also works with **line** selectors. Then, it extracts one value for each raster cell touched by a line. However, the line extraction approach is not recommended to obtain values along the transects, as it is hard to get the correct distance between each pair of extracted raster values.

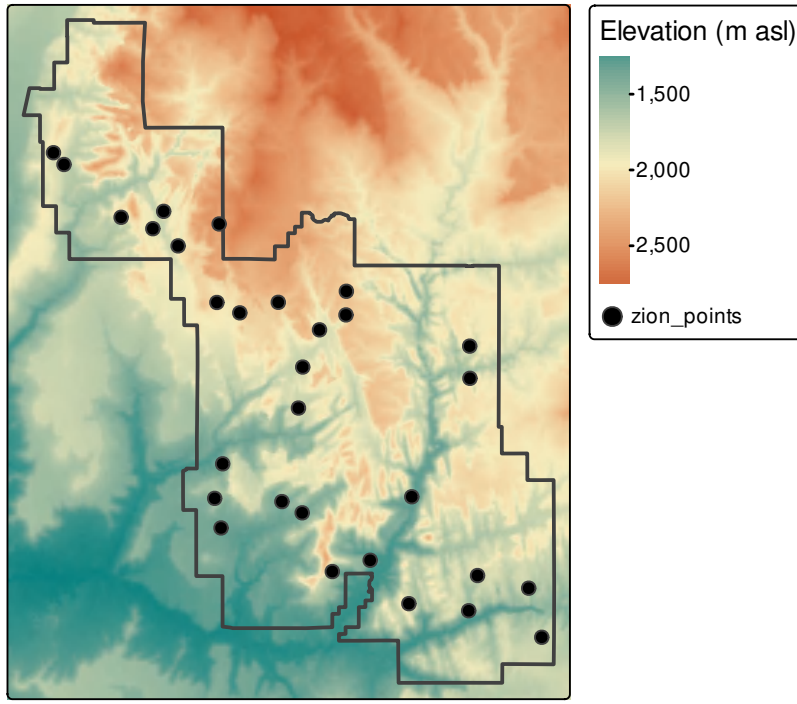

FIGURE 6.2 Locations of points used for raster extraction.

In this case, a better approach is to split the line into many points and then extract the values for these points. To demonstrate this, the code below creates `zion_transect`, a straight line going from northwest to southeast of Zion National Park, illustrated in Figure 6.3(A) (see Section 2.2 for a recap on the vector data model):

```
zion_transect = cbind(c(-113.2, -112.9), c(37.45, 37.2)) |>
  st_linestring() |>
  st_sfc(crs = crs(srtm)) |>
  st_sf(geometry = _)
```

The utility of extracting heights from a linear selector is illustrated by imagining that you are planning a hike. The method demonstrated below provides an 'elevation profile' of the route (the line does not need to be straight), useful for estimating how long it will take due to long climbs.

The first step is to add a unique `id` for each transect. Next, with the `st_segmentize()` function we can add points along our line(s) with a provided density (`dfMaxLength`) and convert them into points with `st_cast()`.

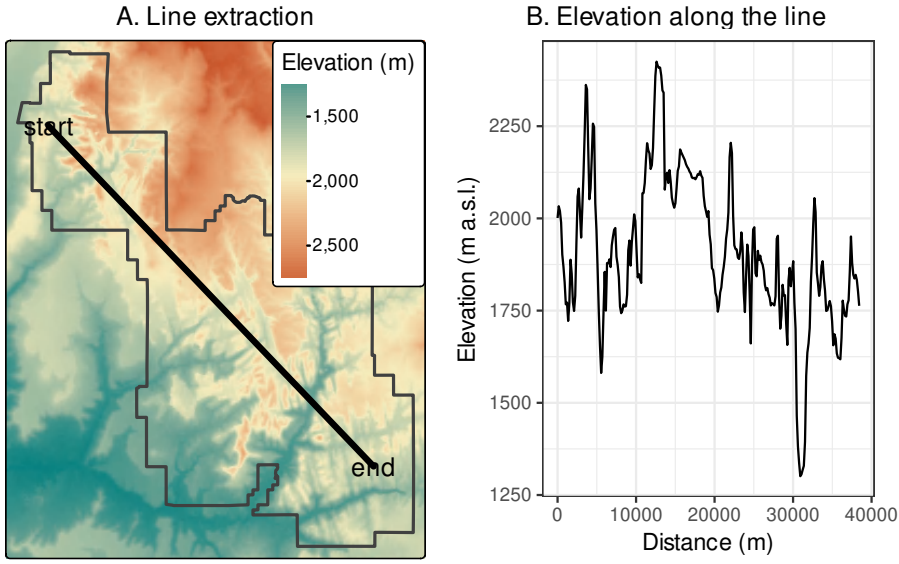

FIGURE 6.3 Location of a line used for (A) raster extraction and (B) the elevation along this line.

```
zion_transect$id = 1:nrow(zion_transect)
zion_transect = st_segmentize(zion_transect, dfMaxLength = 250)
zion_transect = st_cast(zion_transect, "POINT")
```

Now, we have a large set of points, and we want to derive a distance between the first point in our transects and each of the subsequent points. In this case, we only have one transect, but the code, in principle, should work on any number of transects:

```
zion_transect = zion_transect |>
  group_by(id) |>
  mutate(dist = st_distance(geometry)[, 1])
```

Finally, we can extract elevation values for each point in our transects and combine this information with our main object.

```
zion_elev = terra::extract(srtm, zion_transect)
zion_transect = cbind(zion_transect, zion_elev)
```

The resulting `zion_transect` can be used to create elevation profiles, as illustrated in Figure 6.3(B).

The final type of geographic vector object for raster extraction is **polygons**. Like lines, polygons tend to return many raster values per polygon. This is demonstrated in the command below, which results in a data frame with column names `ID` (the row number of the polygon) and `srtm` (associated elevation values):

```
zion_srtm_values = terra::extract(x = srtm, y = zion)
```

Such results can be used to generate summary statistics for raster values per polygon, for example to characterize a single region or to compare many regions. This is shown in the code below, which creates the object `zion_srtm_df` containing summary statistics for elevation values in Zion National Park (see Figure 6.4(A)):

```
group_by(zion_srtm_values, ID) |>
  summarize(across(srtm, list(min = min, mean = mean, max = max)))
#> # A tibble: 1 x 4
#>      ID srtm_min srtm_mean srtm_max
#>   <dbl>    <int>     <dbl>    <int>
#> 1     1     1122     1818.     2661
```

The preceding code chunk used **dplyr** to provide summary statistics for cell values per polygon ID, as described in Chapter 3. The results provide useful summaries, for example that the maximum height in the park is around 2,661 meters above sea level (other summary statistics, such as standard deviation, can also be calculated in this way). Because there is only one polygon in the example, a data frame with a single row is returned; however, the method works when multiple selector polygons are used.

A similar approach works for counting occurrences of categorical raster values within polygons. This is illustrated with a land cover dataset (`nlcd`) from the **spDataLarge** package in Figure 6.4(B), and demonstrated in the code below:

```
nlcd = rast(system.file("raster/nlcd.tif", package = "spDataLarge"))
zion2 = st_transform(zion, st_crs(nlcd))
zion_nlcd = terra::extract(nlcd, zion2)
zion_nlcd |>
  group_by(ID, levels) |>
  count()
#> # A tibble: 7 x 3
#> # Groups:   ID, levels [7]
#>      ID levels         n
#>   <dbl> <fct>      <int>
#> 1     1 Developed   4205
```

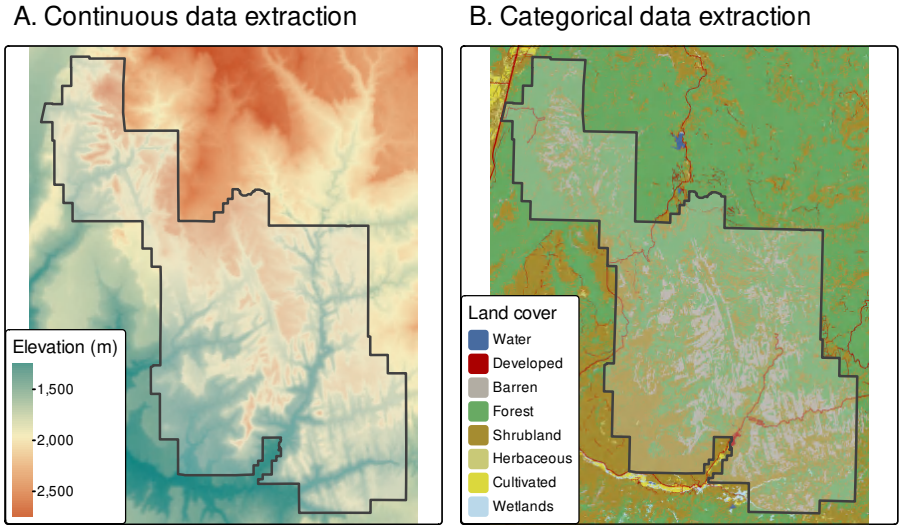

FIGURE 6.4 Area used for (A) continuous and (B) categorical raster extraction.

```
#> 2     1 Barren     98285
#> 3     1 Forest    298299
#> 4     1 Shrubland 203700
#> # i 3 more rows
```

Although the **terra** package offers rapid extraction of raster values within polygons, `extract()` can still be a bottleneck when processing large polygon datasets. The **exactextractr** package offers a significantly faster alternative for extracting pixel values through the `exact_extract()` function. The `exact_extract()` function also computes, by default, the fraction of each raster cell overlapped by the polygon, which is more precise (see note below for details).

Polygons usually have irregular shapes, and, therefore, a polygon can overlap only some parts of a raster's cells. To get more detailed results, the `terra::extract()` function has an argument called `exact`. With `exact = TRUE`, we get one more column `fraction` in the output data frame, which represents a fraction of each cell that is covered by the polygon. This could be useful to calculate, for example, a weighted mean for continuous rasters or more precise coverage for categorical rasters. By default, it is `FALSE`, as this operation requires more computations. The `exactextractr::exact_extract()` function always computes the coverage fraction of the polygon in each cell.

6.4 Rasterization

Rasterization is the conversion of vector objects into their representation in raster objects. Usually, the output raster is then used for quantitative analysis (e.g., analysis of terrain) or modeling. As we saw in Chapter 2, the raster data model has some characteristics that make it conducive to certain methods. Furthermore, the process of rasterization can help simplify datasets because the resulting values all have the same spatial resolution: rasterization can be seen as a special type of geographic data aggregation.

The **terra** package contains the function `rasterize()` for doing this work. Its first two arguments are, `x`, vector object to be rasterized and, `y`, a 'template raster' object defining the extent, resolution and CRS of the output. The geographic resolution of the input raster has a major impact on the results: if it is too low (cell size is too large), the result may miss the full geographic variability of the vector data; if it is too high, computational times may be excessive. There are no simple rules to follow when deciding an appropriate geographic resolution, which is heavily dependent on the intended use of the results. Often the target resolution is imposed on the user, for example when the output of rasterization needs to be aligned to some other existing raster.

To demonstrate rasterization in action, we will use a template raster that has the same extent and CRS as the input vector data `cycle_hire_osm_projected` (a dataset on cycle hire points in London is illustrated in Figure 6.5(A)) and spatial resolution of 1000 meters:

```
cycle_hire_osm = spData::cycle_hire_osm
cycle_hire_osm_projected = st_transform(cycle_hire_osm, "EPSG:27700")
raster_template = rast(ext(cycle_hire_osm_projected), resolution = 1000,
                       crs = crs(cycle_hire_osm_projected))
```

Rasterization is a very flexible operation: the results depend not only on the nature of the template raster, but also on the type of input vector (e.g., points, polygons) and a variety of arguments taken by the `rasterize()` function.

To illustrate this flexibility, we will try three different approaches to rasterization. First, we create a raster representing the presence or absence of cycle hire points (known as presence/absence rasters). In this case `rasterize()` requires no argument in addition to `x` and `y`, the aforementioned vector and raster objects (results illustrated Figure 6.5(B)).

```
ch_raster1 = rasterize(cycle_hire_osm_projected, raster_template)
```

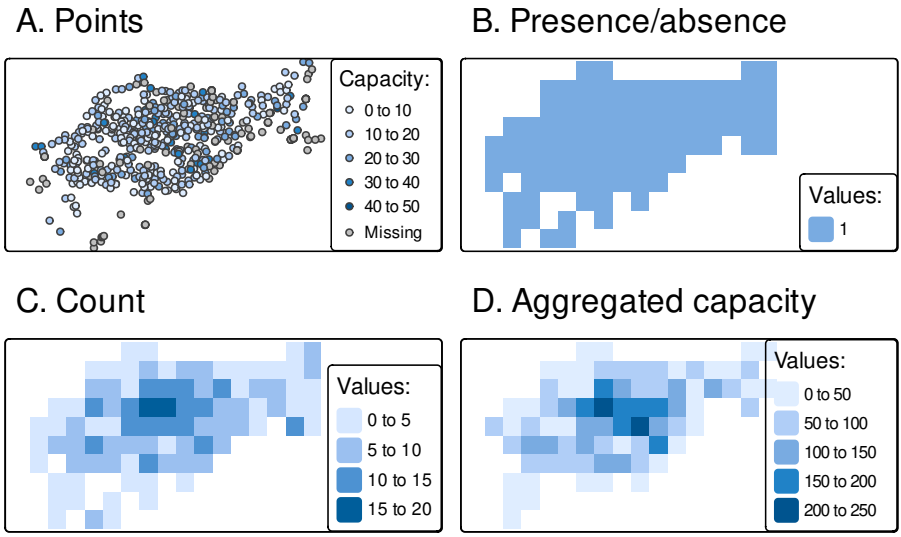

FIGURE 6.5 Examples of point rasterization.

The `fun` argument specifies summary statistics used to convert multiple observations in close proximity into associate cells in the raster object. By default `fun = "last"` is used, but other options such as `fun = "length"` can be used, in this case to count the number of cycle hire points in each grid cell (the results of this operation are illustrated in Figure 6.5(C)).

```
ch_raster2 = rasterize(cycle_hire_osm_projected, raster_template,
                       fun = "length")
```

The new output, `ch_raster2`, shows the number of cycle hire points in each grid cell. The cycle hire locations have different numbers of bicycles described by the `capacity` variable, raising the question, what's the capacity in each grid cell? To calculate that we must `sum` the field (`"capacity"`), resulting in output illustrated in Figure 6.5(D), calculated with the following command (other summary functions such as `mean` could be used).

```
ch_raster3 = rasterize(cycle_hire_osm_projected, raster_template,
                       field = "capacity", fun = sum, na.rm = TRUE)
```

Another dataset based on California's polygons and borders (created below) illustrates rasterization of lines. After casting the polygon objects into a multilinestring, a template raster is created with a resolution of a 0.5 degree:

```
california = dplyr::filter(us_states, NAME == "California")
california_borders = st_cast(california, "MULTILINESTRING")
raster_template2 = rast(ext(california), resolution = 0.5,
                        crs = st_crs(california)$wkt)
```

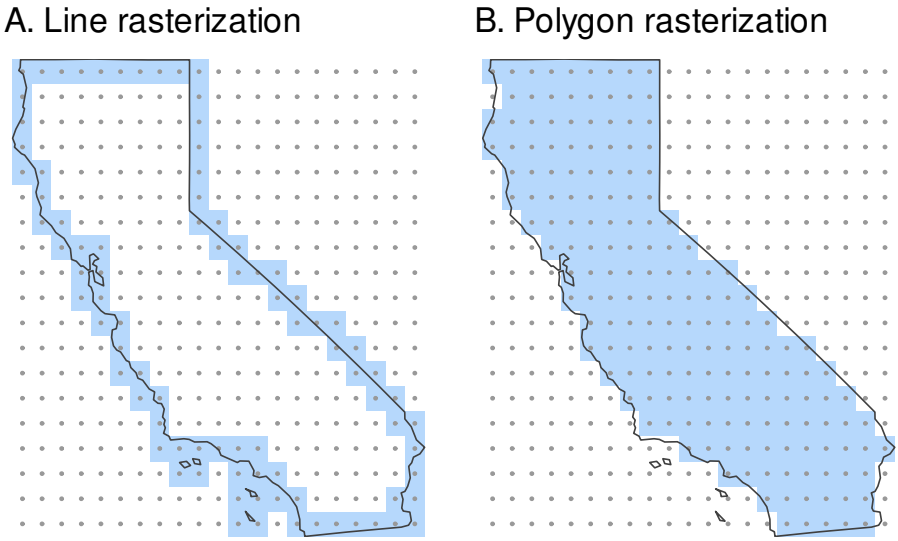

FIGURE 6.6 Examples of line and polygon rasterizations.

When considering line or polygon rasterization, one useful additional argument is `touches`. By default it is `FALSE`, but when changed to `TRUE`, all cells that are touched by a line or polygon border get a value. Line rasterization with `touches = TRUE` is demonstrated in the code below (Figure 6.6(A)).

```
california_raster1 = rasterize(california_borders, raster_template2,
                               touches = TRUE)
```

Compare it to a polygon rasterization, with `touches = FALSE` by default, which selects only raster cells whose centroids are inside the selector polygon, as illustrated in Figure 6.6(B).

```
california_raster2 = rasterize(california, raster_template2)
```

6.5 Spatial vectorization

Spatial vectorization is the counterpart of rasterization (Section 6.4), but in the opposite direction. It involves converting spatially continuous raster data into spatially discrete vector data such as points, lines or polygons.

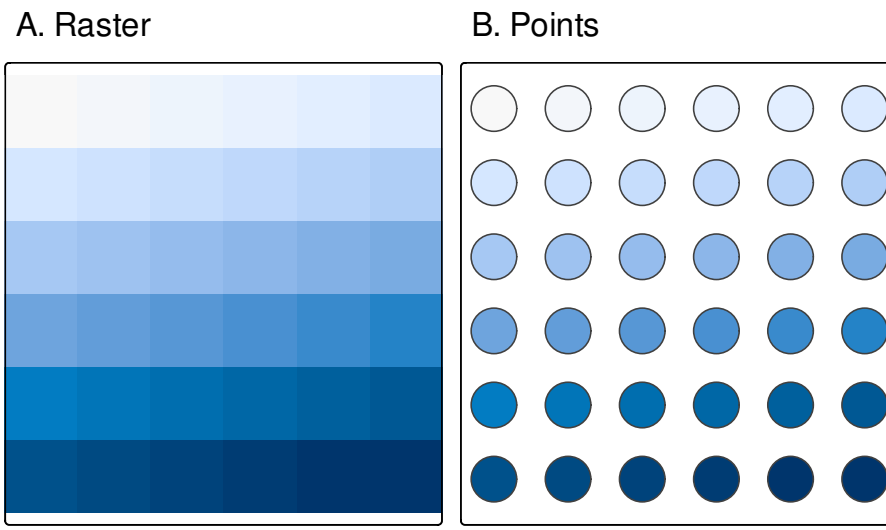

FIGURE 6.7 Raster and point representation of the elev object.

Be careful with the wording! In R, vectorization usually refers to the possibility of replacing `for`-loops and alike by doing things like `1:10 / 2` (see also Wickham (2019)).

The simplest form of vectorization is to convert the centroids of raster cells into points. `as.points()` does exactly this for all non-`NA` raster grid cells (Figure 6.7). Note, here we also used `st_as_sf()` to convert the resulting object to the `sf` class.

```
elev = rast(system.file("raster/elev.tif", package = "spData"))
elev_point = as.points(elev) |>
  st_as_sf()
```

Another common type of spatial vectorization is the creation of contour lines representing lines of continuous height or temperatures (isotherms), for example. We will use a real-world digital elevation model (DEM) because the artificial raster `elev` produces parallel lines (task for the reader: verify this and explain why this happens). Contour lines can be created with the **terra** function `as.contour()`, which is itself a wrapper around the built-in R function `filled.contour()`, as demonstrated below (not shown):

```
dem = rast(system.file("raster/dem.tif", package = "spDataLarge"))
cl = as.contour(dem) |>
```

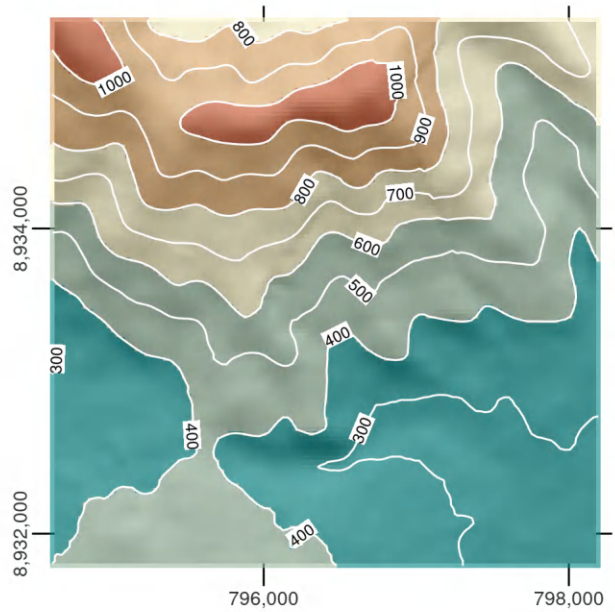

FIGURE 6.8 Digital elevation model with hillshading, showing the southern flank of Mt. Mongón overlaid with contour lines.

```
  st_as_sf()
plot(dem, axes = FALSE)
plot(cl, add = TRUE)
```

Contours can also be added to existing plots with functions such as `contour()`, `rasterVis::contourplot()`. As illustrated in Figure 6.8, isolines can be labeled.

The final type of vectorization involves conversion of rasters to polygons. This can be done with `terra::as.polygons()`, which converts each raster cell into a polygon consisting of five coordinates, all of which are stored in memory (explaining why rasters are often fast compared with vectors!).

This is illustrated below by converting the `grain` object into polygons and subsequently dissolving borders between polygons with the same attribute values (also see the `dissolve` argument in `as.polygons()`).

```
grain = rast(system.file("raster/grain.tif", package = "spData"))
grain_poly = as.polygons(grain) |>
  st_as_sf()
```

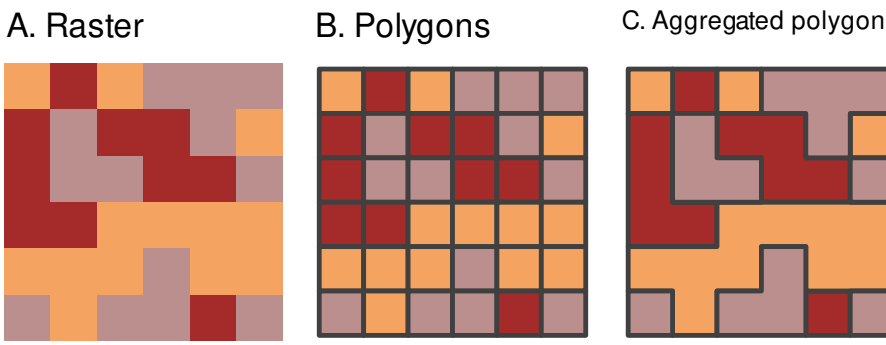

FIGURE 6.9 Vectorization of (A) raster into (B) polygons (dissolve = FALSE) and aggregated polygons (dissolve = TRUE).

The aggregated polygons of the `grain` dataset have rectilinear boundaries which arise from being defined by connecting rectangular pixels. The **smoothr** package described in Chapter 5 can be used to smooth the edges of the polygons. As smoothing removes sharp edges in the polygon boundaries, the smoothed polygons will not have the same exact spatial coverage as the original pixels. Caution should therefore be taken when using the smoothed polygons for further analysis.

6.6 Exercises

Some of the following exercises use a vector (`zion_points`) and raster dataset (`srtm`) from the **spDataLarge** package. They also use a polygonal 'convex hull' derived from the vector dataset (`ch`) to represent the area of interest:

```
library(sf)
library(terra)
library(spData)
zion_points_path = system.file("vector/zion_points.gpkg", package = "spDataLarge")
zion_points = read_sf(zion_points_path)
srtm = rast(system.file("raster/srtm.tif", package = "spDataLarge"))
ch = st_combine(zion_points) |>
  st_convex_hull() |>
  st_as_sf()
```

E1. Crop the `srtm` raster using (1) the `zion_points` dataset and (2) the `ch` dataset. Are there any differences in the output maps? Next, mask `srtm` using these two datasets. Can you see any difference now? How can you explain that?

E2. Firstly, extract values from `srtm` at the points represented in `zion_points`. Next, extract average values of `srtm` using a 90 buffer around each point from `zion_points` and compare these two sets of values. When would extracting values by buffers be more suitable than by points alone?

- Bonus: Implement extraction using the **exactextractr** package and compare the results.

E3. Subset points higher than 3100 meters in New Zealand (the `nz_height` object) and create a template raster with a resolution of 3 km for the extent of the new point dataset. Using these two new objects:

- Count numbers of the highest points in each grid cell.
- Find the maximum elevation in each grid cell.

E4. Aggregate the raster counting high points in New Zealand (created in the previous exercise), reduce its geographic resolution by half (so cells are 6 x 6 km) and plot the result.

- Resample the lower resolution raster back to the original resolution of 3 km. How have the results changed?
- Name two advantages and disadvantages of reducing raster resolution.

E5. Polygonize the `grain` dataset and filter all squares representing clay.

- Name two advantages and disadvantages of vector data over raster data.
- When would it be useful to convert rasters to vectors in your work?

7

Reprojecting geographic data

Prerequisites

- This chapter requires the following packages:

```
library(sf)
library(terra)
library(dplyr)
library(spData)
library(spDataLarge)
```

7.1 Introduction

Section 2.4 introduced coordinate reference systems (CRSs), with a focus on the two major types: *geographic* ('lon/lat', with units in degrees longitude and latitude) and *projected* (typically with units of meters from a datum) coordinate systems. This chapter builds on that knowledge and goes further. It demonstrates how to set and *transform* geographic data from one CRS to another and, furthermore, highlights specific issues that can arise due to ignoring CRSs that you should be aware of, especially if your data is stored with lon/lat coordinates.

In many projects there is no need to worry about, let alone convert between, different CRSs. Nonetheless, it is important to know if your data is in a projected or geographic CRS, and the consequences of this for geometry operations. If you know this information, CRSs should *just work* behind the scenes: people often suddenly need to learn about CRSs when things go wrong. Having a clearly defined CRS that all project data is in, and understanding how and why to use different CRSs, can ensure that things don't go wrong. Furthermore, learning about coordinate systems will deepen your knowledge of geographic datasets and how to use them effectively.

This chapter teaches the fundamentals of CRSs, demonstrates the consequences of using different CRSs (including what can go wrong), and how to 'reproject' datasets from one coordinate system to another. In the next section, we introduce CRSs in R, followed by Section 7.3 which shows how to get and set CRSs associated with spatial objects. Section 7.4 demonstrates the importance of knowing what CRS your data is in with reference to a worked example of creating buffers. We tackle questions of when to reproject and which CRS to use in Section 7.5 and Section 7.6, respectively. Finally, we cover reprojecting vector and raster objects in Sections 7.7 and 7.8 and modifying map projections in Section 7.9.

7.2 Coordinate reference systems

Most modern geographic tools that require CRS conversions, including core R-spatial packages and desktop GIS software such as QGIS, interface with PROJ, an open source C++ library that "transforms coordinates from one coordinate reference system (CRS) to another". CRSs can be described in many ways, including the following:

1. Simple yet potentially ambiguous statements such as "it's in lon/lat coordinates"
2. Formalized yet now outdated 'proj4 strings' (also known as 'proj-string') such as `+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs`
3. With an identifying 'authority:code' text string such as `EPSG:4326`

Each refers to the same thing: the 'WGS84' coordinate system that forms the basis of Global Positioning System (GPS) coordinates and many other datasets. But which one is correct?

The short answer is that the third way to identify CRSs is preferable: `EPSG:4326` is understood by **sf** (and by extension **stars**) and **terra** packages covered in this book, plus many other software projects for working with geographic data including QGIS and PROJ. `EPSG:4326` is future-proof. Furthermore, although it is machine readable, "EPSG:4326" is short, easy to remember and highly 'findable' online (searching for EPSG:4326 yields a dedicated page on the website epsg.io, for example). The more concise identifier `4326` is understood by **sf**, but **we recommend the more explicit `AUTHORITY:CODE` representation to prevent ambiguity and to provide context**.

The longer answer is that none of the three descriptions are sufficient, and more detail is needed for unambiguous CRS handling and transformations: due to the complexity of CRSs, it is not possible to capture all relevant information about them in such short text strings. For this reason, the Open

Geospatial Consortium (OGC, which also developed the simple features specification that the **sf** package implements) developed an open standard format for describing CRSs that is called WKT (Well-Known Text). This is detailed in a 100+ page document that "defines the structure and content of a text string implementation of the abstract model for coordinate reference systems described in ISO 19111:2019" (Open Geospatial Consortium, 2019). The WKT representation of the WGS84 CRS, which has the **identifier** `EPSG:4326` is as follows:

```
st_crs("EPSG:4326")
#> Coordinate Reference System:
#>   User input: EPSG:4326
#>   wkt:
#> GEOGCRS["WGS 84",
#>     ENSEMBLE["World Geodetic System 1984 ensemble",
#>         MEMBER["World Geodetic System 1984 (Transit)"],
#>         MEMBER["World Geodetic System 1984 (G730)"],
#>         MEMBER["World Geodetic System 1984 (G873)"],
#>         MEMBER["World Geodetic System 1984 (G1150)"],
#>         MEMBER["World Geodetic System 1984 (G1674)"],
#>         MEMBER["World Geodetic System 1984 (G1762)"],
#>         MEMBER["World Geodetic System 1984 (G2139)"],
#>         MEMBER["World Geodetic System 1984 (G2296)"],
#>         ELLIPSOID["WGS 84",6378137,298.257223563,
#>             LENGTHUNIT["metre",1]],
#>         ENSEMBLEACCURACY[2.0]],
#>     PRIMEM["Greenwich",0,
#>         ANGLEUNIT["degree",0.0174532925199433]],
#>     CS[ellipsoidal,2],
#>         AXIS["geodetic latitude (Lat)",north,
#>             ORDER[1],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         AXIS["geodetic longitude (Lon)",east,
#>             ORDER[2],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>     USAGE[
#>         SCOPE["Horizontal component of 3D system."],
#>         AREA["World."],
#>         BBOX[-90,-180,90,180]],
#>     ID["EPSG",4326]]
```

The output of the command shows how the CRS identifier (also known as a Spatial Reference Identifier or SRID) works: it is simply a look-up, providing a unique identifier associated with a more complete WKT representation of the CRS. This raises the question: what happens if there is a mismatch between

the identifier and the longer WKT representation of a CRS? On this point Open Geospatial Consortium (2019) is clear, the verbose WKT representation takes precedence over the identifier:

> Should any attributes or values given in the cited identifier be in conflict with attributes or values given explicitly in the WKT description, the WKT values shall prevail.

The convention of referring to CRSs identifiers in the form `AUTHORITY:CODE`, which is also used by geographic software written in other languages, allows a wide range of formally defined coordinate systems to be referred to.[1] The most commonly used authority in CRS identifiers is *EPSG*, an acronym for the European Petroleum Survey Group which published a standardized list of CRSs (the EPSG was taken over by the Geomatics Committee of the International Association of Oil & Gas Producers in 2005). Other authorities can be used in CRS identifiers. `ESRI:54030`, for example, refers to ESRI's implementation of the Robinson projection, which has the following WKT string (only first eight lines shown):

```
st_crs("ESRI:54030")
#> Coordinate Reference System:
#>   User input: ESRI:54030
#>   wkt:
#> PROJCRS["World_Robinson",
#>     BASEGEOGCRS["WGS 84",
#>         DATUM["World Geodetic System 1984",
#>             ELLIPSOID["WGS 84",6378137,298.257223563,
#>                 LENGTHUNIT["metre",1]]],
....
```

WKT strings are exhaustive, detailed, and precise, allowing for unambiguous CRSs storage and transformations. They contain all relevant information about any given CRS, including its datum and ellipsoid, prime meridian, projection, and units.[2]

[1] Several other ways of referring to unique CRSs can be used, with five identifier types (EPSG code, PostGIS SRID, INTERNAL SRID, proj-string, and WKT strings) accepted by QGIS and other identifier types such as a more verbose variant of the `EPSG:4326` identifier, `urn:ogc:def:crs:EPSG::4326` (Open Geospatial Consortium, 2019).

[2] Before the emergence of WKT CRS definitions, proj-string was the standard way to specify coordinate operations and store CRSs. These string representations, built on a key=value

Recent PROJ versions (6+) still allow use of proj-strings to define coordinate operations, but some proj-string keys (`+nadgrids`, `+towgs84`, `+k`, `+init=epsg:`) are either no longer supported or are discouraged. Additionally, only three datums (i.e., WGS84, NAD83, and NAD27) can be directly set in proj-string. Longer explanations of the evolution of CRS definitions and the PROJ library can be found in Bivand (2021), chapter 2 of Pebesma and Bivand (2023b), and a blog post by Floris Vanderhaeghe, available at inbo.github.io/tutorials/tutorials/spatial_crs_coding/. Also, as outlined in the PROJ documentation there are different versions of the WKT CRS format including WKT1 and two variants of WKT2, the latter of which (WKT2, 2018 specification) corresponds to the ISO 19111:2019 (Open Geospatial Consortium, 2019).

7.3 Querying and setting coordinate systems

Let's look at how CRSs are stored in R spatial objects and how they can be queried and set. First, we will look at getting and setting CRSs in **vector** geographic data objects, starting with the following example:

```
vector_filepath = system.file("shapes/world.gpkg", package = "spData")
new_vector = read_sf(vector_filepath)
```

Our new object, `new_vector`, is a data frame of class `sf` that represents countries worldwide (see the help page `?spData::world` for details). The CRS can be retrieved with the **sf** function `st_crs()`.

```
st_crs(new_vector) # get CRS
#> Coordinate Reference System:
#>   User input: WGS 84
#>   wkt:
#>   ...
```

The output is a list containing two main components:

1. `User input` (in this case `WGS 84`, a synonym for `EPSG:4326` which in this case was taken from the input file), corresponding to CRS identifiers described above
2. `wkt`, containing the full WKT string with all relevant information about the CRS

form (e.g, `+proj=longlat +datum=WGS84 +no_defs`), have already been, or should in the future be, superseded by WKT representations in most cases.

The `input` element is flexible, and depending on the input file or user input, can contain the `AUTHORITY:CODE` representation (e.g., `EPSG:4326`), the CRS's name (e.g., `WGS 84`), or even the proj-string definition. The `wkt` element stores the WKT representation, which is used when saving the object to a file or doing any coordinate operations. Above, we can see that the `new_vector` object has the WGS84 ellipsoid, uses the Greenwich prime meridian, and the latitude and longitude axis order. In this case, we also have some additional elements, such as `USAGE` explaining the area suitable for the use of this CRS, and `ID` pointing to the CRS's identifier: `EPSG:4326`.

The `st_crs` function also has one helpful feature – we can retrieve some additional information about the used CRS. For example, try to run:

- `st_crs(new_vector)$IsGeographic` to check if the CRS is geographic or not
- `st_crs(new_vector)$units_gdal` to find out the CRS units
- `st_crs(new_vector)$srid` to extract its 'SRID' identifier (when available)
- `st_crs(new_vector)$proj4string` to extract the proj-string representation

In cases when a CRS is missing or the wrong CRS is set, the `st_set_crs()` function can be used (in this case the WKT string remains unchanged because the CRS was already set correctly when the file was read-in):

```
new_vector = st_set_crs(new_vector, "EPSG:4326") # set CRS
```

Getting and setting CRSs works in a similar way for raster geographic data objects. The `crs()` function in the `terra` package accesses CRS information from a `SpatRaster` object (note the use of the `cat()` function to print it nicely).

```
raster_filepath = system.file("raster/srtm.tif", package = "spDataLarge")
my_rast = rast(raster_filepath)
cat(crs(my_rast)) # get CRS
#> GEOGCRS["WGS 84",
#>     ENSEMBLE["World Geodetic System 1984 ensemble",
#>         MEMBER["World Geodetic System 1984 (Transit)"],
#>         MEMBER["World Geodetic System 1984 (G730)"],
#>         MEMBER["World Geodetic System 1984 (G873)"],
#>         MEMBER["World Geodetic System 1984 (G1150)"],
....
```

The output is the WKT string representation of CRS. The same function, `crs()`, can be also used to set a CRS for raster objects.

```
crs(my_rast) = "EPSG:26912" # set CRS
```

Here, we can use either the identifier (recommended in most cases) or complete WKT string representation. Alternative methods to set `crs` include proj-string

strings or CRSs extracted from other existing objects with `crs()`, although these approaches may be less future-proof.

Importantly, the `st_crs()` and `crs()` functions do not alter coordinates' values or geometries. Their role is only to set a metadata information about the object CRS.

In some cases the CRS of a geographic object is unknown, as is the case in the `london` dataset created in the code chunk below, building on the example of London introduced in Section 2.2:

```
london = data.frame(lon = -0.1, lat = 51.5) |>
  st_as_sf(coords = c("lon", "lat"))
st_is_longlat(london)
#> [1] NA
```

The output `NA` shows that **sf** does not know what the CRS is and is unwilling to guess (`NA` literally means 'not available'). Unless a CRS is manually specified or is loaded from a source that has CRS metadata, **sf** does not make any explicit assumptions about which coordinate systems, other than to say "I don't know". This behavior makes sense given the diversity of available CRSs but differs from some approaches, such as the GeoJSON file format specification, which makes the simplifying assumption that all coordinates have a lon/lat CRS: `EPSG:4326`. Datasets without a specified CRS can cause problems: all geographic coordinates have a coordinate reference system, and software can only make good decisions around plotting and geometry operations if it knows what type of CRS it is working with. Thus, again, it is important to always check the CRS of a dataset and to set it if it is missing.

```
london_geo = st_set_crs(london, "EPSG:4326")
st_is_longlat(london_geo)
#> [1] TRUE
```

7.4 Geometry operations on projected and unprojected data

Since **sf** version 1.0.0, R's ability to work with geographic vector datasets that have lon/lat CRSs has improved substantially, thanks to its integration with the S2 *spherical geometry engine* introduced in Section 2.2.9. As shown in Figure 7.1, **sf** uses either GEOS or the S2 depending on the type of CRS

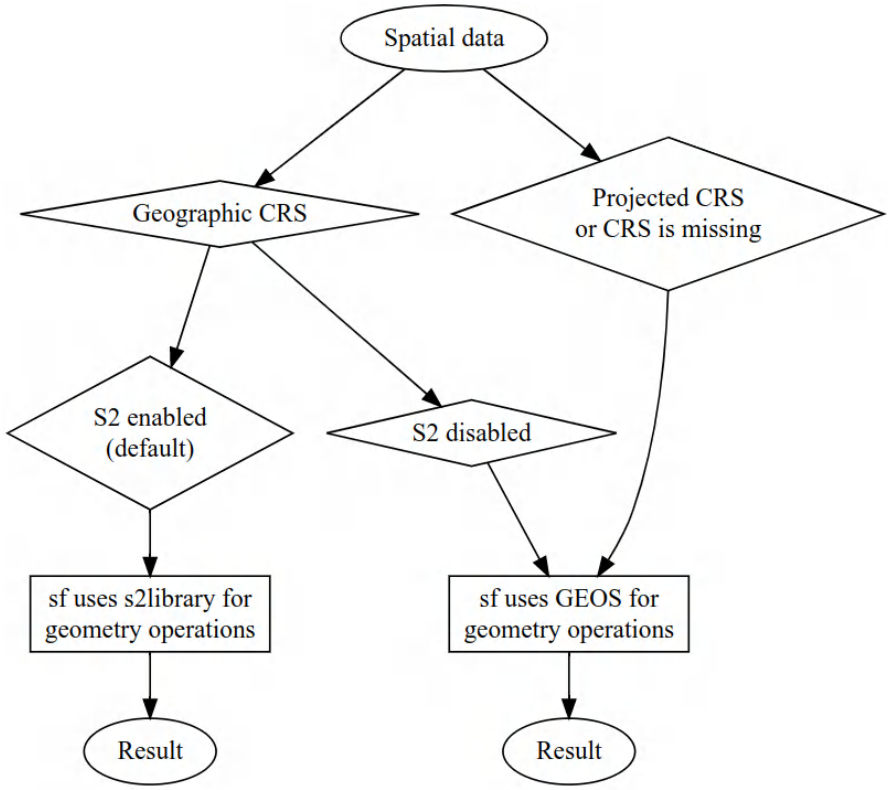

FIGURE 7.1 Behavior of the geometry operations in the sf package depending on the input data's CRS.

and whether S2 has been disabled (it is enabled by default).[3] GEOS is always used for projected data and data with no CRS; for geographic data, S2 is used by default but can be disabled with `sf::sf_use_s2(FALSE)`.

To demonstrate the importance of CRSs, we will create a buffer of 100 km around the `london` object from the previous section. We will also create a deliberately faulty buffer with a 'distance' of 1 degree, which is roughly equivalent to 100 km (1 degree is about 111 km at the equator). Before diving into the code, it may be worth skipping briefly ahead to peek at Figure 7.2 to get a visual handle on the outputs that you should be able to reproduce by following the code chunks below.

The first stage is to create three buffers around the `london` and `london_geo` objects created above with boundary distances of 1 degree and 100 km (or

[3]The `st_area()` function is an exception, as it uses the **lwgeom**'s `st_geod_area()` function to calculate areas for data with geographic CRSs when `sf_use_s2()` is disabled.

100,000 m, which can be expressed as `1e5` in scientific notation) from central London:

```
london_buff_no_crs = st_buffer(london, dist = 1)    # incorrect: no CRS
london_buff_s2 = st_buffer(london_geo, dist = 100000) # silent use of s2
london_buff_s2_100_cells = st_buffer(london_geo, dist = 100000, max_cells = 100)
```

In the first line above, **sf** assumes that the input is projected and generates a result that has a buffer in units of degrees, which is problematic, as we will see. In the second line, **sf** silently uses the spherical geometry engine S2, introduced in Chapter 2, to calculate the extent of the buffer using the default value of `max_cells = 1000` — set to `100` in line three — the consequences which will become apparent shortly. To highlight the impact of **sf**'s use of the S2 geometry engine for unprojected (geographic) coordinate systems, we will temporarily disable it with the command `sf_use_s2()` (which is on, `TRUE`, by default), in the code chunk below. Like `london_buff_no_crs`, the new `london_geo` object is a geographic abomination: it has units of degrees, which makes no sense in the vast majority of cases:

```
sf::sf_use_s2(FALSE)
#> Spherical geometry (s2) switched off
london_buff_lonlat = st_buffer(london_geo, dist = 1) # incorrect result
#> Warning in st_buffer.sfc(st_geometry(x), dist, nQuadSegs, endCapStyle =
#> endCapStyle, : st_buffer does not correctly buffer longitude/latitude data
#> dist is assumed to be in decimal degrees (arc_degrees).
sf::sf_use_s2(TRUE)
#> Spherical geometry (s2) switched on
```

The warning message above hints at issues with performing planar geometry operations on lon/lat data. When spherical geometry operations are turned off, with the command `sf::sf_use_s2(FALSE)`, buffers (and other geometric operations) may result in worthless outputs because they use units of latitude and longitude, a poor substitute for proper units of distances such as meters.

The distance between two lines of longitude, called meridians, is around 111 km at the equator (execute `geosphere::distGeo(c(0, 0), c(1, 0))` to find the precise distance). This shrinks to zero at the poles. At the latitude of London, for example, meridians are less than 70 km apart (challenge: execute code that verifies this). Lines of latitude, by contrast, are equidistant from each other irrespective of latitude: they are always around 111 km apart, including at the equator and near the poles (see Figures 7.2 to 7.4).

Do not interpret the warning about the geographic (`longitude/latitude`) CRS as "the CRS should not be set": it almost always should be! It is better understood as a suggestion to *reproject* the data onto a projected CRS. This

suggestion does not always need to be heeded: performing spatial and geometric operations makes little or no difference in some cases (e.g., spatial subsetting). But for operations involving distances such as buffering, the only way to ensure a good result (without using spherical geometry engines) is to create a projected copy of the data and run the operation on that. This is done in the code chunk below.

```
london_proj = data.frame(x = 530000, y = 180000) |>
  st_as_sf(coords = c("x", "y"), crs = "EPSG:27700")
```

The result is a new object that is identical to `london`, but created using a suitable CRS (the British National Grid, which has an EPSG code of 27700 in this case) that has units of meters. We can verify that the CRS has changed using `st_crs()` as follows (some of the output has been replaced by ...,):

```
st_crs(london_proj)
#> Coordinate Reference System:
#>   User input: EPSG:27700
#>   wkt:
#> PROJCRS["OSGB36 / British National Grid",
#>     BASEGEOGCRS["OSGB36",
#>         DATUM["Ordnance Survey of Great Britain 1936",
#>             ELLIPSOID["Airy 1830",6377563.396,299.3249646,
#>                 LENGTHUNIT["metre",1]]],
....
```

Notable components of this CRS description include the EPSG code (`EPSG: 27700`) and the detailed `wkt` string (only the first five lines of which are shown).[4] The fact that the units of the CRS, described in the LENGTHUNIT field, are meters (rather than degrees) tells us that this is a projected CRS: `st_is_longlat(london_proj)` now returns `FALSE` and geometry operations on `london_proj` will work without a warning. Buffer operations on the `london_proj` will use GEOS, and results will be returned with proper units of distance. The following line of code creates a buffer around *projected* data of exactly 100 km:

```
london_buff_projected = st_buffer(london_proj, 100000)
```

The geometries of the three `london_buff*` objects created in the preceding code that *have* a specified CRS (`london_buff_s2`, `london_buff_lonlat` and `london_buff_projected`) are illustrated in Figure 7.2.

[4] For a short description of the most relevant projection parameters and related concepts, see the fourth lecture by Jochen Albrecht hosted at http://www.geography.hunter.cuny.edu/~jochen/GTECH361/lectures/ and information at https://proj.org/usage/projections.html.

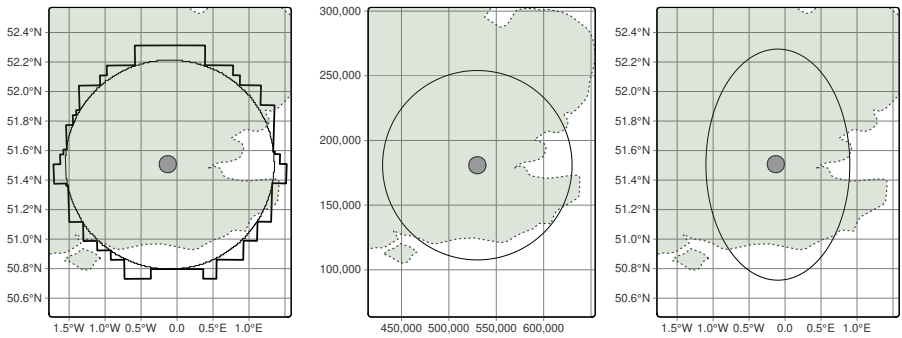

FIGURE 7.2 Buffers around London showing results created with the S2 spherical geometry engine on lon/lat data (left), projected data (middle) and lon/lat data without using spherical geometry (right). The left plot illustrates the result of buffering unprojected data with sf, which calls Google's S2 spherical geometry engine by default with max cells set to 1000 (thin line). The thick, blocky line illustrates the result of the same operation with max cells set to 100.

It is clear from Figure 7.2 that buffers based on `s2` and properly projected CRSs are not 'squashed', meaning that every part of the buffer boundary is equidistant to London. The results that are generated from lon/lat CRSs when `s2` is *not* used, either because the input lacks a CRS or because `sf_use_s2()` is turned off, are heavily distorted, with the result elongated in the north-south axis, highlighting the dangers of using algorithms that assume projected data on lon/lat inputs (as GEOS does). The results generated using S2 are also distorted, however, although less dramatically. Both buffer boundaries in Figure 7.2 (left) are jagged, although this may only be apparent or relevant for the thick boundary representing a buffer created with the `s2` argument `max_cells` set to 100. The lesson is that results obtained from lon/lat data via S2 will be different from results obtained from using projected data. The difference between S2 derived buffers and GEOS derived buffers on projected data reduce as the value of `max_cells` increases: the 'right' value for this argument may depend on many factors and the default value 1000 is often a reasonable default. When choosing `max_cells` values, speed of computation should be balanced against resolution of results. In situations where smooth curved boundaries are advantageous, transforming to a projected CRS before buffering (or performing other geometry operations) may be appropriate.

The importance of CRSs (primarily whether they are projected or geographic) and the impacts of **sf**'s default setting to use S2 for buffers on lon/lat data is clear from the example above. The subsequent sections go into more depth, exploring which CRS to use when projected CRSs *are* needed and the details of reprojecting vector and raster objects.

7.5 When to reproject?

The previous section showed how to set the CRS manually, with `st_set_crs(london, "EPSG:4326")`. In real-world applications, however, CRSs are usually set automatically when data is read-in. In many projects the main CRS-related task is to *transform* objects, from one CRS into another. But when should data be transformed? And into which CRS? There are no clear-cut answers to these questions and CRS selection always involves trade-offs (Maling, 1992). However, there are some general principles provided in this section that can help you decide.

First it's worth considering *when to transform*. In some cases transformation to a geographic CRS is essential, such as when publishing data online with the **leaflet** package. Another case is when two objects with different CRSs must be compared or combined, as shown when we try to find the distance between two `sf` objects with different CRSs:

```
st_distance(london_geo, london_proj)
# > Error: st_crs(x) == st_crs(y) is not TRUE
```

To make the `london` and `london_proj` objects geographically comparable, one of them must be transformed into the CRS of the other. But which CRS to use? The answer depends on context: many projects, especially those involving web mapping, require outputs in EPSG:4326, in which case it is worth transforming the projected object. If, however, the project requires planar geometry operations rather than spherical geometry operations engine (e.g., to create buffers with smooth edges), it may be worth transforming data with a geographic CRS into an equivalent object with a projected CRS, such as the British National Grid (EPSG:27700). That is the subject of Section 7.7.

7.6 Which CRS to use?

The question of *which CRS to use* is tricky, and there is rarely a 'right' answer: "There exist no all-purpose projections, all involve distortion when far from the center of the specified frame" (Bivand et al., 2013). Additionally, you should not be attached just to one projection for every task. It is possible to use one projection for some part of the analysis, another projection for a different part, and even some other for visualization. Always try to pick the CRS that serves your goal best!

When selecting **geographic CRSs**, the answer is often WGS84. It is used not only for web mapping, but also because GPS datasets and thousands of raster and vector datasets are provided in this CRS by default. WGS84 is the most common CRS in the world, so it is worth knowing its EPSG code: 4326.[5] This 'magic number' can be used to convert objects with unusual projected CRSs into something that is widely understood.

What about when a **projected CRS** is required? In some cases, it is not something that we are free to decide: "often the choice of projection is made by a public mapping agency" (Bivand et al., 2013). This means that when working with local data sources, it is likely preferable to work with the CRS in which the data was provided, to ensure compatibility, even if the official CRS is not the most accurate. The example of London was easy to answer because (a) the British National Grid (with its associated EPSG code 27700) is well known and (b) the original dataset (`london`) already had that CRS.

A commonly used default is Universal Transverse Mercator (UTM), a set of CRSs that divides the Earth into 60 longitudinal wedges and 20 latitudinal segments. Almost every place on Earth has a UTM code, such as "60H" which refers to northern New Zealand where R was invented. UTM EPSG codes run sequentially from 32601 to 32660 for northern hemisphere locations and from 32701 to 32760 for southern hemisphere locations.

To show how the system works, let's create a function, `lonlat2UTM()` to calculate the EPSG code associated with any point on the planet as follows:

```
lonlat2UTM = function(lonlat) {
  utm = (floor((lonlat[1] + 180) / 6) %% 60) + 1
  if (lonlat[2] > 0) {
    utm + 32600
  } else{
    utm + 32700
  }
}
```

The following command uses this function to identify the UTM zone and associated EPSG code for Auckland and London:

```
lonlat2UTM(c(174.7, -36.9))
#> [1] 32760
lonlat2UTM(st_coordinates(london))
#> [1] 32630
```

[5] Instead of `"EPSG:4326"`, you may also use `"OGC:CRS84"`. The former assumes that latitude is always ordered before longitude, while the latter is the standard representation used by GeoJSON, with coordinates ordered longitude before latitude.

The transverse Mercator projection used by UTM CRSs is conformal but distorts areas and distances with increasing severity with distance from the center of the UTM zone. Documentation from the GIS software Manifold therefore suggests restricting the longitudinal extent of projects using UTM zones to 6 degrees from the central meridian (manifold.net). Therefore, we recommend using UTM only when your focus is on preserving angles for a relatively small area!

Currently, we also have tools helping us to select a proper CRS, which includes the **crsuggest** package (Walker (2022a)). The main function in this package, `suggest_crs()`, takes a spatial object with geographic CRS and returns a list of possible projected CRSs that could be used for the given area.[6] Another helpful tool is the webpage https://jjimenezshaw.github.io/crs-explorer/ that lists CRSs based on selected location and type. Important note: while these tools are helpful in many situations, you need to be aware of the properties of the recommended CRS before you apply it.

In cases where an appropriate CRS is not immediately clear, the choice of CRS should depend on the properties that are most important to preserve in the subsequent maps and analysis. CRSs are either equal-area, equidistant, conformal (with shapes remaining unchanged), or some combination of compromises of those (Section 2.4.2). Custom CRSs with local parameters can be created for a region of interest and multiple CRSs can be used in projects when no single CRS suits all tasks. 'Geodesic calculations' can provide a fall-back if no CRSs are appropriate (see proj.org/geodesic.html). Regardless of the projected CRS used, the results may not be accurate for geometries covering hundreds of kilometers.

When deciding on a custom CRS, we recommend the following:[7]

- A Lambert azimuthal equal-area (LAEA) projection for a custom local projection (set latitude and longitude of origin to the center of the study area), which is an equal-area projection at all locations but distorts shapes beyond thousands of kilometers
- Azimuthal equidistant (AEQD) projections for a specifically accurate straight-line distance between a point and the center point of the local projection
- Lambert conformal conic (LCC) projections for regions covering thousands of kilometers, with the cone set to keep distance and area properties reasonable between the secant lines
- Stereographic (STERE) projections for polar regions, but taking care not to rely on area and distance calculations thousands of kilometers from the center

[6] This package also allows to figure out the true CRS of the data without any CRS information attached.

[7] Many thanks to an anonymous reviewer whose comments formed the basis of this advice.

One possible approach to automatically select a projected CRS specific to a local dataset is to create an AEQD projection for the center-point of the study area. This involves creating a custom CRS (with no EPSG code) with units of meters based on the center point of a dataset. Note that this approach should be used with caution: no other datasets will be compatible with the custom CRS created, and results may not be accurate when used on extensive datasets covering hundreds of kilometers.

The principles outlined in this section apply equally to vector and raster datasets. Some features of CRS transformation, however, are unique to each geographic data model. We will cover the particularities of vector data transformation in Section 7.7 and those of raster transformation in Section 7.8. Next, Section 7.9, shows how to create custom map projections.

7.7 Reprojecting vector geometries

Chapter 2 demonstrated how vector geometries are made up of points, and how points form the basis of more complex objects such as lines and polygons. Reprojecting vectors thus consists of transforming the coordinates of these points, which form the vertices of lines and polygons.

Section 7.5 contains an example in which at least one `sf` object must be transformed into an equivalent object with a different CRS to calculate the distance between two objects.

```
london2 = st_transform(london_geo, "EPSG:27700")
```

Now that a transformed version of `london` has been created, using the **sf** function `st_transform()`, the distance between the two representations of London can be found.[8] It may come as a surprise that `london` and `london2` are over 2 km apart![9]

```
st_distance(london2, london_proj)
#> Units: [m]
```

[8] An alternative to `st_transform()` is `st_transform_proj()` from the **lwgeom**, which enables transformations and which bypasses GDAL and can support projections not supported by GDAL. However, at the time of writing (2024) we could not find any projections supported by `st_transform_proj()` but not supported by `st_transform()`.

[9] The difference in location between the two points is not due to imperfections in the transforming operation (which is in fact very accurate) but the low precision of the manually-created coordinates that created `london` and `london_proj`. Also surprising may be that the result is provided in a matrix with units of meters. This is because `st_distance()` can provide distances between many features and because the CRS has units of meters. Use `as.numeric()` to coerce the result into a regular number.

```
#>      [,1]
#> [1,] 2016
```

Functions for querying and reprojecting CRSs are demonstrated below with reference to `cycle_hire_osm`, an `sf` object from **spData** that represents 'docking stations' where you can hire bicycles in London. The CRS of `sf` objects can be queried, and as we learned in Section 7.1, set with the function `st_crs()`. The output is printed as multiple lines of text containing information about the coordinate system:

```
st_crs(cycle_hire_osm)
#> Coordinate Reference System:
#>   User input: EPSG:4326
#>   wkt:
#> GEOGCS["WGS 84",
#>     DATUM["WGS_1984",
#>         SPHEROID["WGS 84",6378137,298.257223563,
....
```

As we saw in Section 7.3, the main CRS components, `User input` and `wkt`, are printed as a single entity. The output of `st_crs()` is in fact a named list of class `crs` with two elements, single character strings named `input` and `wkt`, as shown in the output of the following code chunk:

```
crs_lnd = st_crs(london_geo)
class(crs_lnd)
#> [1] "crs"
names(crs_lnd)
#> [1] "input" "wkt"
```

Additional elements can be retrieved with the `$` operator, including `Name`, `proj4string` and `epsg` (see `?st_crs` and the CRS and tranformation tutorial on the GDAL website for details):

```
crs_lnd$Name
#> [1] "WGS 84"
crs_lnd$proj4string
#> [1] "+proj=longlat +datum=WGS84 +no_defs"
crs_lnd$epsg
#> [1] 4326
```

As mentioned in Section 7.2, WKT representation, stored in the `$wkt` element of the `crs_lnd` object is the ultimate source of truth. This means that the

outputs of the previous code chunk are queries from the `wkt` representation provided by PROJ, rather than inherent attributes of the object and its CRS.

Both `wkt` and `User Input` elements of the CRS are changed when the object's CRS is transformed. In the code chunk below, we create a new version of `cycle_hire_osm` with a projected CRS (only the first 4 lines of the CRS output are shown for brevity).

```
cycle_hire_osm_projected = st_transform(cycle_hire_osm, "EPSG:27700")
st_crs(cycle_hire_osm_projected)
#> Coordinate Reference System:
#>   User input: EPSG:27700
#>   wkt:
#> PROJCRS["OSGB36 / British National Grid",
#> ...
```

The resulting object has a new CRS with an EPSG code 27700. But how do we find out more details about this EPSG code, or any code? One option is to search for it online, another is to look at the properties of the CRS object:

```
crs_lnd_new = st_crs("EPSG:27700")
crs_lnd_new$Name
#> [1] "OSGB36 / British National Grid"
crs_lnd_new$proj4string
#> [1] "+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000
+y_0=-100000 +ellps=airy +units=m +no_defs"
crs_lnd_new$epsg
#> [1] 27700
```

The result shows that the EPSG code 27700 represents the British National Grid, which could have been found by searching online for "EPSG 27700".

Printing a spatial object in the console automatically returns its coordinate reference system. To access and modify it explicitly, use the `st_crs` function, for example, `st_crs(cycle_hire_osm)`.

7.8 Reprojecting raster geometries

The projection concepts described in the previous section apply to rasters. However, there are important differences in reprojection of vectors and rasters: transforming a vector object involves changing the coordinates of every vertex,

but this does not apply to raster data. Rasters are composed of rectangular cells of the same size (expressed by map units, such as degrees or meters), so it is usually impracticable to transform coordinates of pixels separately. Thus, raster reprojection involves creating a new raster object, often with a different number of columns and rows than the original. The attributes must subsequently be re-estimated, allowing the new pixels to be 'filled' with appropriate values. In other words, raster reprojection can be thought of as two separate spatial operations: a vector reprojection of the raster extent to another CRS (Section 7.7), and computation of new pixel values through resampling (Section 5.3.4). Thus in most cases when both raster and vector data are used, it is better to avoid reprojecting rasters and to reproject vectors instead.

Reprojection of the regular rasters is also known as warping. Additionally, there is a second similar operation called "transformation". Instead of resampling all of the values, it leaves all values intact but recomputes new coordinates for every raster cell, changing the grid geometry. For example, it could convert the input raster (a regular grid) into a curvilinear grid. The transformation operation can be performed in R using the **stars** package.

The raster reprojection process is done with `project()` from the **terra** package. Like the `st_transform()` function demonstrated in the previous section, `project()` takes a spatial object (a raster dataset in this case) and some CRS representation as the second argument. On a side note, the second argument can also be an existing raster object with a different CRS.

Let's take a look at two examples of raster transformation: using categorical and continuous data. Land cover data are usually represented by categorical maps. The `nlcd.tif` file provides information for a small area in Utah, USA obtained from National Land Cover Database 2011 in the NAD83 / UTM zone 12N CRS, as shown in the output of the code chunk below (only first line of output shown).

```
cat_raster = rast(system.file("raster/nlcd.tif", package = "spDataLarge"))
crs(cat_raster)
#> PROJCRS["NAD83 / UTM zone 12N",
#> ...
```

In this region, eight land cover classes were distinguished (a full list of NLCD2011 land cover classes can be found at mrlc.gov):

```
unique(cat_raster)
#>       levels
#> 1      Water
```

TABLE 7.1 Key attributes in the original (cat_raster) and projected (cat_raster_wgs84) categorical raster datasets.

CRS	nrow	ncol	ncell	resolution	unique_categories
NAD83	1359	1073	1458207	31.5275	8
WGS84	1246	1244	1550024	0.0003	9

```
#> 2  Developed
#> 3     Barren
#> 4     Forest
#> 5  Shrubland
#> 6 Herbaceous
#> 7 Cultivated
#> 8   Wetlands
```

When reprojecting categorical rasters, the estimated values must be the same as those of the original. This could be done using the nearest neighbor method (`near`), which sets each new cell value to the value of the nearest cell (center) of the input raster. An example is reprojecting `cat_raster` to WGS84, a geographic CRS well suited for web mapping. The first step is to obtain the definition of this CRS. The second step is to reproject the raster with the `project()` function which, in the case of categorical data, uses the nearest neighbor method (`near`).

```
cat_raster_wgs84 = project(cat_raster, "EPSG:4326", method = "near")
```

Many properties of the new object differ from the previous one, including the number of columns and rows (and therefore number of cells), resolution (transformed from meters into degrees), and extent, as illustrated in Table 7.1 (note that the number of categories increases from 8 to 9 because of the addition of `NA` values, not because a new category has been created — the land cover classes are preserved).

Reprojecting numeric rasters (with `numeric` or in this case `integer` values) follows an almost identical procedure. This is demonstrated below with `srtm.tif` in **spDataLarge** from the Shuttle Radar Topography Mission (SRTM), which represents height in meters above sea level (elevation) with the WGS84 CRS:

```
con_raster = rast(system.file("raster/srtm.tif", package = "spDataLarge"))
cat(crs(con_raster))
#> GEOGCRS["WGS 84",
#>     ENSEMBLE["World Geodetic System 1984 ensemble",
#>         MEMBER["World Geodetic System 1984 (Transit)"],
```

TABLE 7.2 Key attributes in the original (con_raster) and projected (con_raster_ea) continuous raster datasets.

CRS	nrow	ncol	ncell	resolution	mean
WGS84	457	465	212505	0.0008	1843
UTM zone 12N	515	422	217330	83.5334	1842

```
#>         MEMBER["World Geodetic System 1984 (G730)"],
#>         MEMBER["World Geodetic System 1984 (G873)"],
#>         MEMBER["World Geodetic System 1984 (G1150)"],
....
```

We will reproject this dataset into a projected CRS, but *not* with the nearest neighbor method which is appropriate for categorical data. Instead, we will use the bilinear method which computes the output cell value based on the four nearest cells in the original raster.[10] The values in the projected dataset are the distance-weighted average of the values from these four cells: the closer the input cell is to the center of the output cell, the greater its weight. The following commands create a text string representing WGS 84 / UTM zone 12N, and reproject the raster into this CRS, using the `bilinear` method (output not shown).

```
con_raster_ea = project(con_raster, "EPSG:32612", method = "bilinear")
cat(crs(con_raster_ea))
```

Raster reprojection on numeric variables also leads to changes to values and spatial properties, such as the number of cells, resolution, and extent. These changes are demonstrated in Table 7.2.[11]

Of course, the limitations of 2D Earth projections apply as much to vector as to raster data. At best we can comply with two out of three spatial properties (distance, area, direction). Therefore, the task at hand determines which projection to choose. For instance, if we are interested in a density (points per grid cell or inhabitants per grid cell), we should use an equal-area projection (see also Chapter 14).

[10] Other methods mentioned in Section 5.3.4 also can be used here.

[11] Another minor change, which is not represented in Table 7.2, is that the class of the values in the new projected raster dataset is `numeric`. This is because the `bilinear` method works with continuous data and the results are rarely coerced into whole integer values. This can have implications for file sizes when raster datasets are saved.

7.9 Custom map projections

Established CRSs captured by `AUTHORITY:CODE` identifiers such as `EPSG:4326` are well suited for many applications. However, it is desirable to use alternative projections or to create custom CRSs in some cases. Section 7.6 mentioned reasons for using custom CRSs and provided several possible approaches. Here, we show how to apply these ideas in R.

One is to take an existing WKT definition of a CRS, modify some of its elements, and then use the new definition for reprojecting. This can be done for spatial vectors with `st_crs()` and `st_transform()`, and for spatial rasters with `crs()` and `project()`, as demonstrated in the following example which transforms the `zion` object to a custom azimuthal equidistant (AEQD) CRS.

```
zion = read_sf(system.file("vector/zion.gpkg", package = "spDataLarge"))
```

Using a custom AEQD CRS requires knowing the coordinates of the center point of a dataset in degrees (geographic CRS). In our case, this information can be extracted by calculating a centroid of the `zion` area and transforming it into WGS84.

```
zion_centr = st_centroid(zion)
zion_centr_wgs84 = st_transform(zion_centr, "EPSG:4326")
st_as_text(st_geometry(zion_centr_wgs84))
#> [1] "POINT (-113 37.3)"
```

Next, we can use the newly obtained values to update the WKT definition of the AEQD CRS seen below. Notice that we modified just two values below – `"Central_Meridian"` to the longitude and `"Latitude_Of_Origin"` to the latitude of our centroid.

```
my_wkt = 'PROJCS["Custom_AEQD",
 GEOGCS["GCS_WGS_1984",
  DATUM["WGS_1984",
   SPHEROID["WGS_1984",6378137.0,298.257223563]],
  PRIMEM["Greenwich",0.0],
  UNIT["Degree",0.0174532925199433]],
 PROJECTION["Azimuthal_Equidistant"],
 PARAMETER["Central_Meridian",-113.0263],
 PARAMETER["Latitude_Of_Origin",37.29818],
 UNIT["Meter",1.0]]'
```

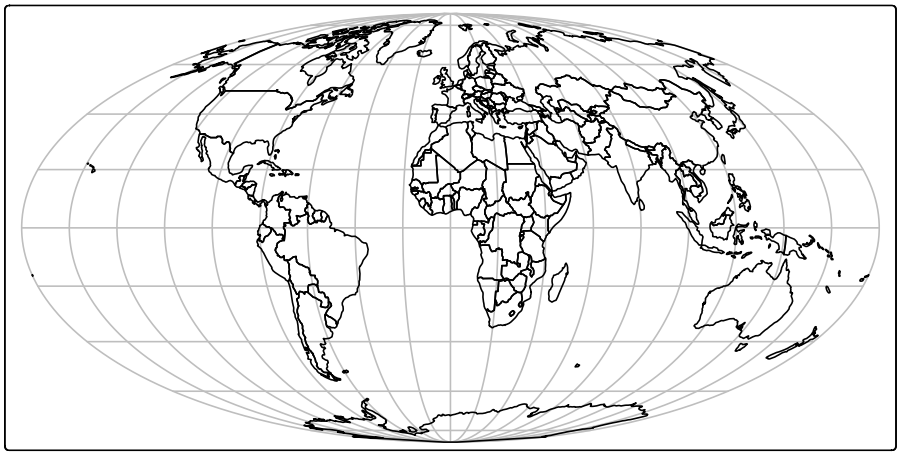

FIGURE 7.3 Mollweide projection of the world.

This approach's last step is to transform our original object (`zion`) to our new custom CRS (`zion_aeqd`).

```
zion_aeqd = st_transform(zion, my_wkt)
```

Custom projections can also be made interactively, for example, using the Projection Wizard web application (Šavrič et al., 2016). This website allows you to select a spatial extent of your data and a distortion property, and returns a list of possible projections. The list also contains WKT definitions of the projections that you can copy and use for reprojections. Also, see Open Geospatial Consortium (2019) for details on creating custom CRS definitions with WKT strings.

PROJ strings can also be used to create custom projections, accepting the limitations inherent to projections, especially of geometries covering large geographic areas, mentioned in Section 7.2. Many projections have been developed and can be set with the `+proj=` element of PROJ strings, with dozens of projects described in detail on the PROJ website alone.

When mapping the world while preserving area relationships, the Mollweide projection, illustrated in Figure 7.3, is a popular and often sensible choice (Jenny et al., 2017). To use this projection, we need to specify it using the proj-string element, `"+proj=moll"`, in the `st_transform` function:

```
world_mollweide = st_transform(world, crs = "+proj=moll")
```

It is often desirable to minimize distortion for all spatial properties (area, direction, distance) when mapping the world. One of the most popular

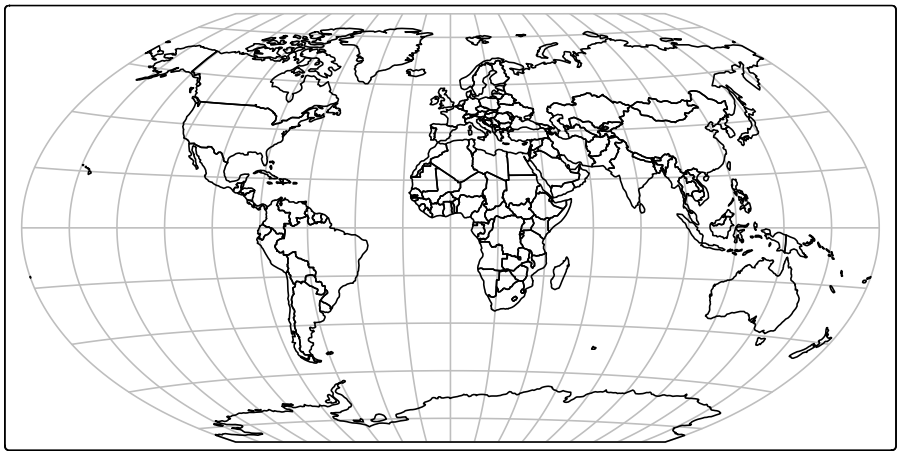

FIGURE 7.4 Winkel tripel projection of the world.

projections to achieve this is Winkel tripel, illustrated in Figure 7.4.[12] The result was created with the following command:

```
world_wintri = st_transform(world, crs = "+proj=wintri")
```

Moreover, proj-string parameters can be modified in most CRS definitions, for example the center of the projection can be adjusted using the `+lon_0` and `+lat_0` parameters. The below code transforms the coordinates to the Lambert azimuthal equal-area projection centered on the longitude and latitude of New York City (Figure 7.5).

```
world_laea2 = st_transform(world,
                           crs = "+proj=laea +x_0=0 +y_0=0 +lon_0=-74 +lat_0=40")
```

More information on CRS modifications can be found in the Using PROJ documentation.

7.10 Exercises

E1. Create a new object called `nz_wgs` by transforming `nz` object into the WGS84 CRS.

- Create an object of class `crs` for both and use this to query their CRSs.

[12] This projection is used, among others, by the National Geographic Society.

FIGURE 7.5 Lambert azimuthal equal-area projection of the world centered on New York City.

- With reference to the bounding box of each object, what units does each CRS use?
- Remove the CRS from `nz_wgs` and plot the result: what is wrong with this map of New Zealand and why?

E2. Transform the `world` dataset to the transverse Mercator projection (`"+proj=tmerc"`) and plot the result. What has changed and why? Try to transform it back into WGS 84 and plot the new object. Why does the new object differ from the original one?

E3. Transform the continuous raster (`con_raster`) into NAD83 / UTM zone 12N using the nearest neighbor interpolation method. What has changed? How does it influence the results?

E4. Transform the categorical raster (`cat_raster`) into WGS 84 using the bilinear interpolation method. What has changed? How does it influence the results?

8

Geographic data I/O

Prerequisites

This chapter requires the following packages:

```
library(sf)
library(terra)
library(dplyr)
library(spData)
```

8.1 Introduction

This chapter is about reading and writing geographic data. Geographic data *input* is essential for geocomputation: real-world applications are impossible without data. Data *output* is also vital, enabling others to use valuable new or improved datasets resulting from your work. Taken together, these processes of input/output can be referred to as data I/O. Geographic data I/O is often done in haste at the beginning and end of projects and otherwise ignored. However, data import and export are fundamental to the success or otherwise of projects: small I/O mistakes made at the beginning of projects (e.g., using an out-of-date dataset) can lead to large problems down the line.

There are many geographic file formats, each with its own advantages and disadvantages, as described in Section 8.2. Reading and writing from and to these file formats is covered in Sections 8.3 and 8.4, respectively. In terms of where to *find* data, Section 8.5 describes *geoportals* and how to import data from them. Dedicated packages to ease geographic data import, from sources including OpenStreetMap, are described in Section 8.6. If you want to put your data 'into production' in web services (or if you want to be sure that your data adheres to established standards), geographic metadata is important, as described in Section 8.7. Another possibility to obtain spatial data is to use web services, outlined in Section 8.8. The final Section 8.9 demonstrates

methods for saving visual outputs (maps), in preparation for Chapter 9 on visualization.

8.2 File formats

Geographic datasets are usually stored as files or in spatial databases. File formats can either store vector or raster data, while spatial databases such as PostGIS can store both (see also Section 10.7). Today the variety of file formats may seem bewildering, but there has been much consolidation and standardization since the beginnings of GIS software in the 1960s when the first widely distributed program (SYMAP) for spatial analysis was created at Harvard University (Coppock and Rhind, 1991).

GDAL (which should be pronounced "goo-dal", with the double "o" making a reference to object-orientation), Geospatial Data Abstraction Library, has resolved many issues associated with incompatibility between geographic file formats since its release in 2000. GDAL provides a unified and high-performance interface for reading and writing many raster and vector data formats.[1] Many open and proprietary GIS programs, including GRASS GIS, ArcGIS and QGIS, use GDAL behind their GUIs for doing the legwork of ingesting and spitting out geographic data in appropriate formats.

GDAL provides access to more than 200 vector and raster data formats. Table 8.1 presents some basic information about selected and often used spatial file formats.

An important development ensuring the standardization and open-sourcing of file formats was the founding of the Open Geospatial Consortium (OGC) in 1994. Beyond defining the simple features data model (see Section 2.2.1), the OGC also coordinates the development of open standards, for example as used in file formats such as GML, KML and GeoPackage. Open file formats of the kind endorsed by the OGC have several advantages over proprietary formats: the standards are published, ensure transparency and open up the possibility for users to further develop and adjust the file formats to their specific needs.

ESRI Shapefile is the most popular vector data exchange format; however, it is not an open format (though its specification is open). It was developed in the early 1990s and has a number of limitations. First of all, it is a multi-file format, which consists of at least three files. It only supports 255 columns, column names are restricted to ten characters and the file size limit is 2 GB. Furthermore, ESRI Shapefile does not support all possible geometry types, for

[1] As we mentioned in Chapter 5, GDAL also contains a set of utility functions allowing for raster mosaicing, resampling, cropping, and reprojecting, etc.

TABLE 8.1 Selected spatial file formats.

Name	Extension	Information	Type	Model
ESRI Shapefile	.shp (the main file)	Popular format consisting of at least three files. No support for: files > 2 GB; mixed types; names > 10 chars; cols > 255.	Vector	Partially open
GeoJSON	.geojson	Extends the JSON exchange format by including a subset of the simple feature representation; mostly used for storing coordinates in longitude and latitude; it is extended by the TopoJSON format.	Vector	Open
KML	.kml	XML-based format for spatial visualization, developed for use with Google Earth. Zipped KML file forms the KMZ format.	Vector	Open
GPX	.gpx	XML schema created for exchange of GPS data.	Vector	Open
FlatGeobuf	.fgb	Single file format allowing for quick reading and writing of vector data. Has streaming capabilities.	Vector	Open
GeoTIFF	.tif/.tiff	Popular raster format. A TIFF file containing additional spatial metadata.	Raster	Open
Arc ASCII	.asc	Text format where the first six lines represent the raster header, followed by the raster cell values arranged in rows and columns.	Raster	Open
SQLite/SpatiaLite	.sqlite	Standalone relational database, SpatiaLite is the spatial extension of SQLite.	Vector and raster	Open
ESRI FileGDB	.gdb	Spatial and non-spatial objects created by ArcGIS. Allows multiple feature classes; topology. Limited support from GDAL.	Vector and raster	Proprietary
GeoPackage	.gpkg	Lightweight database container based on SQLite allowing an easy and platform-independent exchange of geodata.	Vector and (very limited) raster	Open

example, it is unable to distinguish between a polygon and a multipolygon.[2] Despite these limitations, a viable alternative had been missing for a long time. In recent years, GeoPackage emerged, and seems to be a more than suitable replacement candidate for ESRI Shapefile. Geopackage is a format for exchanging geospatial information and an OGC standard. The GeoPackage standard describes the rules on how to store geospatial information in a tiny SQLite container. Hence, GeoPackage is a lightweight spatial database container, which allows the storage of vector and raster data but also of non-spatial data and extensions. Aside from GeoPackage, there are other geospatial data exchange formats worth checking out (Table 8.1).

The GeoTIFF format seems to be the most prominent raster data format. It allows spatial information, such as CRS, to be embedded within a TIFF file. Similar to ESRI Shapefile, this format was first developed in the 1990s, but as an open format. Additionally, GeoTIFF is still being expanded and improved. One of the most significant recent additions to the GeoTIFF format is its variant called COG (*Cloud Optimized GeoTIFF*). Raster objects saved as COGs can be hosted on HTTP servers, so other people can read only parts of the file without downloading the whole file (see Sections 8.3.2 and 8.4.2).

There are many geographic file formats beyond those shown in Table 8.1 and new data formats capable of representing geographic are being developed. Recent examples are formats based on the GeoArrow and Zarr specifications. GDAL's documentation provides a good resource for learning about other vector and raster drivers. Furthermore, some data formats can store other data models (types) beyond the vector and raster data models introduced in Section 2.2.1. It includes LAS and LAZ formats for storing lidar point clouds, and NetCDF and HDF for storing multidimensional arrays.

Spatial data is also often stored using tabular (non-spatial) text formats, including CSV files or Excel spreadsheets. For example, this can be convenient to share spatial samples with people who do not use GIS tools or exchange data with other software that does not accept spatial data formats. However, this approach has downsides: it is challenging for storing geometries that are more complex than POINTs and omits important spatial metadata such as the CRS.

8.3 Data input (I)

Executing commands such as `sf::read_sf()` (the main function we use for loading vector data) or `terra::rast()` (the main function used for loading raster

[2] To learn more about ESRI Shapefile limitations and possible alternative file formats, visit http://switchfromshapefile.org/.

TABLE 8.2 Popular drivers/formats for reading/writing vector data.

name	long_name	write	copy	is_raster	is_vector	vsi
ESRI Shapefile	ESRI Shapefile	TRUE	FALSE	FALSE	TRUE	TRUE
GPX	GPX	TRUE	FALSE	FALSE	TRUE	TRUE
KML	Keyhole Markup Language (KML)	TRUE	FALSE	FALSE	TRUE	TRUE
GeoJSON	GeoJSON	TRUE	FALSE	FALSE	TRUE	TRUE
GPKG	GeoPackage	TRUE	TRUE	TRUE	TRUE	TRUE
FlatGeobuf	FlatGeobuf	TRUE	FALSE	FALSE	TRUE	TRUE

data) silently sets off a chain of events that reads data from files. Many R packages provide example datasets (e.g., the dataset `spData::world` that we used in earlier chapters) and functions to get geographic datasets from a range of data sources. All of them load the data into R or, more precisely, assign objects to your workspace. This means that when objects are imported into R they are stored in RAM[3], can be listed with `ls()` (and should be viewable in 'Environment' panels in your development environement) and can be accessed from the `.GlobalEnv` of the R session.

8.3.1 Vector data

Spatial vector data comes in a wide variety of file formats. Most popular representations such as `.geojson` and `.gpkg` files can be imported directly into R with the **sf** function `read_sf()` (or the equivalent `st_read()`), which uses GDAL's vector drivers behind the scenes. The `st_drivers()` function returns a data frame containing `name` and `long_name` in the first two columns, and features of each driver available to GDAL (and therefore **sf**), including ability to write data and store raster data in the subsequent columns, as illustrated for key file formats in Table 8.2.

The following commands show the first three drivers reported the computer's GDAL installation (results can vary depending on the GDAL version installed) and a summary of their features. Note that the majority of drivers can write data, while only a dozen or so formats can efficiently represent raster data in addition to vector data (see `?st_drivers()` for details):

[3]There are some exceptions to this rule, e.g., **terra**'s `SpatRaster` objects (which are C++ pointers to the actual data) and objects that represent database connections that can be imported into memory with functions such as `dplyr::collect()`, as outlined in Section 10.7.

```
sf_drivers = st_drivers()
head(sf_drivers, n = 3)
summary(sf_drivers[-c(1:2)])
```

The first argument of `read_sf()` is `dsn`, which should be a text string or an object containing a single text string. The content of a text string could vary between different drivers. In most cases, as with the ESRI Shapefile (`.shp`) or the `GeoPackage` format (`.gpkg`), the `dsn` would be a file name. `read_sf()` guesses the driver based on the file extension, as illustrated for a `.gpkg` file below:

```
f = system.file("shapes/world.gpkg", package = "spData")
world = read_sf(f)
```

For some drivers, `dsn` could be provided as a folder name, access credentials for a database, or a GeoJSON string representation (see the examples of the `read_sf()` help page for more details).

Some vector driver formats can store multiple data layers. By default, `read_sf()` automatically reads the first layer of the file specified in `dsn`; however, using the `layer` argument you can specify any other layer.

The `read_sf()` function also allows for reading just parts of the file into RAM with two possible mechanisms. The first one is related to the `query` argument, which allows specifying what part of the data to read with the OGR SQL query text. An example below extracts data for Tanzania only (Figure 8.1A). It is done by specifying that we want to get all columns (`SELECT *`) from the `"world"` layer for which the `name_long` is equal to `"Tanzania"`:

```
tanzania = read_sf(f, query = 'SELECT * FROM world WHERE name_long = "Tanzania"')
```

If you do not know the names of the available columns, a good approach is to just read one row of the data with `'SELECT * FROM world WHERE FID = 1'`. `FID` represents a *feature ID* – most often, it is a row number; however, its values depend on the used file format. For example, `FID` starts from 0 in ESRI Shapefile, from 1 in some other file formats, or can be even arbitrary.

The second mechanism uses the `wkt_filter` argument. This argument expects a well-known text representing a study area for which we want to extract the data. Let's try it using a small example – we want to read polygons from our file that intersect with the buffer of 50,000 meters of Tanzania's borders. To do it, we need to prepare our "filter" by (a) creating the buffer (Section 5.2.3), (b) converting the `sf` buffer object into an `sfc` geometry object with `st_geometry()`, and (c) translating geometries into their well-known text representation with `st_as_text()`:

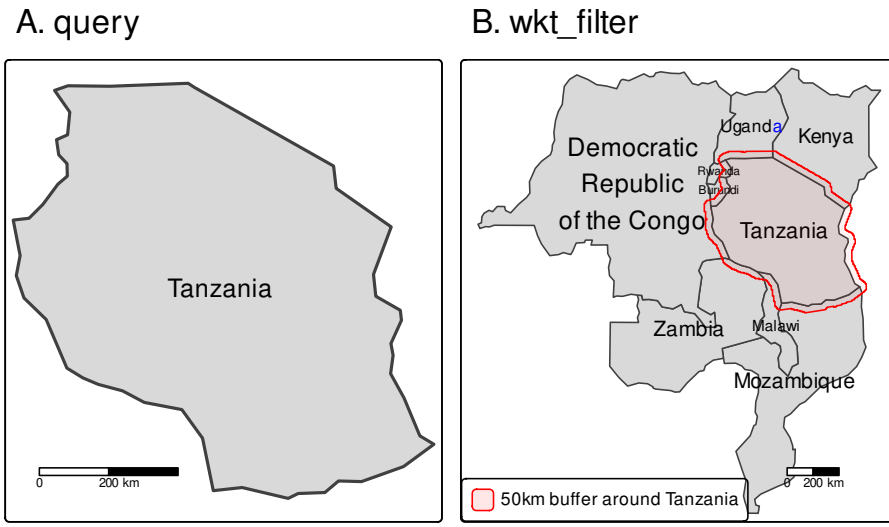

FIGURE 8.1 Reading a subset of the vector data using (A) a query and (B) a wkt filter.

```
tanzania_buf = st_buffer(tanzania, 50000)
tanzania_buf_geom = st_geometry(tanzania_buf)
tanzania_buf_wkt = st_as_text(tanzania_buf_geom)
```

Now, we can apply this "filter" using the `wkt_filter` argument.

```
tanzania_neigh = read_sf(f, wkt_filter = tanzania_buf_wkt)
```

Our result, shown in Figure 8.1(B), contains Tanzania and every country within its 50-km buffer.

Naturally, some options are specific to certain drivers.[4] For example, think of coordinates stored in a spreadsheet format (`.csv`). To read-in such files as spatial objects, we naturally have to specify the names of the columns (`X` and `Y` in our example below) representing the coordinates. We can do this with the help of the `options` parameter. To find out about possible options, please refer to the 'Open Options' section of the corresponding GDAL driver description. For the comma-separated value (csv) format, visit https://gdal.org/drv_csv.html.

[4] A list of supported vector formats and options can be found at https://gdal.org/ogr_formats.html.

```
cycle_hire_txt = system.file("misc/cycle_hire_xy.csv", package = "spData")
cycle_hire_xy = read_sf(cycle_hire_txt,
  options = c("X_POSSIBLE_NAMES=X", "Y_POSSIBLE_NAMES=Y"))
```

Instead of columns describing 'XY' coordinates, a single column can also contain the geometry information. Well-known text (WKT), well-known binary (WKB), and the GeoJSON formats are examples of this. For instance, the `world_wkt.csv` file has a column named `WKT` representing polygons of the world's countries. We will again use the `options` parameter to indicate this.

```
world_txt = system.file("misc/world_wkt.csv", package = "spData")
world_wkt = read_sf(world_txt, options = "GEOM_POSSIBLE_NAMES=WKT")
```

Not all of the supported vector file formats store information about their coordinate reference system. In these situations, it is possible to add the missing information using the `st_set_crs()` function. Please refer also to Section 7.3 for more information.

As a final example, we will show how `read_sf()` also reads KML files. A KML file stores geographic information in XML format, which is a data format for the creation of web pages and the transfer of data in an application-independent way (Nolan and Lang, 2014). Here, we access a KML file from the web. This file contains more than one layer. `st_layers()` lists all available layers. We choose the first layer `Placemarks` and say so with the help of the `layer` parameter in `read_sf()`.

```
u = "https://developers.google.com/kml/documentation/KML_Samples.kml"
download.file(u, "KML_Samples.kml")
st_layers("KML_Samples.kml")
#> Driver: LIBKML
#> Available layers:
#>               layer_name geometry_type features fields crs_name
#> 1             Placemarks                       3     11   WGS 84
#> 2      Styles and Markup                       1     11   WGS 84
#> 3       Highlighted Icon                       1     11   WGS 84
....
kml = read_sf("KML_Samples.kml", layer = "Placemarks")
```

All the examples presented in this section so far have used the **sf** package for geographic data import. It is fast and flexible, but it may be worth looking at other packages such as **duckdb**, an R interface to the DuckDB database system which has a spatial extension.

8.3.2 Raster data

Similar to vector data, raster data comes in many file formats with some supporting multi-layer files. **terra**'s `rast()` command reads in a single layer when a file with just one layer is provided.

```
raster_filepath = system.file("raster/srtm.tif", package = "spDataLarge")
single_layer = rast(raster_filepath)
```

It also works in case you want to read a multi-layer file.

```
multilayer_filepath = system.file("raster/landsat.tif", package = "spDataLarge")
multilayer_rast = rast(multilayer_filepath)
```

All of the previous examples read spatial information from files stored on your hard drive. However, GDAL also allows reading data directly from online resources, such as HTTP/HTTPS/FTP web resources. The only thing we need to do is to add a `/vsicurl/` prefix before the path to the file. Let's try it by connecting to the global monthly snow probability at 500-m resolution for the period 2000-2012. Snow probability for December is stored as a Cloud Optimized GeoTIFF (COG) file (see Section 8.2) at zenodo.org. To read an online file, we just need to provide its URL together with the `/vsicurl/` prefix.

```
myurl = paste0("/vsicurl/https://zenodo.org/record/5774954/files/",
               "clm_snow.prob_esacci.dec_p.90_500m_s0..0cm_2000..2012_v2.0.tif")
snow = rast(myurl)
snow
#> class       : SpatRaster
#> dimensions  : 35849, 86400, 1  (nrow, ncol, nlyr)
#> resolution  : 0.00417, 0.00417  (x, y)
#> extent      : -180, 180, -62, 87.4  (xmin, xmax, ymin, ymax)
#> coord. ref. : lon/lat WGS 84 (EPSG:4326)
#> source      : clm_snow.prob_esacci.dec_p.90_500m_s0..0cm_2000..2012_v2.0.tif
#> name        : clm_snow.prob_esacci.dec_p.90_500m_s0..0cm_2000..2012_v2.0
```

Due to the fact that the input data is COG, we are actually not reading this file to our RAM, but rather creating a connection to it without obtaining any values. Its values will be read if we apply any value-based operation (e.g., `crop()` or `extract()`). This allows us also to just read a tiny portion of the data without downloading the entire file. For example, we can get the snow probability for December in Reykjavik (70%) by specifying its coordinates and applying the `extract()` function:

```
rey = data.frame(lon = -21.94, lat = 64.15)
snow_rey = extract(snow, rey)
snow_rey
#>   ID clm_snow.prob_esacci.dec_p.90_500m_s0..0cm_2000..2012_v2.0
#> 1  1                                                         70
```

This way, we just downloaded a single value instead of the whole, large GeoTIFF file. The above example just shows one simple (but useful) case, but there is more to explore. The `/vsicurl/` prefix also works not only for raster but also for vector file formats. It allows reading vectors directly from online storage with `read_sf()` just by adding the prefix before the vector file URL.

Importantly, `/vsicurl/` is not the only prefix provided by GDAL – many more exist, such as `/vsizip/` to read spatial files from ZIP archives without decompressing them beforehand or `/vsis3/` for on-the-fly reading files available in AWS S3 buckets. You can learn more about it at https://gdal.org/user/virtual_file_systems.html.

As with vector data, raster datasets can also be stored in and read from some spatial databases, notably PostGIS. See Section 10.7 for more details.

8.4 Data output (O)

Writing geographic data allows you to convert from one format to another and to save newly created objects. Depending on the data type (vector or raster), object class (e.g., `sf` or `SpatRaster`), and type and amount of stored information (e.g., object size, range of values), it is important to know how to store spatial files in the most efficient way. The next two sections will demonstrate how to do this.

8.4.1 Vector data

The counterpart of `read_sf()` is `write_sf()`. It allows you to write **sf** objects to a wide range of geographic vector file formats, including the most common such as `.geojson`, `.shp` and `.gpkg`. Based on the file name, `write_sf()` decides automatically which driver to use. The speed of the writing process depends also on the driver.

```
write_sf(obj = world, dsn = "world.gpkg")
```

Note: if you try to write to the same data source again, the function will overwrite the file:

```
write_sf(obj = world, dsn = "world.gpkg")
```

Instead of overwriting the file, we could add a new layer to the file by specifying the `layer` argument. This is supported by several spatial formats, including GeoPackage.

```
write_sf(obj = world, dsn = "world_many_layers.gpkg", layer = "second_layer")
```

Alternatively, you can use `st_write()` since it is equivalent to `write_sf()`. However, it has different defaults – it does not overwrite files (returns an error when you try to do it) and shows a short summary of the written file format and the object.

```
st_write(obj = world, dsn = "world2.gpkg")
#> Writing layer `world2' to data source `world2.gpkg' using driver `GPKG'
#> Writing 177 features with 10 fields and geometry type Multi Polygon.
```

The `layer_options` argument could be also used for many different purposes. One of them is to write spatial data to a text file. This can be done by specifying `GEOMETRY` inside of `layer_options`. It could be either `AS_XY` for simple point datasets (it creates two new columns for coordinates) or `AS_WKT` for more complex spatial data (one new column is created which contains the well-known text representation of spatial objects).

```
write_sf(cycle_hire_xy, "cycle_hire_xy.csv", layer_options = "GEOMETRY=AS_XY")
write_sf(world_wkt, "world_wkt.csv", layer_options = "GEOMETRY=AS_WKT")
```

8.4.2 Raster data

The `writeRaster()` function saves `SpatRaster` objects to files on disk. The function expects input regarding output data type and file format, but also accepts GDAL options specific to a selected file format (see `?writeRaster` for more details).

The **terra** package offers seven data types when saving a raster: INT1U, INT2S, INT2U, INT4S, INT4U, FLT4S, and FLT8S,[5] which determine the bit representation of the raster object written to disk (Table 8.3). Which data type to use depends on the range of the values of your raster object. The more values a data type can represent, the larger the file will get on disk. Unsigned integers (INT1U, INT2U, INT4U) are suitable for categorical data, while float numbers (FLT4S and FLT8S) usually represent continuous data.

[5] Using INT4U is not recommended, as R does not support 32-bit unsigned integers.

TABLE 8.3 Data types supported by the terra package.

Data Type	Minimum Value	Maximum Value
INT1U	0	255
INT2S	–32,767	32,767
INT2U	0	65,534
INT4S	–2,147,483,647	2,147,483,647
INT4U	0	4,294,967,296
FLT4S	–3.4e+38	3.4e+38
FLT8S	–1.7e+308	1.7e+308

`writeRaster()` uses FLT4S as the default. While this works in most cases, the size of the output file will be unnecessarily large if you save binary or categorical data. Therefore, we would recommend to use the data type that needs the least storage space but is still able to represent all values (check the range of values with the `summary()` function).

By default, the output file format is derived from the filename. Naming a file `*.tif` will create a GeoTIFF file, as demonstrated below:

```
writeRaster(single_layer, filename = "my_raster.tif", datatype = "INT2U")
```

Some raster file formats have additional options that can be set by providing GDAL parameters to the `options` argument of `writeRaster()`. GeoTIFF files are written in **terra**, by default, with the LZW compression `gdal = c("COMPRESS=LZW")`. To change or disable the compression, we need to modify this argument.

```
writeRaster(x = single_layer, filename = "my_raster.tif",
            gdal = c("COMPRESS=NONE"), overwrite = TRUE)
```

Additionally, we can save our raster object as COG (Cloud Optimized GeoTIFF, Section 8.2) with the `filetype = "COG"` options.

```
writeRaster(x = single_layer, filename = "my_raster.tif",
            filetype = "COG", overwrite = TRUE)
```

To learn more about the compression of GeoTIFF files, we recommend Paul Ramsey's comprehensive blog post, GeoTiff Compression for Dummies which can be found online.

8.5 Geoportals

A vast and ever-increasing amount of geographic data is available on the internet, much of which is free to access and use (with appropriate credit given to its providers).[6] In some ways there is now *too much* data, in the sense that there are often multiple places to access the same dataset. Some datasets are of poor quality. In this context, it is vital to know where to look, so the first section covers some of the most important sources. Various 'geoportals' (web services providing geospatial datasets such as Data.gov) are a good place to start, providing a wide range of data but often only for specific locations (as illustrated in the updated Wikipedia page on the topic).

Some global geoportals overcome this issue. The GEOSS portal and the Copernicus Data Space Ecosystem, for example, contain many raster datasets with global coverage. A wealth of vector datasets can be accessed from the SEDAC portal run by the National Aeronautics and Space Administration (NASA) and the European Union's INSPIRE geoportal, with global and regional coverage.

Most geoportals provide a graphical interface allowing datasets to be queried based on characteristics such as spatial and temporal extent, the United States Geological Survey's EarthExplorer being a prime example. *Exploring* datasets interactively on a browser is an effective way of understanding available layers. *Downloading* data is best done with code, however, from reproducibility and efficiency perspectives. Downloads can be initiated from the command line using a variety of techniques, primarily via URLs and APIs (see the Copernicus APIs for example).[7] Files hosted on static URLs can be downloaded with `download.file()`, as illustrated in the code chunk below which accesses PeRL: Permafrost Region Pond and Lake Database from pangaea.de:

```
download.file(url = "https://hs.pangaea.de/Maps/PeRL/
                     PeRL_permafrost_landscapes.zip",
              destfile = "PeRL_permafrost_landscapes.zip",
              mode = "wb")
unzip("PeRL_permafrost_landscapes.zip")
canada_perma_land = read_sf("PeRL_permafrost_landscapes/canada_perma_land.shp")
```

[6] For example, visit freegisdata.rtwilson.com for a long list of websites with freely available geographic datasets.

[7] An example of using a STAC-API to download Sentinel-2 data is provided in Section 10.8.1.

TABLE 8.4 Selected R packages for geographic data retrieval.

Package	Description
climateR	Access over 100,000 gridded climate and landscape datasets from over 2,000 data providers by area of interest.
elevatr	Access point and raster elevation data from various sources.
FedData	Datasets maintained by the US federal government, including elevation and land cover.
geodata	Download and import imports administrative, elevation, WorldClim data.
osmdata	Download and import small OpenStreetMap datasets.
osmextract	Download and import large OpenStreetMap datasets.
rnaturalearth	Access Natural Earth vector and raster data.
rnoaa	Import National Oceanic and Atmospheric Administration (NOAA) climate data.

8.6 Geographic data packages

Many R packages have been developed for accessing geographic data, some of which are presented in Table 8.4. These provide interfaces to one or more spatial libraries or geoportals and aim to make data access even quicker from the command line.

It should be emphasized that Table 8.4 represents only a small number of available geographic data packages. For example, a large number of R packages exist to obtain various socio-demographic data, such as **tidycensus** and **tigris** (USA), **cancensus** (Canada), **eurostat** and **giscoR** (European Union), or **idbr** (international databases) – read Analyzing US Census Data (Walker, 2022b) to find some examples of how to analyze such data. Similarly, several R packages exist giving access to spatial data for various regions and countries, such as **bcdata** (Province of British Columbia), **geobr** (Brazil), **RCzechia** (Czech Republic), or **rgugik** (Poland).

Each data package has its own syntax for accessing data. This diversity is demonstrated in the subsequent code chunks, which show how to get data using three packages from Table 8.4.[8] Country borders are often useful and

[8] More examples of data downloading using dedicated R packages can be found at https://rspatialdata.github.io/.

these can be accessed with the `ne_countries()` function from the **rnaturalearth** package (Massicotte and South, 2023) as follows:

```
library(rnaturalearth)
usa_sf = ne_countries(country = "United States of America", returnclass = "sf")
```

Country borders can be also accessed with other packages, such as **geodata**, **giscoR**, or **rgeoboundaries**.

A second example downloads a series of rasters containing global monthly precipitation sums with spatial resolution of 10 minutes (~18.5 km at the equator) using the **geodata** package (Hijmans, 2023a). The result is a multi-layer object of class `SpatRaster`.

```
library(geodata)
worldclim_prec = worldclim_global("prec", res = 10, path = tempdir())
class(worldclim_prec)
```

A third example uses the **osmdata** package (Padgham et al., 2023) to find parks from the OpenStreetMap (OSM) database. As illustrated in the code-chunk below, queries begin with the function `opq()` (short for OpenStreetMap query), the first argument of which is a bounding box, or text string representing a bounding box (the city of Leeds, in this case). The result is passed to a function for selecting which OSM elements we're interested in (parks in this case), represented by *key-value pairs*. Next, they are passed to the function `osmdata_sf()` which does the work of downloading the data and converting it into a list of `sf` objects (see `vignette('osmdata')` for further details):

```
library(osmdata)
parks = opq(bbox = "leeds uk") |>
  add_osm_feature(key = "leisure", value = "park") |>
  osmdata_sf()
```

A limitation with the **osmdata** package is that it is *rate-limited*, meaning that it cannot download large OSM datasets (e.g., all the OSM data for a large city). To overcome this limitation, the **osmextract** package was developed, which can be used to download and import binary `.pbf` files containing compressed versions of the OSM database for predefined regions.

OpenStreetMap is a vast global database of crowd-sourced data, is growing daily, and has a wider ecosystem of tools enabling easy access to the data, from the Overpass turbo web service for rapid development and testing of OSM queries to osm2pgsql for importing the data into a PostGIS database. Although the quality of datasets derived from OSM varies, the data source and wider OSM ecosystems have many advantages: they provide datasets that

are available globally, free of charge, and constantly improving thanks to an army of volunteers. Using OSM encourages 'citizen science' and contributions back to the digital commons (you can start editing data representing a part of the world you know well at www.openstreetmap.org). Further examples of OSM data in action are provided in Chapters 10, 13 and 14.

Sometimes, packages come with built-in datasets. These can be accessed in four ways: by attaching the package (if the package uses 'lazy loading' as **spData** does), with `data(dataset, package = mypackage)`, by referring to the dataset with `mypackage::dataset`, or with `system.file(filepath, package = mypackage)` to access raw data files. The following code chunk illustrates the latter two options using the `world` dataset (already loaded by attaching its parent package with `library(spData)`):[9]

```
world2 = spData::world
world3 = read_sf(system.file("shapes/world.gpkg", package = "spData"))
```

The last example, `system.file("shapes/world.gpkg", package = "spData")`, returns a path to the `world.gpkg` file, which is stored inside of the `"shapes/"` folder of the **spData** package.

Another way to obtain spatial information is to perform geocoding – transform a description of a location, usually an address, into its coordinates. This is usually done by sending a query to an online service and getting the location as a result. Many such services exist that differ in the used method of geocoding, usage limitations, costs, or application programming interface (API) key requirements. R has several packages for geocoding; however, **tidygeocoder** seems to allow to connect to the largest number of geocoding services with a consistent interface. The **tidygeocoder** main function is `geocode`, which takes a data frame with addresses and adds coordinates as `"lat"` and `"long"`. This function also allows to select a geocoding service with the `method` argument and has many additional parameters.

Let's try this package by searching for coordinates of the John Snow blue plaque located on a building in the Soho district of London.

```
library(tidygeocoder)
geo_df = data.frame(address = "54 Frith St, London W1D 4SJ, UK")
geo_df = geocode(geo_df, address, method = "osm")
geo_df
```

[9]For more information on data import with R packages, see Sections 5.5 and 5.6 of Gillespie and Lovelace (2016).

The resulting data frame can be converted into an `sf` object with `st_as_sf()`.

```
geo_sf = st_as_sf(geo_df, coords = c("long", "lat"), crs = "EPSG:4326")
```

tidygeocoder also allows performing the opposite process called reverse geocoding used to get a set of information (name, address, etc.) based on a pair of coordinates. Geographic data can also be imported into R from various 'bridges' to geographic software, as described in Chapter 10.

8.7 Geographic metadata

Geographic metadata are a cornerstone of geographic information management, used to describe datasets, data structures and services. They help make datasets FAIR (Findable, Accessible, Interoperable, Reusable) and are defined by the ISO/OGC standards, in particular the ISO 19115 standard and underlying schemas. These standards are widely used within spatial data infrastructures, handled through metadata catalogs.

Geographic metadata can be managed with **geometa**, a package that allows writing, reading and validating geographic metadata according to the ISO/OGC standards. It already supports various international standards for geographic metadata information, such as the ISO 19110 (feature catalogue), ISO 19115-1 and 19115-2 (geographic metadata for vector and gridded/imagery datasets), ISO 19119 (geographic metadata for service), and ISO 19136 (Geographic Markup Language) providing methods to read, validate and write geographic metadata from R using the ISO/TS 19139 (XML) technical specification. Geographic metadata can be created with **geometa** as follows, which creates and saves a metadata file:

```
library(geometa)
# create a metadata
md = ISOMetadata$new()
#... fill the metadata 'md' object
# validate metadata
md$validate()
# XML representation of the ISOMetadata
xml = md$encode()
# save metadata
md$save("my_metadata.xml")
# read a metadata from an XML file
md = readISO19139("my_metadata.xml")
```

The package comes with more examples and has been extended by packages such as **geoflow** to ease and automate the management of metadata.

In the field of standard geographic information management, the distinction between data and metadata is less clear. The Geography Markup Language (GML) standard and file format covers both data and metadata, for example. The **geometa** package allows exporting GML (ISO 19136) objects from geometry objects modeled with **sf**. Such functionality allows use of geographic metadata (enabling the inclusion of metadata on detailed geographic and temporal extents, rather than simple bounding boxes, for example) and the provision of services that extend the GML standard (e.g., Open Geospatial Consortium Web Coverage Service, OGC-WCS).

8.8 Geographic web services

In an effort to standardize web APIs for accessing spatial data, the Open Geospatial Consortium (OGC) has created a number of standard specifications for web services (collectively known as OWS, which is short for OGC Web Services). These services complement and use core standards developed to model geographic information, such as the ISO/OGC Spatial Schema (ISO 19107:2019), or the Simple Features (ISO 19125-1:2004), and to format data, such as with the Geographic Markup Language (GML). These specifications cover common access services for data and metadata. Vector data can be accessed with the Web Feature Service (WFS), whereas grid/imagery can be accessed with the Web Coverage Service (WCS). Map image representations, such as tiles, can be accessed with the Web Map Service (WMS) or the Web Map Tile Service (WMTS). Metadata is also covered by means of the Catalogue Service for the Web (CSW). Finally, standard processing is handled through the Web Processing Service (WPS) or the the Web Coverage Processing Service (WCPS).

Various open-source projects have adopted these protocols, such as GeoServer and MapServer for data-handling, or GeoNetwork and PyCSW for metadata-handling, leading to standardization of queries. Integrated tools for Spatial Data Infrastructures (SDI), such as GeoNode, GeOrchestra or Examind have also adopted these standard webservices, either directly or by using previously mentioned open-source tools.

Like other web APIs, OWS APIs use a 'base URL', an 'endpoint' and 'URL query arguments' following a `?` to request data (see the `best-practices-api-packages` vignette in the **httr** package).

There are many requests that can be made to a OWS service. Below examples illustrate how some requests can be made directly with **httr** or more straightforward with the **ows4R** package (OGC Web-Services for R).

Let's start with examples using the **httr** package, which can be useful for understanding how web services work. One of the most fundamental requests is `getCapabilities`, demonstrated with **httr** functions `GET()` and `modify_url()` below. The following code chunk demonstrates how API queries can be constructed and dispatched, in this case to discover the capabilities of a service run by the Fisheries and Aquaculture Division of the Food and Agriculture Organization of the United Nations (UN-FAO).

```
library(httr)
base_url = "https://www.fao.org"
endpoint = "/fishery/geoserver/wfs"
q = list(request = "GetCapabilities")
res = GET(url = modify_url(base_url, path = endpoint), query = q)
res$url
#> [1] "https://www.fao.org/fishery/geoserver/wfs?request=GetCapabilities"
```

The above code chunk demonstrates how API requests can be constructed programmatically with the `GET()` function, which takes a base URL and a list of query parameters that can easily be extended. The result of the request is saved in `res`, an object of class `response` defined in the **httr** package, which is a list containing information of the request, including the URL. As can be seen by executing `browseURL(res$url)`, the results can also be read directly in a browser. One way of extracting the contents of the request is as follows:

```
txt = content(res, "text")
xml = xml2::read_xml(txt)
xml
#> {xml_document} ...
#> [1] <ows:ServiceIdentification>\n  <ows:Title>GeoServer WFS...
#> [2] <ows:ServiceProvider>\n  <ows:ProviderName>UN-FAO Fishe...
#> ...
```

Data can be downloaded from WFS services with the `GetFeature` request and a specific `typeName` (as illustrated in the code chunk below).

Available names differ depending on the accessed web feature service. One can extract them programmatically using web technologies (Nolan and Lang, 2014) or scrolling manually through the contents of the `GetCapabilities` output in a browser.

```
library(sf)
sf::sf_use_s2(FALSE)
```

```
qf = list(request = "GetFeature", typeName = "fifao:FAO_MAJOR")
file = tempfile(fileext = ".gml")
GET(url = base_url, path = endpoint, query = qf, write_disk(file))
fao_areas = read_sf(file)
```

In order to keep geometry validity along the data access chain, and since standards and underlying open-source server solutions (such as GeoServer) have been built on the Simple Features access, it is important to deactivate the new default behavior introduced in **sf**, and to not use the S2 geometry model at data access time. This is done with above code `sf::sf_use_s2(FALSE)`. Also note the use of `write_disk()` to ensure that the results are written to disk rather than loaded into memory, allowing them to be imported with **sf**.

For many everyday tasks, however, a higher-level interface may be more appropriate, and a number of R packages, and tutorials, have been developed precisely for this purpose. The package **ows4R** has been developed for working with OWS services. It provides a stable interface to common access services, such as the WFS, WCS for data, CSW for metadata, and WPS for processing. The OGC services coverage is described in the README of the package, hosted at github.com/eblondel/ows4R, with new standard protocols under investigation/development.

Based on the above example, the code below shows how to perform `getCapabilities` and `getFeatures` operations with this package. The **ows4R** package relies on the principle of clients. To interact with an OWS service (such as WFS), a client is created as follows:

```
library(ows4R)
WFS = WFSClient$new(
  url = "https://www.fao.org/fishery/geoserver/wfs",
  serviceVersion = "1.0.0",
  logger = "INFO"
)
```

The operations are then accessible from this client object, e.g., `getCapabilities` or `getFeatures`.

```
library(ows4R)
caps = WFS$getCapabilities()
features = WFS$getFeatures("fifao:FAO_MAJOR")
```

As explained before, when accessing data with OGC services, handling **sf** features should be done by deactivating the new default behavior introduced in **sf**, with `sf::sf_use_s2(FALSE)`. This is done by default with **ows4R**.

Additional examples are available through vignettes, such as how to access raster data with the WCS, or how to access metadata with the CSW.

8.9 Visual outputs

R supports many different static and interactive graphics formats. Chapter 9 covers map-making in detail, but it is worth mentioning ways to output visualizations here. The most general method to save a static plot is to open a graphic device, create a plot, and close it, for example:

```
png(filename = "lifeExp.png", width = 500, height = 350)
plot(world["lifeExp"])
dev.off()
```

Other available graphic devices include `pdf()`, `bmp()`, `jpeg()`, and `tiff()`. You can specify several properties of the output plot, including width, height and resolution.

Additionally, several graphic packages provide their own functions to save a graphical output. For example, the **tmap** package has the `tmap_save()` function. You can save a `tmap` object to different graphic formats or an HTML file by specifying the object name and a file path to a new file.

```
library(tmap)
tmap_obj = tm_shape(world) + tm_polygons(col = "lifeExp")
tmap_save(tmap_obj, filename = "lifeExp_tmap.png")
```

On the other hand, you can save interactive maps created in the **mapview** package as an HTML file or image using the `mapshot2()` function:

```
library(mapview)
mapview_obj = mapview(world, zcol = "lifeExp", legend = TRUE)
mapshot2(mapview_obj, url = "my_interactive_map.html")
```

8.10 Exercises

E1. List and describe three types of vector, raster, and geodatabase formats.

E2. Name at least two differences between the **sf** functions `read_sf()` and `st_read()`.

E3. Read the `cycle_hire_xy.csv` file from the **spData** package as a spatial object (Hint: it is located in the `misc` folder). What is a geometry type of the loaded object?

E4. Download the borders of Germany using **rnaturalearth**, and create a new object called `germany_borders`. Write this new object to a file of the GeoPackage format.

E5. Download the global monthly minimum temperature with a spatial resolution of 5 minutes using the **geodata** package. Extract the June values, and save them to a file named `tmin_june.tif` file (hint: use `terra::subset()`).

E6. Create a static map of Germany's borders, and save it to a PNG file.

E7. Create an interactive map using data from the `cycle_hire_xy.csv` file. Export this map to a file called `cycle_hire.html`.

Part II

Extensions

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

9

Making maps with R

Prerequisites

- This chapter requires the following packages that we have already been using:

```
library(sf)
library(terra)
library(dplyr)
library(spData)
library(spDataLarge)
```

- The main package used in this chapter is **tmap**. We recommend you to install its development version from the r-universe repository, which is updated more frequently than the CRAN version:

```
install.packages("tmap", repos = c("https://r-tmap.r-universe.dev",
                                   "https://cloud.r-project.org"))
```

- It uses the following visualization packages (also install **shiny** if you want to develop interactive mapping applications):

```
library(tmap)    # for static and interactive maps
library(leaflet) # for interactive maps
library(ggplot2) # tidyverse data visualization package
```

- You also need to read-in a couple of datasets as follows for Section 4.3:

```
nz_elev = rast(system.file("raster/nz_elev.tif", package = "spDataLarge"))
```

9.1 Introduction

A satisfying and important aspect of geographic research is communicating the results. Map-making — the art of cartography — is an ancient skill involving communication, attention to detail, and an element of creativity. Static mapping in R is straightforward with the `plot()` function, as we saw in Section 2.2.3. It is possible to create advanced maps using base R methods (Murrell, 2016). The focus of this chapter, however, is cartography with dedicated map-making packages. When learning a new skill, it makes sense to gain depth-of-knowledge in one area before branching out. Map-making is no exception, hence this chapter's coverage of one package (**tmap**) in depth rather than many superficially.

In addition to being fun and creative, cartography also has important practical applications. A carefully crafted map can be the best way of communicating the results of your work, but poorly designed maps can leave a bad impression. Common design issues include poor placement, size and readability of text and careless selection of colors, as outlined in the style guide of the *Journal of Maps*. Furthermore, poor map-making can hinder the communication of results (Brewer, 2015):

> Amateur-looking maps can undermine your audience's ability to understand important information and weaken the presentation of a professional data investigation. Maps have been used for several thousand years for a wide variety of purposes. Historic examples include maps of buildings and land ownership in the Old Babylonian dynasty more than 3000 years ago and Ptolemy's world map in his masterpiece *Geography* nearly 2000 years ago (Talbert, 2014).

Map-making has historically been an activity undertaken only by, or on behalf of, the elite. This has changed with the emergence of open source mapping software such as the R package **tmap** and the 'print layout' in QGIS, which allow anyone to make high-quality maps, enabling 'citizen science'. Maps are also often the best way to present the findings of geocomputational research in a way that is accessible. Map-making is therefore a critical part of geocomputation and its emphasis is not only on describing, but also *changing* the world.

This chapter shows how to make a wide range of maps. The next section covers a range of static maps, including aesthetic considerations, facets and inset maps. Sections 9.3 to 9.5 cover animated and interactive maps (including web maps and mapping applications). Finally, Section 9.6 covers a range of alternative map-making packages including **ggplot2** and **cartogram**.

9.2 Static maps

Static maps are the most common type of visual output from geocomputation. They are usually stored in standard formats including `.png` and `.pdf` for *graphical* raster and vector outputs, respectively. Initially, static maps were the only type of maps that R could produce. Things have advanced with the release of **sp** (see Pebesma and Bivand, 2005), and many map-making techniques, functions, and packages have been developed since then. However, despite the innovation of interactive mapping, static plotting was still the emphasis of geographic data visualization in R a decade later (Cheshire and Lovelace, 2015).

The generic `plot()` function is often the fastest way to create static maps from vector and raster spatial objects (see Sections 2.2.3 and 2.3.3). Sometimes, simplicity and speed are priorities, especially during the development phase of a project, and this is where `plot()` excels. The base R approach is also extensible, with `plot()` offering dozens of arguments. Another approach is the **grid** package which allows low-level control of static maps, as illustrated in chapter 14 of Murrell (2016). This part of the book focuses on **tmap** and emphasizes the essential aesthetic and layout options.

tmap is a powerful and flexible map-making package with sensible defaults. It has a concise syntax that allows for the creation of attractive maps with minimal code which will be familiar to **ggplot2** users. It also has the unique capability to generate static and interactive maps using the same code via `tmap_mode()`. Finally, it accepts a wider range of spatial classes (including **sf** and **terra** objects) than alternatives such as **ggplot2**.

9.2.1 tmap basics

Like **ggplot2**, **tmap** is based on the idea of a 'grammar of graphics' (Wilkinson and Wills, 2005). This involves a separation between the input data and the aesthetics (how data are visualized): each input dataset can be 'mapped' in a range of different ways including location on the map (defined by data's `geometry`), color, and other visual variables. The basic building block is `tm_shape()` (which defines input data: a vector or raster object), followed by one or more

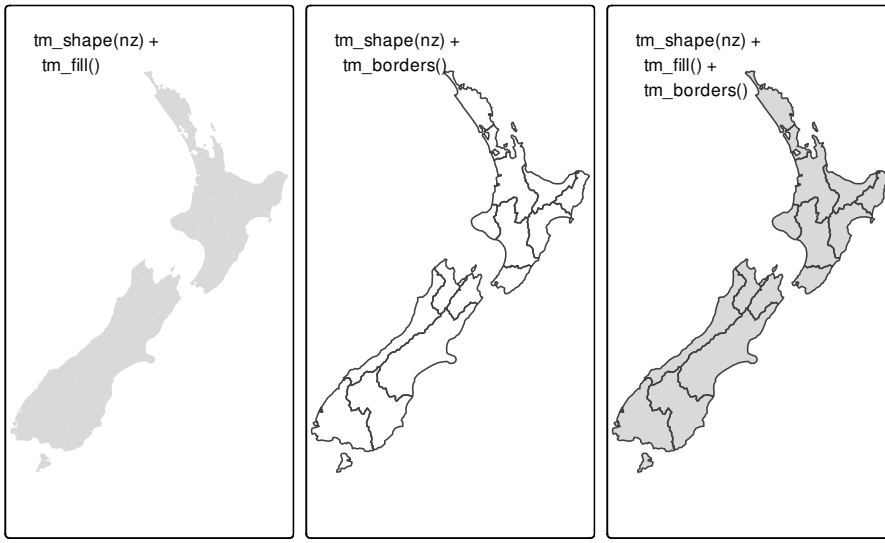

FIGURE 9.1 New Zealand's shape plotted with fill (left), border (middle) and fill and border (right) layers added using tmap functions.

layer elements such as `tm_fill()` and `tm_dots()`. This layering is demonstrated in the chunk below, which generates the maps presented in Figure 9.1:

```
# Add fill layer to nz shape
tm_shape(nz) +
  tm_fill()
# Add border layer to nz shape
tm_shape(nz) +
  tm_borders()
# Add fill and border layers to nz shape
tm_shape(nz) +
  tm_fill() +
  tm_borders()
```

The object passed to `tm_shape()` in this case is `nz`, an `sf` object representing the regions of New Zealand (see Section 2.2.1 for more on `sf` objects). Layers are added to represent `nz` visually, with `tm_fill()` and `tm_borders()` creating shaded areas (left panel) and border outlines (middle panel) in Figure 9.1, respectively.

This is an intuitive approach to map-making: the common task of *adding* new layers is undertaken by the addition operator `+`, followed by `tm_*()`. The asterisk (*) refers to a wide range of layer types which have self-explanatory names including:

- `tm_fill()`: shaded areas for (multi)polygons
- `tm_borders()`: border outlines for (multi)polygons
- `tm_polygons()`: both, shaded areas and border outlines for (multi)polygons
- `tm_lines()`: lines for (multi)linestrings
- `tm_symbols()`: symbols for (multi)points, (multi)linestrings, and (multi)polygons
- `tm_raster()`: colored cells of raster data (there is also `tm_rgb()` for rasters with three layers)
- `tm_text()`: text information for (multi)points, (multi)linestrings, and (multi)polygons

This layering is illustrated in the right panel of Figure 9.1, the result of adding a border *on top of* the fill layer.

`qtm()` is a handy function to create **q**uick **t**hematic **m**aps (hence the snappy name). It is concise and provides a good default visualization in many cases: `qtm(nz)`, for example, is equivalent to `tm_shape(nz) + tm_fill() + tm_borders()`. Further, layers can be added concisely using multiple `qtm()` calls, such as `qtm(nz) + qtm(nz_height)`. The disadvantage is that it makes aesthetics of individual layers harder to control, explaining why we avoid teaching it in this chapter.

9.2.2 Map objects

A useful feature of **tmap** is its ability to store *objects* representing maps. The code chunk below demonstrates this by saving the last plot in Figure 9.1 as an object of class `tmap` (note the use of `tm_polygons()` which condenses `tm_fill() + tm_borders()` into a single function):

```
map_nz = tm_shape(nz) + tm_polygons()
class(map_nz)
#> [1] "tmap"
```

`map_nz` can be plotted later, for example by adding other layers (as shown below) or simply running `map_nz` in the console, which is equivalent to `print(map_nz)`.

New *shapes* can be added with `+ tm_shape(new_obj)`. In this case, `new_obj` represents a new spatial object to be plotted on top of preceding layers. When a new shape is added in this way, all subsequent aesthetic functions refer to it, until another new shape is added. This syntax allows the creation of maps with multiple shapes and layers, as illustrated in the next code chunk which uses the function `tm_raster()` to plot a raster layer (with `col_alpha` set to make the layer semi-transparent):

```
map_nz1 = map_nz +
  tm_shape(nz_elev) + tm_raster(col_alpha = 0.7)
```

Building on the previously created `map_nz` object, the preceding code creates a new map object `map_nz1` that contains another shape (`nz_elev`) representing average elevation across New Zealand (see Figure 9.2, left). More shapes and layers can be added, as illustrated in the code chunk below which creates `nz_water`, representing New Zealand's territorial waters, and adds the resulting lines to an existing map object.

```
nz_water = st_union(nz) |>
  st_buffer(22200) |>
  st_cast(to = "LINESTRING")
map_nz2 = map_nz1 +
  tm_shape(nz_water) + tm_lines()
```

There is no limit to the number of layers or shapes that can be added to `tmap` objects, and the same shape can even be used multiple times. The final map illustrated in Figure 9.2 is created by adding a layer representing high points (stored in the object `nz_height`) onto the previously created `map_nz2` object with `tm_symbols()` (see `?tm_symbols` for details on **tmap**'s point plotting functions). The resulting map, which has four layers, is illustrated in the right-hand panel of Figure 9.2:

```
map_nz3 = map_nz2 +
  tm_shape(nz_height) + tm_symbols()
```

A useful and little known feature of **tmap** is that multiple map objects can be arranged in a single 'metaplot' with `tmap_arrange()`. This is demonstrated in the code chunk below which plots `map_nz1` to `map_nz3`, resulting in Figure 9.2.

```
tmap_arrange(map_nz1, map_nz2, map_nz3)
```

More elements can also be added with the `+` operator. Aesthetic settings, however, are controlled by arguments to layer functions.

9.2.3 Visual variables

The plots in the previous section demonstrate **tmap**'s default aesthetic settings. Gray shades are used for `tm_fill()` and `tm_symbols()` layers and a continuous black line is used to represent lines created with `tm_lines()`. Of course, these default values and other aesthetics can be overridden. The purpose of this section is to show how.

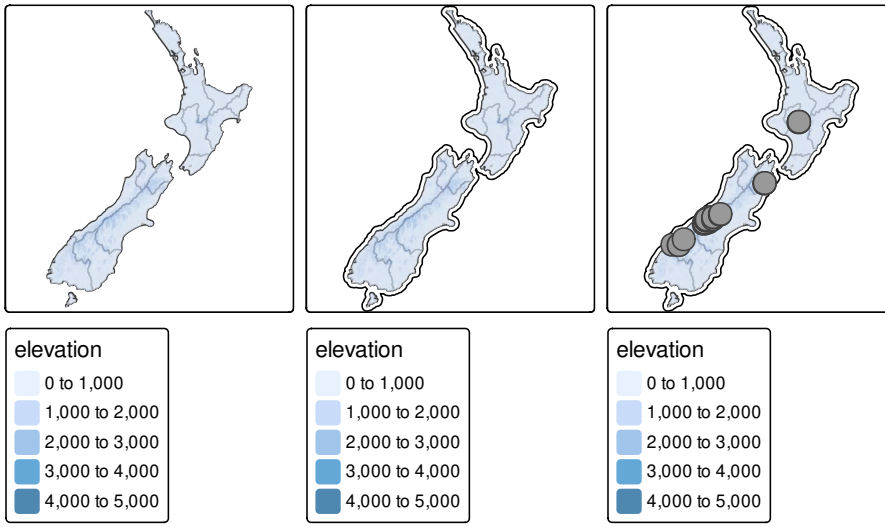

FIGURE 9.2 Maps with added layers to the final map of Figure 9.1.

There are two main types of map aesthetics: those that change with the data and those that are constant. Unlike **ggplot2**, which uses the helper function `aes()` to represent variable aesthetics, **tmap** accepts a few aesthetic arguments, depending on a selected layer type:

- `fill`: fill color of a polygon
- `col`: color of a polygon border, line, point, or raster
- `lwd`: line width
- `lty`: line type
- `size`: size of a symbol
- `shape`: shape of a symbol

Additionally, we may customize the fill and border color transparency using `fill_alpha` and `col_alpha`.

To map a variable to an aesthetic, pass its column name to the corresponding argument, and to set a fixed aesthetic, pass the desired value instead.[1] The impact of setting these with fixed values is illustrated in Figure 9.3.

```
ma1 = tm_shape(nz) + tm_polygons(fill = "red")
ma2 = tm_shape(nz) + tm_polygons(fill = "red", fill_alpha = 0.3)
ma3 = tm_shape(nz) + tm_polygons(col = "blue")
ma4 = tm_shape(nz) + tm_polygons(lwd = 3)
```

[1] If there is a clash between a fixed value and a column name, the column name takes precedence. This can be verified by running the next code chunk after running `nz$red = 1:nrow(nz)`.

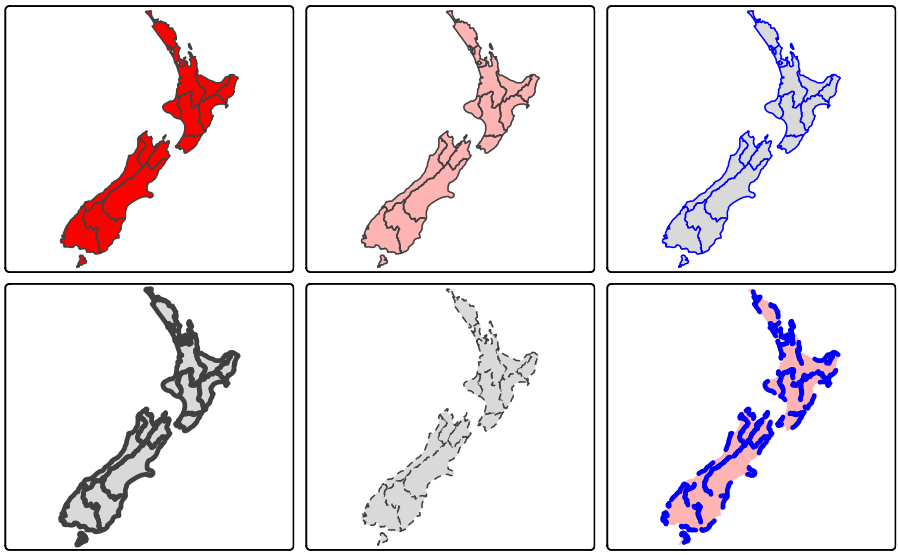

FIGURE 9.3 Impact of changing commonly used fill and border aesthetics to fixed values.

```
ma5 = tm_shape(nz) + tm_polygons(lty = 2)
ma6 = tm_shape(nz) + tm_polygons(fill = "red", fill_alpha = 0.3,
                                 col = "blue", lwd = 3, lty = 2)
tmap_arrange(ma1, ma2, ma3, ma4, ma5, ma6)
```

Like base R plots, arguments defining aesthetics can also receive values that vary. Unlike the base R code below (which generates the left panel in Figure 9.4), **tmap** aesthetic arguments will not accept a numeric vector:

```
plot(st_geometry(nz), col = nz$Land_area)  # works
tm_shape(nz) + tm_fill(fill = nz$Land_area) # fails
#> Error: palette should be a character value
```

Instead `fill` (and other aesthetics that can vary such as `lwd` for line layers and `size` for point layers) requires a character string naming an attribute associated with the geometry to be plotted. Thus, one would achieve the desired result as follows (Figure 9.4, right panel):

```
tm_shape(nz) + tm_fill(fill = "Land_area")
```

Each visual variable has three related additional arguments, with suffixes of `.scale`, `.legend`, and `.free`. For example, the `tm_fill()` function has arguments

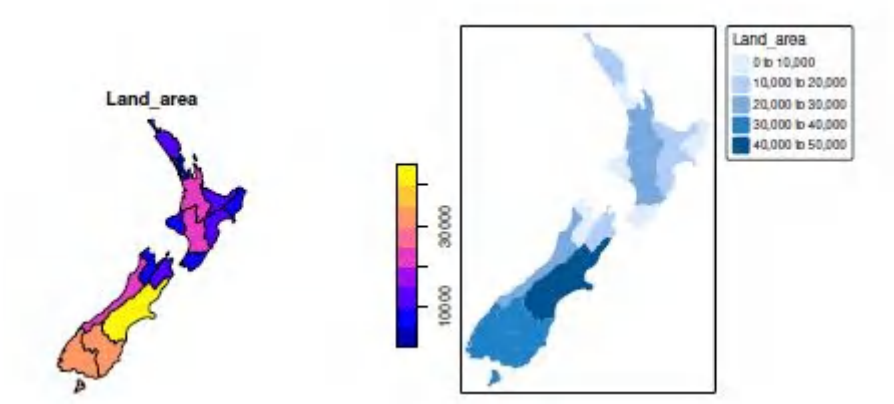

FIGURE 9.4 Comparison of base (left) and tmap (right) handling of a numeric color field.

such as `fill`, `fill.scale`, `fill.legend`, and `fill.free`. The `.scale` argument determines how the provided values are represented on the map and in the legend (Section 9.2.4), while the `.legend` argument is used to customize the legend settings, such as its title, orientation, or position (Section 9.2.5). The `.free` argument is relevant only for maps with many facets to determine if each facet has the same or different scale and legend.

9.2.4 Scales

Scales control how the values are represented on the map and in the legend, and they largely depend on the selected visual variable. For example, when our visual variable is `col`, then `col.scale` controls how the colors of spatial objects are related to the provided values; and when our visual variable is `size`, then `size.scale` controls how the sizes represent the provided values. By default, the used scale is `tm_scale()`, which selects the visual settings automatically given by the input data type (factor, numeric, and integer).

Let's see how the scales work by customizing polygons' fill colors. Color settings are an important part of map design – they can have a major impact on how spatial variability is portrayed as illustrated in Figure 9.5. This figure shows four ways of coloring regions in New Zealand depending on median income, from left to right (and demonstrated in the code chunk below):

- The default setting uses 'pretty' breaks, described in the next paragraph
- `breaks` allows you to manually set the breaks
- `n` sets the number of bins into which numeric variables are categorized
- `values` defines the color scheme, for example, `BuGn`

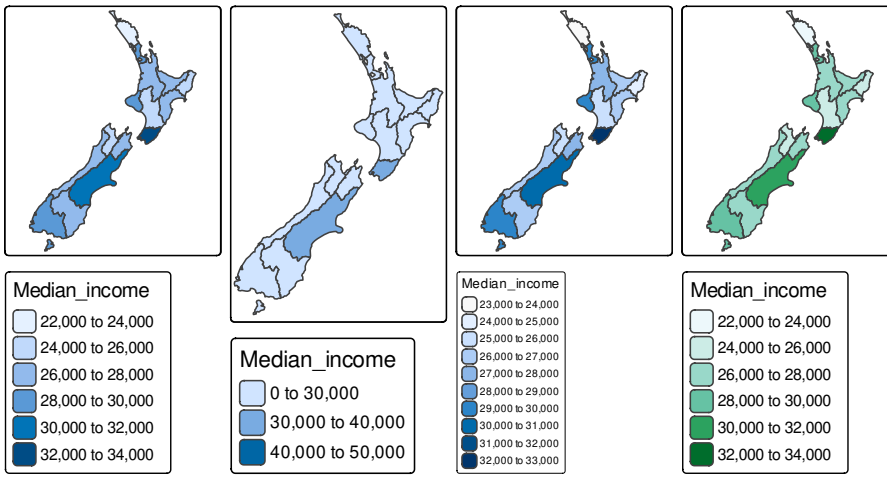

FIGURE 9.5 Color settings. The results show (from left to right): default settings, manual breaks, n breaks, and the impact of changing the palette.

```
tm_shape(nz) + tm_polygons(fill = "Median_income")
tm_shape(nz) + tm_polygons(fill = "Median_income",
                           fill.scale = tm_scale(breaks = c(0, 30000, 40000, 50000)))
tm_shape(nz) + tm_polygons(fill = "Median_income",
                           fill.scale = tm_scale(n = 10))
tm_shape(nz) + tm_polygons(fill = "Median_income",
                           fill.scale = tm_scale(values = "BuGn"))
```

All of the above arguments (`breaks`, `n`, and `values`) also work for other types of visual variables. For example, `values` expects a vector of colors or a palette name for `fill.scale` or `col.scale`, a vector of sizes for `size.scale`, or a vector of symbols for `shape.scale`.

We are also able to customize scales using a family of functions that start with the `tm_scale_` prefix. The most important ones are `tm_scale_intervals()`, `tm_scale_continuous()`, and `tm_scale_categorical()`.

The `tm_scale_intervals()` function splits the input data values into a set of intervals. In addition to manually setting `breaks`, **tmap** allows users to specify algorithms to create breaks with the `style` argument automatically. The default is `tm_scale_intervals(style = "pretty")`, which rounds breaks into whole numbers where possible and spaces them evenly. Other options are listed below and presented in Figure 9.6.

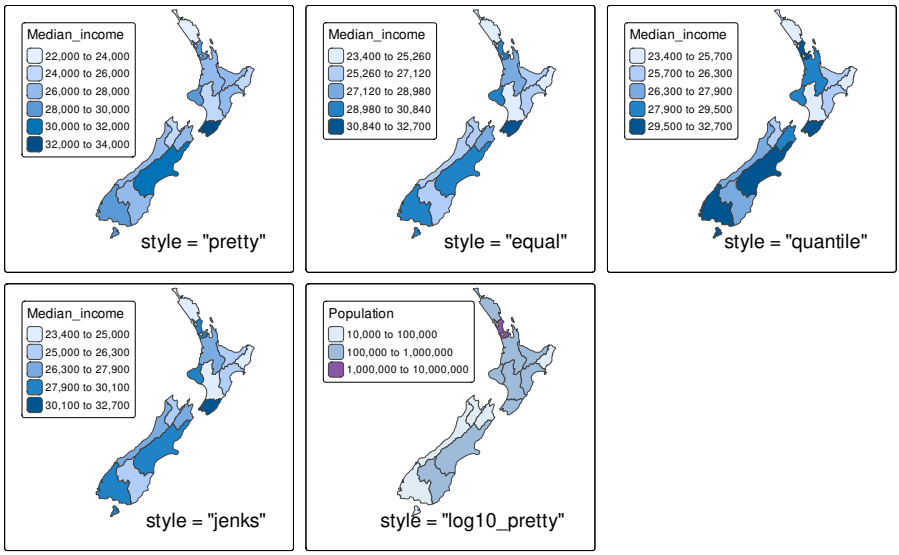

FIGURE 9.6 Different interval scale methods set using the style argument in tmap.

- `style = "equal"`: divides input values into bins of equal range and is appropriate for variables with a uniform distribution (not recommended for variables with a skewed distribution as the resulting map may end up having little color diversity)
- `style = "quantile"`: ensures the same number of observations fall into each category (with the potential downside that bin ranges can vary widely)
- `style = "jenks"`: identifies groups of similar values in the data and maximizes the differences between categories
- `style = "log10_pretty"`: a common logarithmic (the logarithm to base 10) version of the regular pretty style used for variables with a right-skewed distribution

Although `style` is an argument of **tmap** functions, in fact it originates as an argument in `classInt::classIntervals()` — see the help page of this function for details.

The `tm_scale_continuous()` function presents a continuous color field and is particularly suited for continuous rasters (Figure 9.7, left panel). In case of variables with q skewed distribution, you can also use its variants – `tm_scale_continuous_log()` and `tm_scale_continuous_log1p()`. Finally, `tm_scale_categorical()` was designed to represent categorical values and ensures that each category receives a unique color (Figure 9.7, right panel).

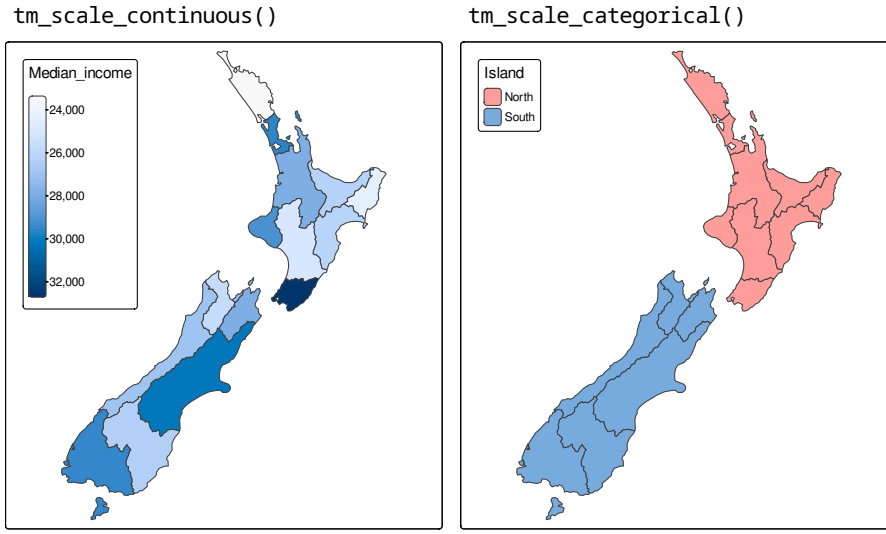

FIGURE 9.7 Continuous and categorical scales in tmap.

Palettes define the color ranges associated with the bins and determined by the `tm_scale_*()` functions, and its `breaks` and `n` arguments described above. It expects a vector of colors or a new color palette name, which can be found interactively with `cols4all::c4a_gui()`. You can also add a - as the color palette name prefix to reverse the palette order.

All of the default `values` of the visual variables, such as default color palettes for different types of input variables, can be found with `tmap_options()`. For example, run `tmap_options()$values.var`.

There are three main groups of color palettes: categorical, sequential and diverging (Figure 9.8), and each of them serves a different purpose.[2] Categorical palettes consist of easily distinguishable colors and are most appropriate for categorical data without any particular order such as state names or land cover classes. Colors should be intuitive: rivers should be blue, for example, and pastures green. Avoid too many categories: maps with large legends and many colors can be uninterpretable.[3]

The second group is sequential palettes. These follow a gradient, for example from light to dark colors (light colors often tend to represent lower values), and are appropriate for continuous (numeric) variables. Sequential palettes

[2] A fourth group of color palettes, called bivariate, also exists. They are used when we want to represent relations between two variables on one map.

[3] `fill = "MAP_COLORS"` can be used in maps with a large number of individual polygons (for example, a map of individual countries) to create unique fill colors for adjacent polygons.

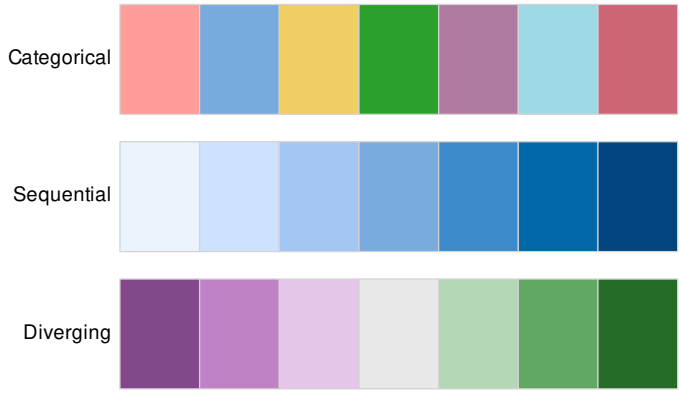

FIGURE 9.8 Examples of categorical, sequential and diverging palettes.

can be single (`greens` goes from light to dark green, for example) or multi-color/hue (`yl_gn_bu` is gradient from light yellow to blue via green, for example), as demonstrated in the code chunk below — output not shown, run the code yourself to see the results!

```
tm_shape(nz) +
  tm_polygons("Median_income", fill.scale = tm_scale(values = "greens"))
tm_shape(nz) +
  tm_polygons("Median_income", fill.scale = tm_scale(values = "yl_gn_bu"))
```

The third group, diverging palettes, typically range between three distinct colors (purple-white-green in Figure 9.8) and are usually created by joining two single-color sequential palettes with the darker colors at each end. Their main purpose is to visualize the difference from an important reference point, e.g., a certain temperature, the median household income or the mean probability for a drought event. The reference point's value can be adjusted in **tmap** using the `midpoint` argument.

```
tm_shape(nz) +
  tm_polygons("Median_income",
              fill.scale = tm_scale_continuous(values = "pu_gn_div",
                                               midpoint = 28000))
```

There are two important principles for consideration when working with colors: perceptibility and accessibility. Firstly, colors on maps should match our perception. This means that certain colors are viewed through our experience and also cultural lenses. For example, green colors usually represent vegetation or lowlands, and blue is connected with water or coolness. Color palettes should also be easy to understand to effectively convey information. It should

be clear which values are lower and which are higher, and colors should change gradually. Secondly, changes in colors should be accessible to the largest number of people. Therefore, it is important to use colorblind friendly palettes as often as possible.[4]

9.2.5 Legends

After we decided on our visual variable and its properties, we should move our attention toward the related map legend style. Using the `tm_legend()` function, we may change its title, position, orientation, or even disable it. The most important argument in this function is `title`, which sets the title of the associated legend. In general, a map legend title should provide two pieces of information: what the legend represents and what the units are of the presented variable. The following code chunk demonstrates this functionality by providing a more attractive name than the variable name `Land_area` (note the use of `expression()` to create superscript text):

```
legend_title = expression("Area (km"^2*")")
tm_shape(nz) +
  tm_polygons(fill = "Land_area", fill.legend = tm_legend(title = legend_title))
```

The default legend orientation in **tmap** is `"portrait"`, however, an alternative legend orientation, `"landscape"`, is also possible. Other than that, we can also customize the location of the legend using the `position` argument.

```
tm_shape(nz) +
  tm_polygons(fill = "Land_area",
              fill.legend = tm_legend(title = legend_title,
                                      orientation = "landscape",
                                      position = tm_pos_out("center", "bottom")))
```

The legend position (and also the position of several other map elements in **tmap**) can be customized using one of a few functions. The two most important are:

- `tm_pos_out()`: the default, adds the legend outside of the map frame area. We can customize its location with two values that represent the horizontal position (`"left"`, `"center"`, or `"right"`), and the vertical position (`"bottom"`, `"center"`, or `"top"`)
- `tm_pos_in()`: puts the legend inside of the map frame area. We may decide on its position using two arguments, where the first one can be `"left"`, `"center"`, or `"right"`, and the second one can be `"bottom"`, `"center"`, or `"top"`.

[4] See the "Color vision" options and the "Color Blind Friendliness" panel in `cols4all::c4a_gui()`.

Alternatively, we may just provide a vector of two values (or two numbers between 0 and 1) here – and in such case, the legend will be put inside the map frame.

9.2.6 Layouts

The map layout refers to the combination of all map elements into a cohesive map. Map elements include among others the objects to be mapped, the map grid, the scale bar, the title, and margins, while the color settings covered in the previous section relate to the palette and breakpoints used to affect how the map looks. Both may result in subtle changes that can have an equally large impact on the impression left by your maps.

Additional map elements such as graticules , north arrows, scale bars and map titles have their own functions: `tm_graticules()`, `tm_compass()`, `tm_scalebar()`, and `tm_title()` (Figure 9.9).[5]

```
map_nz +
  tm_graticules() +
  tm_compass(type = "8star", position = c("left", "top")) +
  tm_scalebar(breaks = c(0, 100, 200), text.size = 1, position = c("left", "top")) +
  tm_title("New Zealand")
```

tmap also allows a wide variety of layout settings to be changed, some of which, produced using the following code (see `args(tm_layout)` or `?tm_layout` for a full list), are illustrated in Figure 9.10.

```
map_nz + tm_layout(scale = 4)
map_nz + tm_layout(bg.color = "lightblue")
map_nz + tm_layout(frame = FALSE)
```

The other arguments in `tm_layout()` provide control over many more aspects of the map in relation to the canvas on which it is placed. Here are some useful layout settings (some of which are illustrated in Figure 9.11):

- Margin settings including `inner.margin` and `outer.margin`
- Font settings controlled by `fontface` and `fontfamily`
- Legend settings including options such as `legend.show` (whether or not to show the legend) `legend.orientation`, `legend.position`, and `legend.frame`
- Frame width (`frame.lwd`) and an option to allow double lines (`frame.double.line`)
- Color settings controlling `color.sepia.intensity` (how *yellowy* the map looks) and `color.saturation` (a color-grayscale)

[5] Another additional map elements include `tm_grid()`, `tm_logo()` and `tm_credits()`.

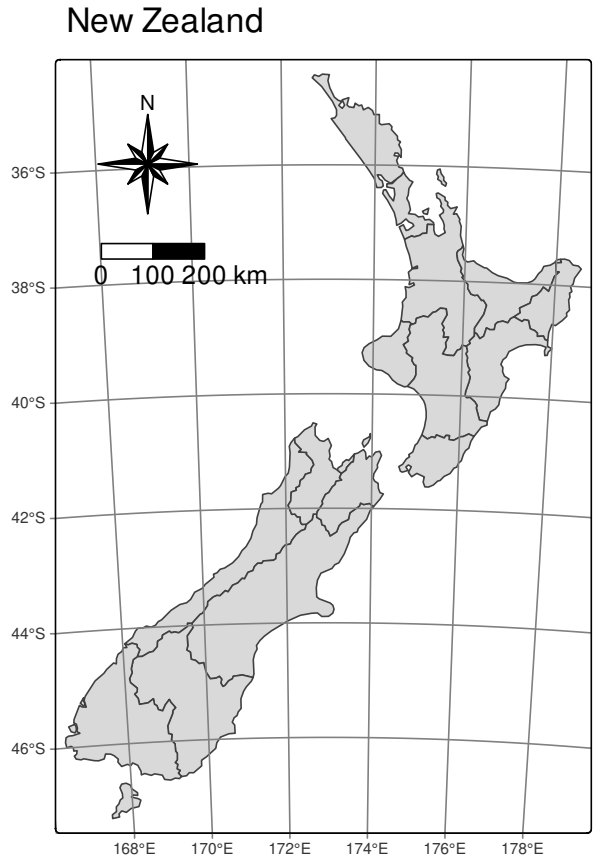

FIGURE 9.9 Map with additional elements: a north arrow and scale bar.

9.2.7 Faceted maps

Faceted maps, also referred to as 'small multiples', are composed of many maps arranged side-by-side, and sometimes stacked vertically (Meulemans et al., 2017). Facets enable the visualization of how spatial relationships change with respect to another variable, such as time. The changing populations of settlements, for example, can be represented in a faceted map with each panel representing the population at a particular moment in time. The time dimension could be represented via another *visual variable* such as color. However, this risks cluttering the map because it will involve multiple overlapping points (cities do not tend to move over time!).

Typically all individual facets in a faceted map contain the same geometry data repeated multiple times, once for each column in the attribute data (this is the default plotting method for `sf` objects, see Chapter 2). However, facets

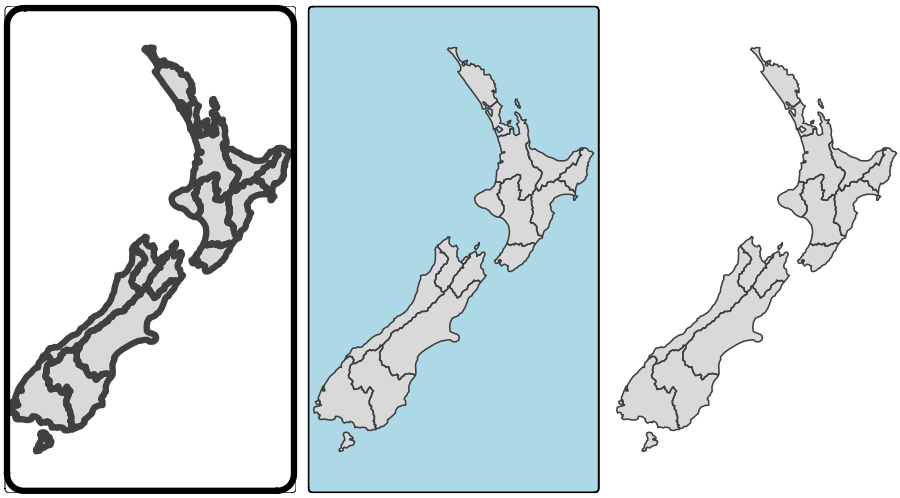

FIGURE 9.10 Layout options specified by (from left to right) scale, bg.color, and frame arguments.

can also represent shifting geometries such as the evolution of a point pattern over time. This use case of a faceted plot is illustrated in Figure 9.12.

```
urb_1970_2030 = urban_agglomerations |>
  filter(year %in% c(1970, 1990, 2010, 2030))

tm_shape(world) +
  tm_polygons() +
  tm_shape(urb_1970_2030) +
  tm_symbols(fill = "black", col = "white", size = "population_millions") +
  tm_facets_wrap(by = "year", nrow = 2)
```

The preceding code chunk demonstrates key features of faceted maps created using the `tm_facets_wrap()` function:

- Shapes that do not have a facet variable are repeated (countries in `world` in this case)
- The `by` argument which varies depending on a variable (`"year"` in this case)
- The `nrow`/`ncol` setting specifying the number of rows and columns that facets should be arranged into

Alternatively, it is possible to use the `tm_facets_grid()` function that allows to have facets based on up to three different variables: one for `rows`, one for `columns`, and possibly one for `pages`.

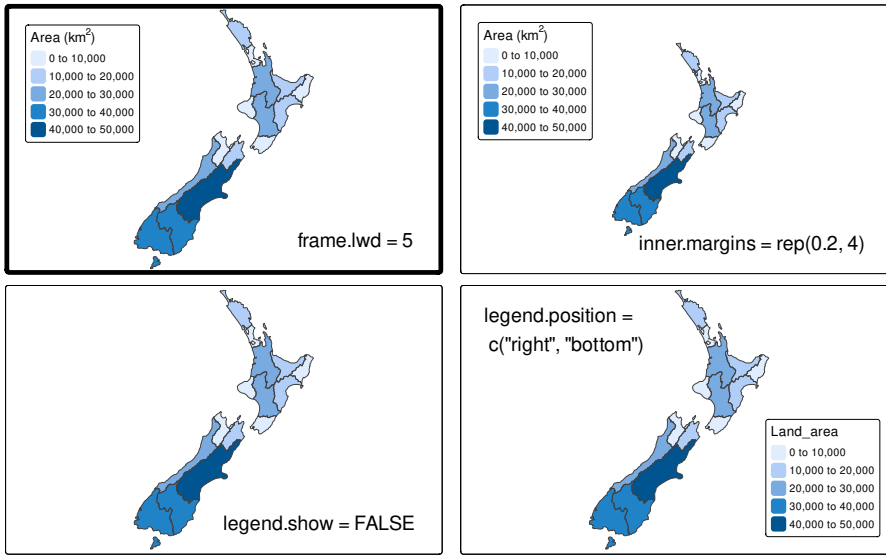

FIGURE 9.11 Selected layout options.

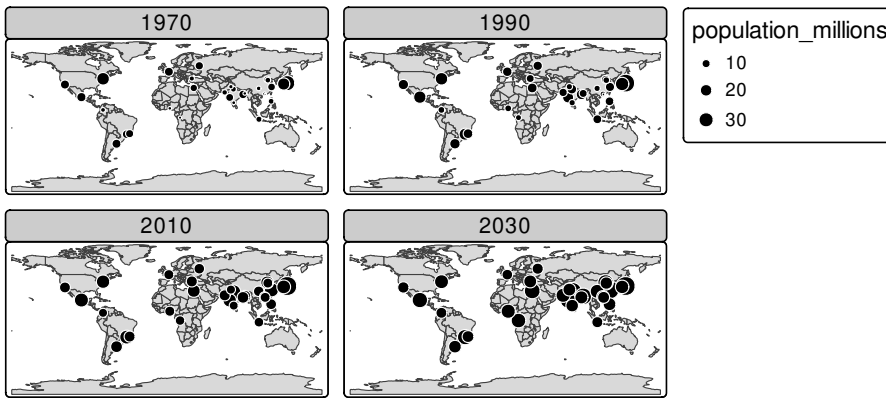

FIGURE 9.12 Faceted map showing the top 30 largest urban agglomerations from 1970 to 2030 based on population projections by the United Nations.

In addition to their utility for showing changing spatial relationships, faceted maps are also useful as the foundation for animated maps (see Section 9.3).

9.2.8 Inset maps

An inset map is a smaller map rendered within or next to the main map. It could serve many different purposes, including providing a context (Figure 9.13) or bringing some non-contiguous regions closer to ease their comparison

(Figure 9.14). They could be also used to focus on a smaller area in more detail or to cover the same area as the map, but representing a different topic.

In the example below, we create a map of the central part of New Zealand's Southern Alps. Our inset map will show where the main map is in relation to the whole New Zealand. The first step is to define the area of interest, which can be done by creating a new spatial object, `nz_region`.

```
nz_region = st_bbox(c(xmin = 1340000, xmax = 1450000,
                      ymin = 5130000, ymax = 5210000),
                    crs = st_crs(nz_height)) |>
  st_as_sfc()
```

In the second step, we create a base-map showing New Zealand's Southern Alps area. This is a place where the most important message is stated.

```
nz_height_map = tm_shape(nz_elev, bbox = nz_region) +
  tm_raster(col.scale = tm_scale_continuous(values = "YlGn"),
            col.legend = tm_legend(position = c("left", "top"))) +
  tm_shape(nz_height) + tm_symbols(shape = 2, col = "red", size = 1) +
  tm_scalebar(position = c("left", "bottom"))
```

The third step consists of the inset map creation. It gives a context and helps to locate the area of interest. Importantly, this map needs to clearly indicate the location of the main map, for example by stating its borders.

```
nz_map = tm_shape(nz) + tm_polygons() +
  tm_shape(nz_height) + tm_symbols(shape = 2, col = "red", size = 0.1) +
  tm_shape(nz_region) + tm_borders(lwd = 3) +
  tm_layout(bg.color = "lightblue")
```

One of the main differences between regular charts (e.g., scatterplots) and maps is that the input data determine the aspect ratio of maps. Thus, in this case, we need to calculate the aspect ratios of our two main datasets, `nz_region` and `nz`. The following function, `norm_dim()` returns the normalized width (`"w"`) and height (`"h"`) of the object (as `"snpc"` units understood by the graphic device).

```
library(grid)
norm_dim = function(obj){
    bbox = st_bbox(obj)
    width = bbox["xmax"] - bbox["xmin"]
    height = bbox["ymax"] - bbox["ymin"]
    w = width / max(width, height)
```

```
    h = height / max(width, height)
    return(unit(c(w, h), "snpc"))
}
main_dim = norm_dim(nz_region)
ins_dim = norm_dim(nz)
```

Next, knowing the aspect ratios, we need to specify the sizes and locations of our two maps – the main map and the inset map – using the `viewport()` function. A viewport is part of a graphics device we use to draw the graphical elements at a given moment. The viewport of our main map is just the representation of its aspect ratio.

```
main_vp = viewport(width = main_dim[1], height = main_dim[2])
```

On the other hand, the viewport of the inset map needs to specify its size and location. Here, we would make the inset map twice smaller as the main one by multiplying the width and height by 0.5, and we will locate it 0.5 cm from the bottom right of the main map frame.

```
ins_vp = viewport(width = ins_dim[1] * 0.5, height = ins_dim[2] * 0.5,
                  x = unit(1, "npc") - unit(0.5, "cm"), y = unit(0.5, "cm"),
                  just = c("right", "bottom"))
```

Finally, we combine the two maps by creating a new, blank canvas, printing out the main map, and then placing the inset map inside of the main map viewport.

```
grid.newpage()
print(nz_height_map, vp = main_vp)
pushViewport(main_vp)
print(nz_map, vp = ins_vp)
```

Inset maps can be saved to file either by using a graphic device (see Section 8.9) or the `tmap_save()` function and its arguments: `insets_tm` and `insets_vp`.

Inset maps are also used to create one map of non-contiguous areas. Probably, the most often used example is a map of the United States, which consists of the contiguous United States, Hawaii and Alaska. It is very important to find the best projection for each individual inset in these types of cases (see Chapter 7 to learn more). We can use US National Atlas Equal Area for the map of the contiguous United States by putting its EPSG code in the `crs` argument of `tm_shape()`.

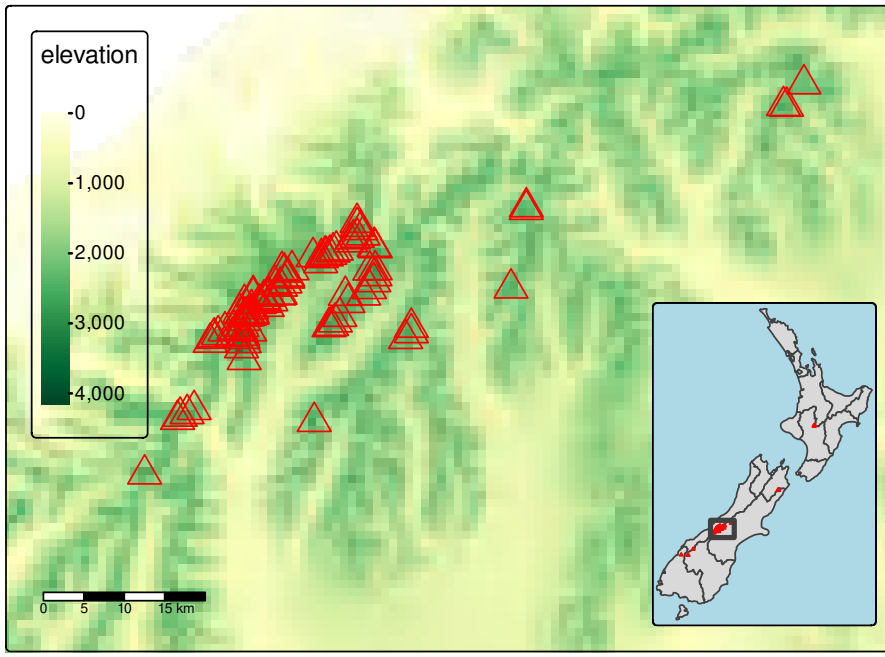

FIGURE 9.13 Inset map providing a context – location of the central part of the Southern Alps in New Zealand.

```
us_states_map = tm_shape(us_states, crs = "EPSG:9311") +
  tm_polygons() +
  tm_layout(frame = FALSE)
```

The rest of our objects, `hawaii` and `alaska`, already have proper projections; therefore, we just need to create two separate maps:

```
hawaii_map = tm_shape(hawaii) +
  tm_polygons() +
  tm_title("Hawaii") +
  tm_layout(frame = FALSE, bg.color = NA,
            title.position = c("LEFT", "BOTTOM"))
alaska_map = tm_shape(alaska) +
  tm_polygons() +
  tm_title("Alaska") +
  tm_layout(frame = FALSE, bg.color = NA)
```

The final map is created by combining, resizing and arranging these three maps:

FIGURE 9.14 Map of the United States.

```
us_states_map
print(hawaii_map, vp = grid::viewport(0.35, 0.1, width = 0.2, height = 0.1))
print(alaska_map, vp = grid::viewport(0.15, 0.15, width = 0.3, height = 0.3))
```

The code presented above is compact and can be used as the basis for other inset maps, but the results, in Figure 9.14, provide a poor representation of the locations and sizes of Hawaii and Alaska. For a more in-depth approach, see the `us-map` vignette from the **geocompkg**.

9.3 Animated maps

Faceted maps, described in Section 9.2.7, can show how spatial distributions of variables change (e.g., over time), but the approach has disadvantages. Facets become tiny when there are many of them. Furthermore, the fact that each facet is physically separated on the screen or page means that subtle differences between facets can be hard to detect.

Animated maps solve these issues. Although they depend on digital publication, this is becoming less of an issue as more and more content moves online. Animated maps can still enhance paper reports: you can always link readers

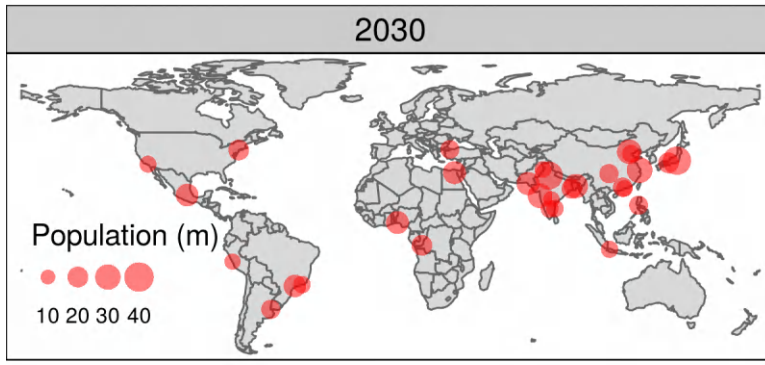

FIGURE 9.15 Animated map showing the top 30 largest urban agglomerations from 1950 to 2030 based on population projects by the United Nations. Animated version available online at: r.geocompx.org.

to a webpage containing an animated (or interactive) version of a printed map to help make it come alive. There are several ways to generate animations in R, including with animation packages such as **gganimate**, which builds on **ggplot2** (see Section 9.6). This section focuses on creating animated maps with **tmap** because its syntax will be familiar from previous sections and the flexibility of the approach.

Figure 9.15 is a simple example of an animated map. Unlike the faceted plot, it does not squeeze multiple maps into a single screen and allows the reader to see how the spatial distribution of the world's most populous agglomerations evolve over time (see the book's website for the animated version).

The animated map illustrated in Figure 9.15 can be created using the same **tmap** techniques that generate faceted maps, demonstrated in Section 9.2.7. There are two differences, however, related to arguments in `tm_facets_wrap()`:

- `nrow = 1, ncol = 1` are added to keep one moment in time as one layer
- `free.coords = FALSE`, which maintains the map extent for each map iteration

These additional arguments are demonstrated in the subsequent code chunk[6]:

[6]There is also a shortcut for this approach: `tm_facets_pagewise()`.

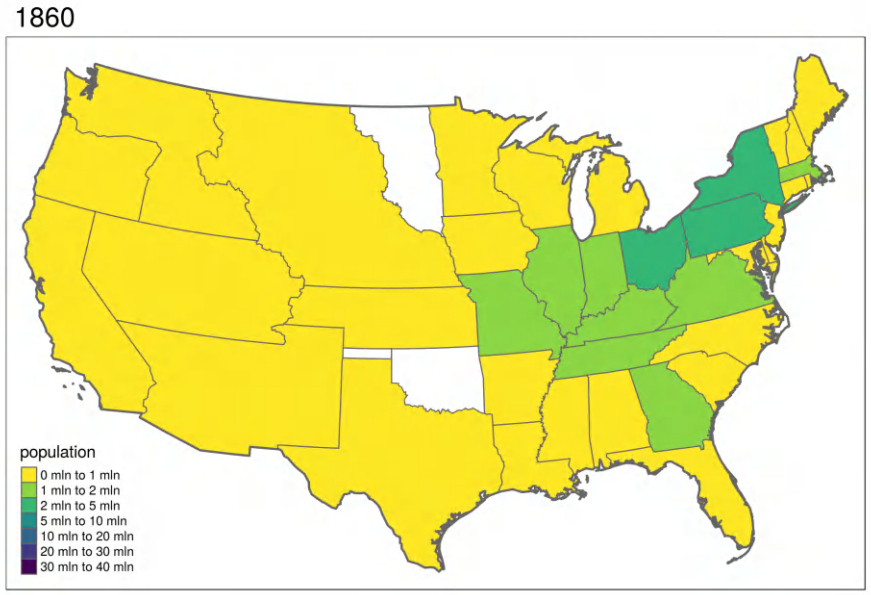

FIGURE 9.16 Animated map showing population growth, state formation and boundary changes in the United States, 1790-2010. Animated version available online at r.geocompx.org.

```
urb_anim = tm_shape(world) + tm_polygons() +
  tm_shape(urban_agglomerations) + tm_symbols(size = "population_millions") +
  tm_facets_wrap(by = "year", nrow = 1, ncol = 1, free.coords = FALSE)
```

The resulting `urb_anim` represents a set of separate maps for each year. The final stage is to combine them and save the result as a `.gif` file with `tmap_animation()`. The following command creates the animation illustrated in Figure 9.15, with a few elements missing, that we will add during the exercises:

```
tmap_animation(urb_anim, filename = "urb_anim.gif", delay = 25)
```

Another illustration of the power of animated maps is provided in Figure 9.16. This shows the development of states in the United States, which first formed in the east and then incrementally to the west and finally into the interior. Code to reproduce this map can be found in the script `code/09-usboundaries.R` in the book GitHub repository.

9.4 Interactive maps

While static and animated maps can enliven geographic datasets, interactive maps can take them to a new level. Interactivity can take many forms, the most common and useful of which is the ability to pan around and zoom into any part of a geographic dataset overlaid on a 'web map' to show context. Less advanced interactivity levels include pop-ups which appear when you click on different features, a kind of interactive label. More advanced levels of interactivity include the ability to tilt and rotate maps, as demonstrated in the **mapdeck** example below, and the provision of "dynamically linked" subplots which automatically update when the user pans and zooms (Pezanowski et al., 2018).

The most important type of interactivity, however, is the display of geographic data on interactive or 'slippy' web maps. The release of the **leaflet** package in 2015 (that uses the leaflet JavaScript library) revolutionized interactive web map creation from within R, and a number of packages have built on these foundations adding new features (e.g., **leaflet.extras2**) and making the creation of web maps as simple as creating static maps (e.g., **mapview** and **tmap**). This section illustrates each approach in the opposite order. We will explore how to make slippy maps with **tmap** (the syntax of which we have already learned), **mapview**, **mapdeck** and finally **leaflet** (which provides low-level control over interactive maps).

A unique feature of **tmap** mentioned in Section 9.2 is its ability to create static and interactive maps using the same code. Maps can be viewed interactively at any point by switching to view mode, using the command `tmap_mode("view")`. This is demonstrated in the code below, which creates an interactive map of New Zealand based on the `tmap` object `map_nz`, created in Section 9.2.2, and illustrated in Figure 9.17:

```
tmap_mode("view")
map_nz
```

Now that the interactive mode has been 'turned on', all maps produced with **tmap** will launch (another way to create interactive maps is with the `tmap_leaflet()` function). Notable features of this interactive mode include the ability to specify the basemap with `tm_basemap()` (or `tmap_options()`) as demonstrated below (result not shown):

```
map_nz + tm_basemap(server = "OpenTopoMap")
```

FIGURE 9.17 Interactive map of New Zealand created with tmap in view mode. Interactive version available online at: r.geocompx.org.

An impressive and little-known feature of **tmap**'s view mode is that it also works with faceted plots. The argument `sync` in `tm_facets_wrap()` can be used in this case to produce multiple maps with synchronized zoom and pan settings, as illustrated in Figure 9.18, which was produced by the following code:

```
world_coffee = left_join(world, coffee_data, by = "name_long")
facets = c("coffee_production_2016", "coffee_production_2017")
tm_shape(world_coffee) + tm_polygons(facets) +
  tm_facets_wrap(nrow = 1, sync = TRUE)
```

Switch **tmap** back to plotting mode with the same function:

```
tmap_mode("plot")
#> i tmap mode set to "plot".
```

If you are not proficient with **tmap**, the quickest way to create interactive maps in R may be with **mapview**. The following 'one liner' is a reliable way to interactively explore a wide range of geographic data formats:

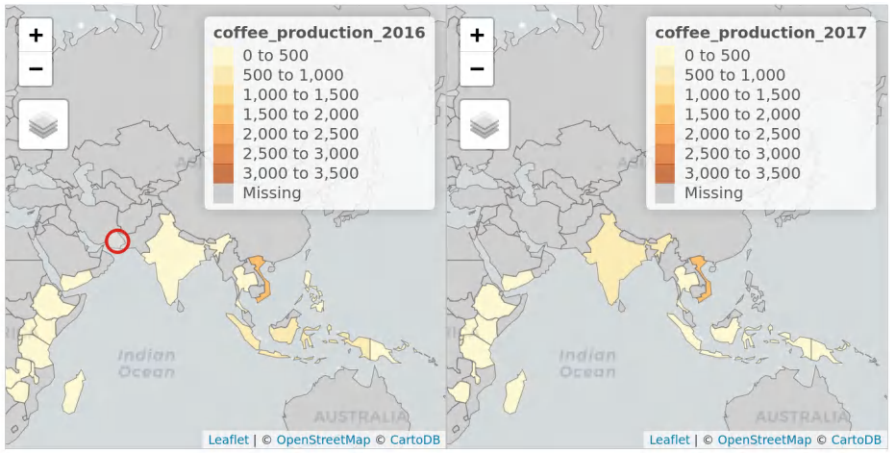

FIGURE 9.18 Faceted interactive maps of global coffee production in 2016 and 2017 in sync, demonstrating tmap's view mode in action.

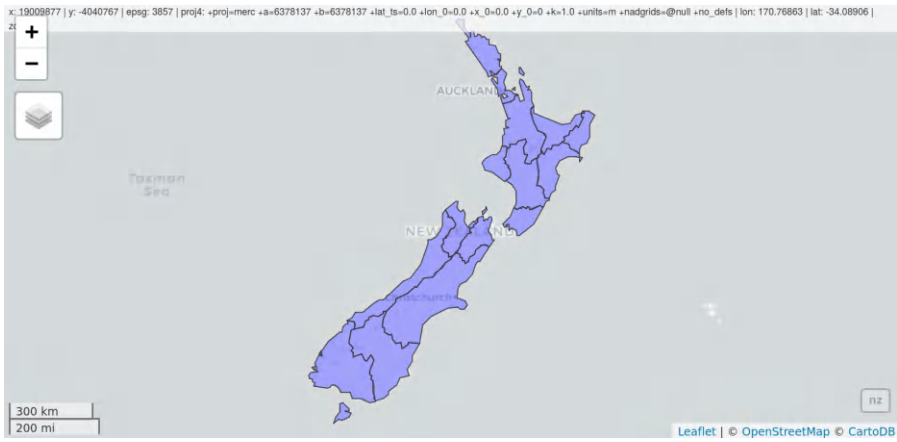

FIGURE 9.19 Illustration of mapview in action.

```
mapview::mapview(nz)
```

mapview has a concise syntax, yet, it is powerful. By default, it has some standard GIS functionality such as mouse position information, attribute queries (via pop-ups), scale bar, and zoom-to-layer buttons. It also offers advanced controls including the ability to 'burst' datasets into multiple layers and the addition of multiple layers with `+` followed by the name of a geographic object. Additionally, it provides automatic coloring of attributes via the `zcol` argument. In essence, it can be considered a data-driven **leaflet** API (see below

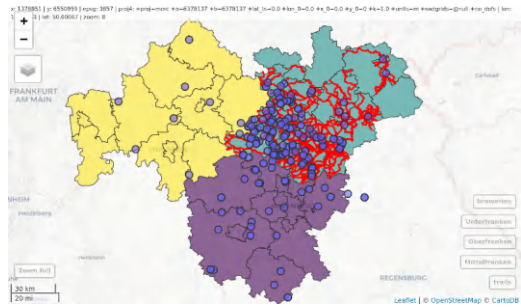

FIGURE 9.20 Using mapview at the end of an sf-based pipe expression.

for more information about **leaflet**). Given that **mapview** always expects a spatial object (including `sf` and `SpatRaster`) as its first argument, it works well at the end of piped expressions. Consider the following example where **sf** is used to intersect lines and polygons and then is visualized with **mapview** (Figure 9.20).

```
library(mapview)
oberfranken = subset(franconia, district == "Oberfranken")
trails |>
  st_transform(st_crs(oberfranken)) |>
  st_intersection(oberfranken) |>
  st_collection_extract("LINESTRING") |>
  mapview(color = "red", lwd = 3, layer.name = "trails") +
  mapview(franconia, zcol = "district") +
  breweries
```

One important thing to keep in mind is that **mapview** layers are added via the `+` operator (similar to **ggplot2** or **tmap**). By default, **mapview** uses the leaflet JavaScript library to render the output maps, which is user-friendly and has a lot of features. However, some alternative rendering libraries could be more performant (work more smoothly on larger datasets). **mapview** allows to set alternative rendering libraries (`"leafgl"` and `"mapdeck"`) in the `mapviewOptions()`.[7] For further information on **mapview**, see the package's website at: r-spatial.github.io/mapview/.

There are other ways to create interactive maps with R. The **googleway** package, for example, provides an interactive mapping interface that is flexible and extensible (see the `googleway-vignette` for details). Another approach by the same author is **mapdeck**, which provides access to Uber's `Deck.gl` framework. Its use of WebGL enables it to interactively visualize large datasets up to

[7] You may also try to use `mapviewOptions(georaster = TRUE)` for more performant visualizations of large raster data.

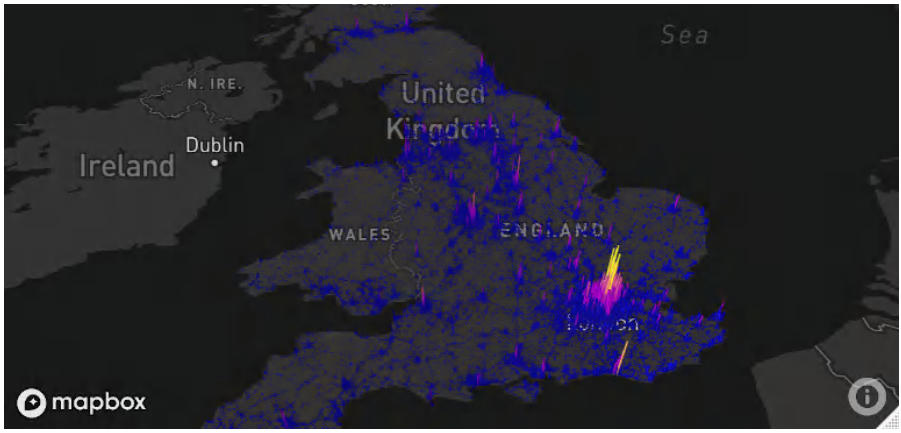

FIGURE 9.21 Map generated by mapdeck, representing road traffic casualties across the UK. Height of 1-km cells represents number of crashes.

millions of points. The package uses Mapbox access tokens, which you must register for before using the package.

Note that the following block assumes the access token is stored in your R environment as `MAPBOX=your_unique_key`. This can be added with `usethis::edit_r_environ()`.

A unique feature of **mapdeck** is its provision of interactive 2.5D perspectives, illustrated in Figure 9.21. This means you can can pan, zoom and rotate around the maps, and view the data 'extruded' from the map. Figure 9.21, generated by the following code chunk, visualizes road traffic crashes in the UK, with bar height representing casualties per area.

```
library(mapdeck)
set_token(Sys.getenv("MAPBOX"))
crash_data = read.csv("https://git.io/geocompr-mapdeck")
crash_data = na.omit(crash_data)
ms = mapdeck_style("dark")
mapdeck(style = ms, pitch = 45, location = c(0, 52), zoom = 4) |>
  add_grid(data = crash_data, lat = "lat", lon = "lng", cell_size = 1000,
           elevation_scale = 50, colour_range = hcl.colors(6, "plasma"))
```

You can zoom and drag the map in the browser, in addition to rotating and tilting it when pressing `Cmd`/`Ctrl`. Multiple layers can be added with the pipe operator, as demonstrated in the `mapdeck` vignettes. **mapdeck** also supports `sf` objects, as can be seen by replacing the `add_grid()` function call in the

preceding code chunk with `add_polygon(data = lnd, layer_id = "polygon_layer")`, to add polygons representing London to an interactive tilted map.

Last is **leaflet** which is the most mature and widely used interactive mapping package in R. **leaflet** provides a relatively low-level interface to the Leaflet JavaScript library and many of its arguments can be understood by reading the documentation of the original JavaScript library (see leafletjs.com).

Leaflet maps are created with `leaflet()`, the result of which is a `leaflet` map object which can be piped to other **leaflet** functions. This allows multiple map layers and control settings to be added interactively, as demonstrated in the code below which generates Figure 9.22 (see rstudio.github.io/leaflet/ for details).

```
pal = colorNumeric("RdYlBu", domain = cycle_hire$nbikes)
leaflet(data = cycle_hire) |>
  addProviderTiles(providers$CartoDB.Positron) |>
  addCircles(col = ~pal(nbikes), opacity = 0.9) |>
  addPolygons(data = lnd, fill = FALSE) |>
  addLegend(pal = pal, values = ~nbikes) |>
  setView(lng = -0.1, 51.5, zoom = 12) |>
  addMiniMap()
```

9.5 Mapping applications

The interactive web maps demonstrated in Section 9.4 can go far. Careful selection of layers to display, basemaps and pop-ups can be used to communicate the main results of many projects involving geocomputation. But the web mapping approach to interactivity has limitations:

- Although the map is interactive in terms of panning, zooming and clicking, the code is static, meaning the user interface is fixed
- All map content is generally static in a web map, meaning that web maps cannot scale to handle large datasets easily
- Additional layers of interactivity, such a graphs showing relationships between variables and 'dashboards', are difficult to create using the web mapping-approach

Overcoming these limitations involves going beyond static web mapping and toward geospatial frameworks and map servers. Products in this field include GeoDjango (which extends the Django web framework and is written in Python), MapServer (a framework for developing web applications, largely written in C and C++) and GeoServer (a mature and powerful map server

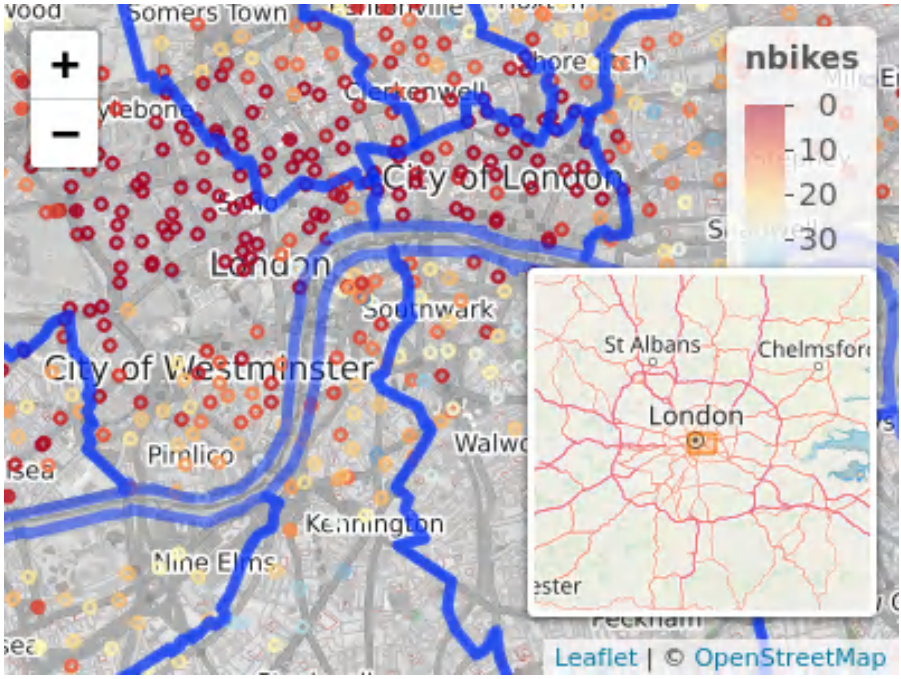

FIGURE 9.22 The leaflet package in action, showing cycle hire points in London. See interactive version [online](https://geocompr.github.io/img/leaflet.html).

written in Java). Each of these is scalable, enabling maps to be served to thousands of people daily, assuming there is sufficient public interest in your maps! The bad news is that such server-side solutions require much skilled developer time to set up and maintain, often involving teams of people with roles such as a dedicated geospatial database administrator (DBA).

Fortunately for R programmers, web mapping applications can now be rapidly created with **shiny**. As described in the open source book Mastering Shiny, **shiny** is an R package and framework for converting R code into interactive web applications (Wickham, 2021). You can embed interactive maps in shiny apps thanks to functions such as `leaflet::renderLeaflet()`. This section gives some context, teaches the basics of **shiny** from a web mapping perspective, and culminates in a full-screen mapping application in less than 100 lines of code.

shiny is well documented at shiny.posit.co, which highlights the two components of every **shiny** app: 'front end' (the bit the user sees) and 'back end' code. In **shiny** apps, these elements are typically created in objects named `ui` and `server` within an R script named `app.R`, which lives in an 'app folder'. This

allows web mapping applications to be represented in a single file, such as the `CycleHireApp/app.R` file in the book's GitHub repo.

In **shiny** apps these are often split into `ui.R` (short for user interface) and `server.R` files, naming conventions used by `shiny-server`, a server-side Linux application for serving shiny apps on public-facing websites. `shiny-server` also serves apps defined by a single `app.R` file in an 'app folder'. Learn more at: https://github.com/rstudio/shiny-server.

Before considering large apps, it is worth seeing a minimal example, named 'lifeApp', in action.[8] The code below defines and launches — with the command `shinyApp()` — a lifeApp, which provides an interactive slider allowing users to make countries appear with progressively lower levels of life expectancy (see Figure 9.23):

```
library(shiny)    # for shiny apps
library(leaflet)  # renderLeaflet function
library(spData)   # loads the world dataset
ui = fluidPage(
  sliderInput(inputId = "life", "Life expectancy", 49, 84, value = 80),
      leafletOutput(outputId = "map")
  )
server = function(input, output) {
  output$map = renderLeaflet({
    leaflet() |>
      # addProviderTiles("OpenStreetMap.BlackAndWhite") |>
      addPolygons(data = world[world$lifeExp < input$life, ])})
}
shinyApp(ui, server)
```

The **user interface** (`ui`) of lifeApp is created by `fluidPage()`. This contains input and output 'widgets' — in this case, a `sliderInput()` (many other `*Input()` functions are available) and a `leafletOutput()`. These are arranged row-wise by default, explaining why the slider interface is placed directly above the map in Figure 9.23 (see `?column` for adding content column-wise).

The **server side** (`server`) is a function with `input` and `output` arguments. `output` is a list of objects containing elements generated by `render*()` function — `renderLeaflet()` which in this example generates `output$map`. Input elements such as `input$life` referred to in the server must relate to elements that exist in the `ui` — defined by `inputId = "life"` in the code above. The function `shinyApp()` combines both the `ui` and `server` elements and serves the results interactively

[8] The word 'app' in this context refers to 'web application' and should not be confused with smartphone apps, the more common meaning of the word.

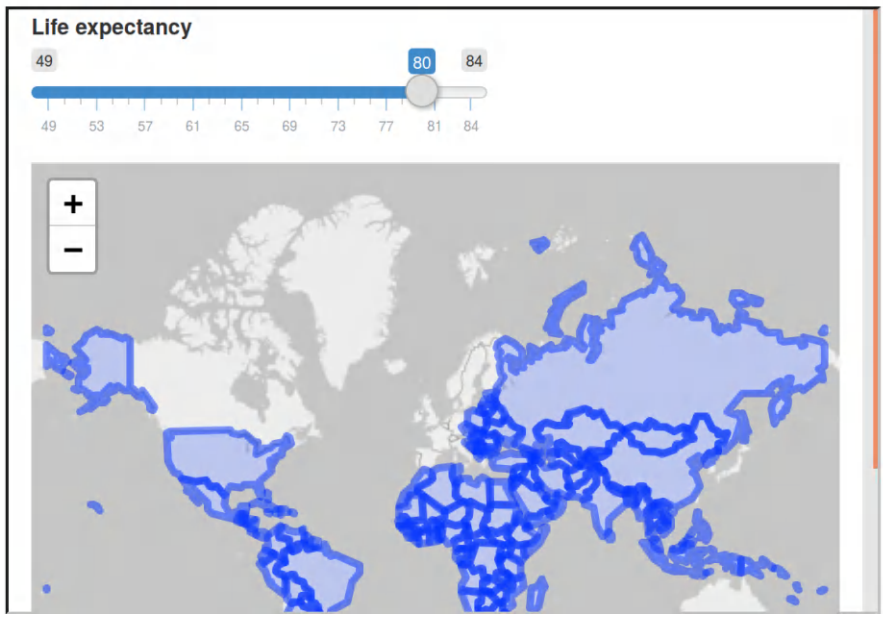

FIGURE 9.23 Screenshot showing minimal example of a web mapping application created with shiny.

via a new R process. When you move the slider in the map shown in Figure 9.23, you are actually causing R code to re-run, although this is hidden from view in the user interface.

Building on this basic example and knowing where to find help (see `?shiny`), the best way forward now may be to stop reading and start programming! The recommended next step is to open the previously mentioned `CycleHireApp/app.R` script in an integrated development environment (IDE) of choice, modify it and re-run it repeatedly. The example contains some of the components of a web mapping application implemented in **shiny** and should 'shine' a light on how they behave.

The `CycleHireApp/app.R` script contains **shiny** functions that go beyond those demonstrated in the simple 'lifeApp' example, deployed at shiny.robinlovelace.net/CycleHireApp. These include `reactive()` and `observe()`, (for creating outputs that respond to the user interface, see `?reactive`) and `leafletProxy()` (for modifying a `leaflet` object that has already been created). Such elements enable web mapping applications implemented in **shiny** (Lovelace et al., 2017). A range of 'events' can be programmed including advanced functionality such as drawing new layers or subsetting data, as described in the shiny section of RStudio's **leaflet** website.

There are a number of ways to run a **shiny** app. For RStudio users, the simplest way is probably to click on the 'Run App' button located in the top right of the source pane when an `app.R`, `ui.R` or `server.R` script is open. **shiny** apps can also be initiated by using `runApp()` with the first argument being the folder containing the app code and data: `runApp("CycleHireApp")` in this case (which assumes a folder named `CycleHireApp` containing the `app.R` script is in your working directory). You can also launch apps from a Unix command line with the command `Rscript -e 'shiny::runApp("CycleHireApp")'`.

Experimenting with apps such as `CycleHireApp` will build not only your knowledge of web mapping applications in R, but also your practical skills. Changing the contents of `setView()`, for example, will change the starting bounding box that the user sees when the app is initiated. Such experimentation should not be done at random, but with reference to relevant documentation, starting with `?shiny`, and motivated by a desire to solve problems such as those posed in the exercises.

shiny used in this way can make prototyping mapping applications faster and more accessible than ever before (deploying **shiny** apps, https://shiny.posit.co/deploy/, is a separate topic beyond the scope of this chapter). Even if your applications are eventually deployed using different technologies, **shiny** undoubtedly allows web mapping applications to be developed in relatively few lines of code (86 in the case of CycleHireApp). That does not stop shiny apps getting rather large. The Propensity to Cycle Tool (PCT) hosted at pct.bike, for example, is a national mapping tool funded by the UK's Department for Transport. The PCT is used by dozens of people each day and has multiple interactive elements based on more than 1000 lines of code (Lovelace et al., 2017).

While such apps undoubtedly take time and effort to develop, **shiny** provides a framework for reproducible prototyping that should aid the development process. One potential problem with the ease of developing prototypes with **shiny** is the temptation to start programming too early, before the purpose of the mapping application has been envisioned in detail. For that reason, despite advocating **shiny**, we recommend starting with the longer established technology of a pen and paper as the first stage for interactive mapping projects. This way your prototype web applications should be limited not by technical considerations, but by your motivations and imagination.

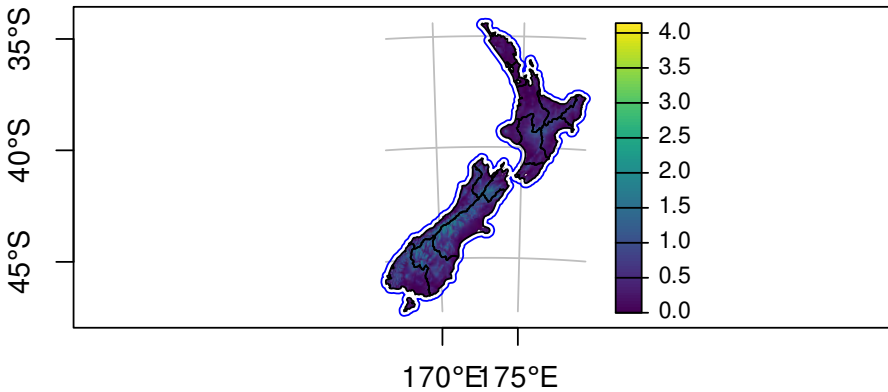

FIGURE 9.24 Map of New Zealand created with plot(). The legend to the right refers to elevation (1000 m above sea level).

9.6 Other mapping packages

tmap provides a powerful interface for creating a wide range of static maps (Section 9.2) and also supports interactive maps (Section 9.4). But there are many other options for creating maps in R. The aim of this section is to provide a taste of some of these and pointers for additional resources: map-making is a surprisingly active area of R package development, so there is more to learn than can be covered here.

The most mature option is to use `plot()` methods provided by core spatial packages **sf** and **terra**, covered in Sections 2.2.3 and 2.3.3, respectively. What we have not mentioned in those sections was that plot methods for vector and raster objects can be combined when the results draw onto the same plot area (elements such as keys in **sf** plots and multi-band rasters will interfere with this). This behavior is illustrated in the subsequent code chunk which generates Figure 9.24. `plot()` has many other options which can be explored by following links in the `?plot` help page and the fifth **sf** vignette `sf5`.

```
g = st_graticule(nz, lon = c(170, 175), lat = c(-45, -40, -35))
plot(nz_water, graticule = g, axes = TRUE, col = "blue")
terra::plot(nz_elev / 1000, add = TRUE, axes = FALSE)
plot(st_geometry(nz), add = TRUE)
```

The **tidyverse** plotting package **ggplot2** also supports `sf` objects with `geom_sf()`. The syntax is similar to that used by **tmap**: an initial `ggplot()` call is followed by one or more layers, that are added with `+ geom_*()`, where `*`

represents a layer type such as `geom_sf()` (for `sf` objects) or `geom_points()` (for points).

ggplot2 plots graticules by default. The default settings for the graticules can be overridden using `scale_x_continuous()`, `scale_y_continuous()` or `coord_sf(datum = NA)`. Other notable features include the use of unquoted variable names encapsulated in `aes()` to indicate which aesthetics vary and switching data sources using the `data` argument, as demonstrated in the code chunk below which creates Figure 9.25:

```
library(ggplot2)
g1 = ggplot() + geom_sf(data = nz, aes(fill = Median_income)) +
  geom_sf(data = nz_height) +
  scale_x_continuous(breaks = c(170, 175))
g1
```

Another benefit of maps based on **ggplot2** is that they can easily be given a level of interactivity when printed using the function `ggplotly()` from the **plotly** package. Try `plotly::ggplotly(g1)`, for example, and compare the result with other **plotly** mapping functions described at: blog.cpsievert.me.

An advantage of **ggplot2** is that it has a strong user community and many add-on packages. It includes **ggspatial**, which enhances **ggplot2**'s mapping capabilities by providing options to add a north arrow (`annotation_north_arrow()`) and a scale bar (`annotation_scale()`), or to add background tiles (`annotation_map_tile()`). It also accepts various spatial data classes with `layer_spatial()`. Thus, we are able to plot `SpatRaster` objects from **terra** using this function as seen in Figure 9.25.

```
library(ggspatial)
ggplot() +
  layer_spatial(nz_elev) +
  geom_sf(data = nz, fill = NA) +
  annotation_scale() +
  scale_x_continuous(breaks = c(170, 175)) +
  scale_fill_continuous(na.value = NA)
```

At the same time, **ggplot2** has a few drawbacks, for example the `geom_sf()` function is not always able to create a desired legend to use from the spatial data. Good additional **ggplot2** resources can be found in the open source ggplot2 book (Wickham, 2016) and in the descriptions of the multitude of '**gg**packages' such as **ggrepel** and **tidygraph**.

We have covered mapping with **sf**, **terra** and **ggplot2** first because these packages are highly flexible, allowing for the creation of a wide range of static maps. Before we cover mapping packages for plotting a specific type of map

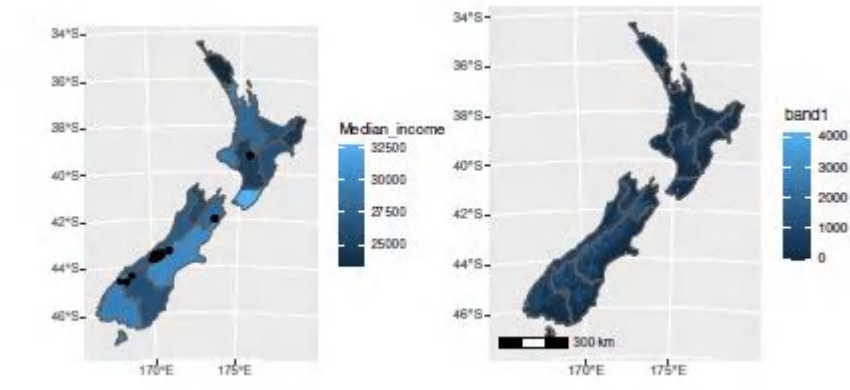

FIGURE 9.25 Comparison of map of New Zealand created with ggplot2 alone (left) and ggplot2 and ggspatial (right).

TABLE 9.1 Selected general-purpose mapping packages.

Package	Title
ggplot2	Create Elegant Data Visualisations Using the Grammar of Graphics
googleway	Accesses Google Maps APIs to Retrieve Data and Plot Maps
ggspatial	Spatial Data Framework for ggplot2
leaflet	Create Interactive Web Maps with Leaflet
mapview	Interactive Viewing of Spatial Data in R
plotly	Create Interactive Web Graphics via 'plotly.js'
rasterVis	Visualization Methods for Raster Data
tmap	Thematic Maps

(in the next paragraph), it is worth considering alternatives to the packages already covered for general-purpose mapping (Table 9.1).

Table 9.1 shows a range of mapping packages that are available, and there are many others not listed in this table. Of note is **mapsf**, which can generate a range of geographic visualizations including choropleth, 'proportional symbol' and 'flow' maps. These are documented in the `mapsf` vignette.

Several packages focus on specific map types, as illustrated in Table 9.2. Such packages create cartograms that distort geographical space, create line maps, transform polygons into regular or hexagonal grids, visualize complex data on grids representing geographic topologies, and create 3D visualizations.

All of the aforementioned packages, however, have different approaches for data preparation and map creation. In the next paragraph, we focus solely

TABLE 9.2 Selected specific-purpose mapping packages, with associated metrics.

Package	Title
cartogram	Create Cartograms with R
geogrid	Turn Geospatial Polygons into Regular or Hexagonal Grids
geofacet	'ggplot2' Faceting Utilities for Geographical Data
linemap	Line Maps
tanaka	Design Shaded Contour Lines (or Tanaka) Maps
rayshader	Create Maps and Visualize Data in 2D and 3D

on the **cartogram** package (Jeworutzki, 2023). Therefore, we suggest to read the geogrid, geofacet, linemap, tanaka, and rayshader documentations to learn more about them.

A cartogram is a map in which the geometry is proportionately distorted to represent a mapping variable. Creation of this type of map is possible in R with **cartogram**, which allows for creating contiguous and non-contiguous area cartograms. It is not a mapping package per se, but it allows for construction of distorted spatial objects that could be plotted using any generic mapping package.

The `cartogram_cont()` function creates contiguous area cartograms. It accepts an `sf` object and name of the variable (column) as inputs. Additionally, it is possible to modify the `intermax` argument – maximum number of iterations for the cartogram transformation. For example, we could represent median income in New Zeleand's regions as a contiguous cartogram (Figure 9.26, right panel) as follows:

```
library(cartogram)
nz_carto = cartogram_cont(nz, "Median_income", itermax = 5)
tm_shape(nz_carto) + tm_polygons("Median_income")
```

cartogram also offers creation of non-contiguous area cartograms using `cartogram_ncont()` and Dorling cartograms using `cartogram_dorling()`. Non-contiguous area cartograms are created by scaling down each region based on the provided weighting variable. Dorling cartograms consist of circles with their area proportional to the weighting variable. The code chunk below demonstrates creation of non-contiguous area and Dorling cartograms of US states' population (Figure 9.27):

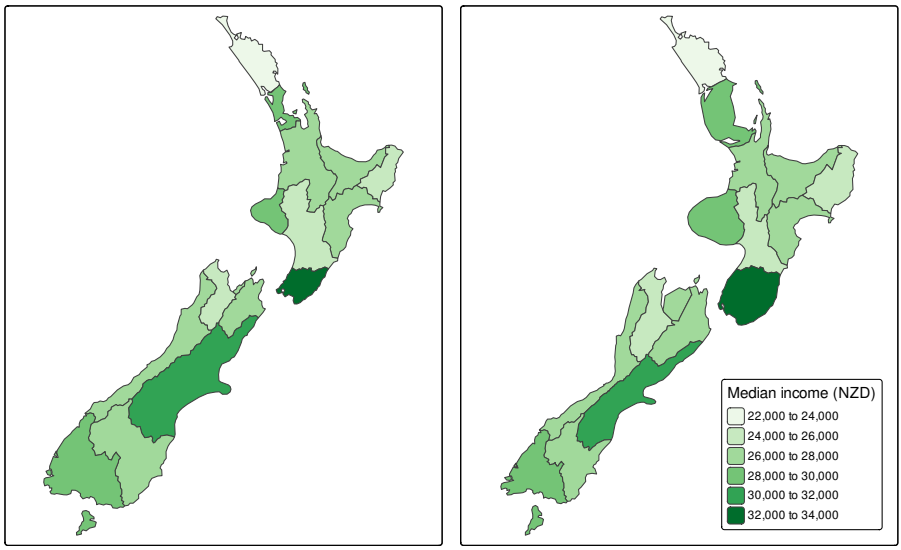

FIGURE 9.26 Comparison of standard map (left) and contiguous area cartogram (right).

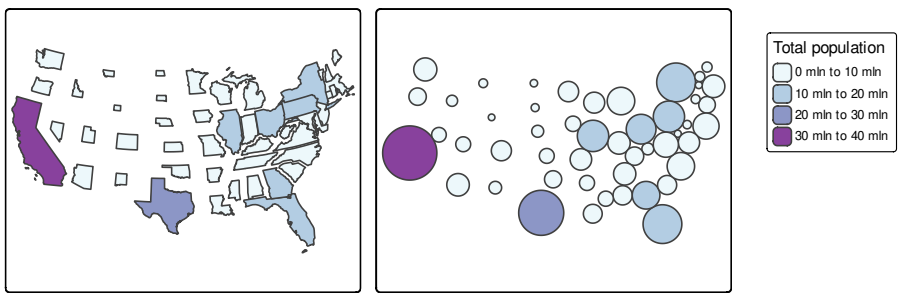

FIGURE 9.27 Comparison of non-contiguous area cartogram (left) and Dorling cartogram (right).

```
us_states9311 = st_transform(us_states, "EPSG:9311")
us_states9311_ncont = cartogram_ncont(us_states9311, "total_pop_15")
us_states9311_dorling = cartogram_dorling(us_states9311, "total_pop_15")
```

9.7 Exercises

These exercises rely on a new object, `africa`. Create it using the `world` and `worldbank_df` datasets from the **spData** package as follows:

```
library(spData)
africa = world |>
  filter(continent == "Africa", !is.na(iso_a2)) |>
  left_join(worldbank_df, by = "iso_a2") |>
  select(name, subregion, gdpPercap, HDI, pop_growth) |>
  st_transform("ESRI:102022") |>
  st_make_valid() |>
  st_collection_extract("POLYGON")
```

We will also use `zion` and `nlcd` datasets from **spDataLarge**:

```
zion = read_sf((system.file("vector/zion.gpkg", package = "spDataLarge")))
nlcd = rast(system.file("raster/nlcd.tif", package = "spDataLarge"))
```

E1. Create a map showing the geographic distribution of the Human Development Index (`HDI`) across Africa with base **graphics** (hint: use `plot()`) and **tmap** packages (hint: use `tm_shape(africa) + ...`).

- Name two advantages of each based on the experience.
- Name three other mapping packages and an advantage of each.
- Bonus: create three more maps of Africa using these three other packages.

E2. Extend the **tmap** created for the previous exercise so the legend has three bins: "High" (`HDI` above 0.7), "Medium" (`HDI` between 0.55 and 0.7) and "Low" (`HDI` below 0.55). Bonus: improve the map aesthetics, for example by changing the legend title, class labels and color palette.

E3. Represent `africa`'s subregions on the map. Change the default color palette and legend title. Next, combine this map and the map created in the previous exercise into a single plot.

E4. Create a land cover map of Zion National Park.

- Change the default colors to match your perception of the land cover categories
- Add a scale bar and north arrow and change the position of both to improve the map's aesthetic appeal
- Bonus: Add an inset map of Zion National Park's location in the context of the state of Utah. (Hint: an object representing Utah can be subset from the `us_states` dataset.)

E5. Create facet maps of countries in Eastern Africa:

- With one facet showing HDI and the other representing population growth (hint: using variables `HDI` and `pop_growth`, respectively)
- With a 'small multiple' per country

E6. Building on the previous facet map examples, create animated maps of East Africa:

- Showing each country in order
- Showing each country in order with a legend showing the HDI

E7. Create an interactive map of HDI in Africa:

- With **tmap**
- With **mapview**
- With **leaflet**
- Bonus: For each approach, add a legend (if not automatically provided) and a scale bar

E8. Sketch on paper ideas for a web mapping application that could be used to make transport or land-use policies more evidence-based:

- In the city you live, for a couple of users per day
- In the country you live, for dozens of users per day
- Worldwide for hundreds of users per day and large data serving requirements

E9. Update the code in `coffeeApp/app.R` so that instead of centering on Brazil the user can select which country to focus on:

- Using `textInput()`
- Using `selectInput()`

E10. Reproduce Figure 9.1 and Figure 9.7 as closely as possible using the **ggplot2** package.

E11. Join `us_states` and `us_states_df` together and calculate a poverty rate for each state using the new dataset. Next, construct a continuous area cartogram based on total population. Finally, create and compare two maps of the poverty rate: (1) a standard choropleth map and (2) a map using the created cartogram boundaries. What is the information provided by the first and the second map? How do they differ from each other?

E12. Visualize population growth in Africa. Next, compare it with the maps of a hexagonal and regular grid created using the **geogrid** package.

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

10

Bridges to GIS software

Prerequisites

- This chapter requires QGIS, SAGA and GRASS GIS to be installed and the following packages to be attached:

```
library(sf)
library(terra)
library(qgisprocess)
library(Rsagacmd)
library(rgrass)
library(rstac)
library(gdalcubes)
```

10.1 Introduction

A defining feature of interpreted languages with an interactive console — technically a read-eval-print loop (REPL) — such as R is the way you interact with them: rather than relying on pointing and clicking on different parts of a screen, you type commands into the console and execute them with the `Enter` key. A common and effective workflow when using interactive development environments such as RStudio or VS Code is to type code into source files in a source editor and control interactive execution of the code with a shortcut such as `Ctrl+Enter`.

Command line interfaces (CLIs) are not unique to R: most early computing environments relied on a command line 'shell' and it was only after the invention and widespread adoption of the computer mouse in the 1990s that graphical user interfaces (GUIs) became common. GRASS GIS, the longest-standing continuously developed open source GIS software, for example, relied on its CLI before it gained a GUI (Landa, 2008). Most popular GIS software projects are GUI-driven. You *can* interact with QGIS, SAGA, GRASS GIS

and gvSIG from system terminals and embedded CLIs, but their design encourages most people to interact with them by 'pointing and clicking'. An unintended consequence of this is that most GIS users miss out on the advantages of CLI-driven and scriptable approaches. According to the creator of the popular QGIS software (Sherman, 2008):

> With the advent of 'modern' GIS software, most people want to point and click their way through life. That's good, but there is a tremendous amount of flexibility and power waiting for you with the command line. Many times you can do something on the command line in a fraction of the time you can do it with a GUI.

The 'CLI vs. GUI' debate does not have to be adversarial: both ways of working have advantages, depending on a range of factors including the task (with drawing new features being well suited to GUIs), the level of reproducibility desired, and the user's skillset. GRASS GIS is a good example of GIS software that is primarily based on a CLI but which also has a prominent GUI. Likewise, while R is focused on its CLI, IDEs such as RStudio provide a GUI for improving accessibility. Software cannot be neatly categorized into CLI or GUI-based. However, interactive command-line interfaces have several important advantages in terms of:

- Automating repetitive tasks
- Enabling transparency and reproducibility
- Encouraging software development by providing tools to modify existing functions and implement new ones
- Developing future-proof and efficient programming skills which are in high demand
- Improving touch typing, a key skill in the digital age

On the other hand, good GUIs also have advantages, including:

- 'Shallow' learning curves meaning geographic data can be explored and visualized without hours of learning a new language
- Support for 'digitizing' (creating new vector datasets), including trace, snap and topological tools[1]
- Enables georeferencing (matching raster images to existing maps) with ground control points and orthorectification

[1]The **mapedit** R package allows the quick editing of a few spatial features in a browser window opened from R but not professional, large-scale cartographic digitizing.

- Supports stereoscopic mapping (e.g., LiDAR and structure from motion)

Another advantage of dedicated GIS software projects is that they provide access to hundreds of 'geoalgorithms' via 'GIS bridges' (Neteler and Mitasova, 2008). Such bridges to these computational recipes for enhancing R's capabilities for solving geographic data problems are the topic of this chapter.

A command line interface is an environment for interacting with computer programs by typing and entering successive commands (command lines). `bash` in Linux and `PowerShell` in Windows are well-known examples that allow the user to control almost any part of their operating system. IDEs such as RStudio and VS Code provide code auto-completion and other features to improve the user experience when developing code.

R is a natural choice for people wanting to build bridges between reproducible data analysis workflows and GIS because it *originated* as an interface language. A key feature of R (and its predecessor S) is that it provides access to statistical algorithms in other languages (particularly FORTRAN and C), but from a powerful high-level functional language with an intuitive REPL environment, which C and FORTRAN lacked (Chambers, 2016). R continues this tradition with interfaces to numerous languages, notably C++.

Although R was not designed as a command line GIS, its ability to interface with dedicated GISs gives it astonishing geospatial capabilities. With GIS bridges, R can replicate more diverse workflows, with the additional reproducibility, scalability and productivity benefits of controlling them from a programming environment and a consistent CLI. Furthermore, R outperforms GISs in some areas of geocomputation, including interactive/animated map-making (see Chapter 9) and spatial statistical modeling (see Chapter 12).

This chapter focuses on 'bridges' to three mature open source GIS products, summarized in Table 10.1:

- QGIS, via the package **qgisprocess** [Dunnington et al. (2024); Section 10.2]
- SAGA, via **Rsagacmd** [Pawley (2023); Section 10.3]
- GRASS GIS, via **rgrass** [Bivand (2023); Section 10.4]

There have also been major developments in enabling open source GIS software to write and execute R scripts inside QGIS (see docs.qgis.org) and GRASS GIS (see grasswiki.osgeo.org).

In addition to the three R-GIS bridges mentioned above, this chapter also provides a brief introduction to R interfaces to spatial libraries (Section 10.6), spatial databases (Section 10.7), and cloud-based processing of Earth observation data (Section 10.8).

TABLE 10.1 Comparison between three open-source GIS. Hybrid refers to the support of vector and raster operations.

GIS	First Release	No. Functions	Support
QGIS	2002	>1000	hybrid
SAGA	2004	>600	hybrid
GRASS GIS	1982	>500	hybrid

10.2 qgisprocess: a bridge to QGIS and beyond

QGIS is the most popular open-source GIS (Table 10.1; Graser and Olaya (2015)). QGIS provides a unified interface to QGIS's native geoalgorithms, GDAL, and — when they are installed — from other *providers* such as GRASS GIS, and SAGA (Graser and Olaya, 2015). Since version 3.14 (released in summer 2020), QGIS ships with the `qgis_process` command-line utility for accessing a bounty of functionality for geocomputation. `qgis_process` provides access to 300+ geoalgorithms in the standard QGIS installation and 1,000+ via plugins to external providers such as GRASS GIS and SAGA.

The **qgisprocess** package provides access to `qgis_process` from R. The package requires QGIS — and any other relevant plugins such as GRASS GIS and SAGA, used in this chapter — to be installed and available to the system. For installation instructions, see **qgisprocess**'s documentation.

A quick way to get up-and-running with **qgisprocess** if you have Docker installed is via the `qgis` image developed as part of this project. Assuming you have Docker installed and sufficient computational resources, you can run an R session with **qgisprocess** and relevant plugins with the following command (see the geocompx/docker repository for details):

```
docker run -e DISABLE_AUTH=true -p 8786:8787 ghcr.io/geocompx/docker:qgis
```

```
library(qgisprocess)
#> Attempting to load the cache ... Success!
#> QGIS version: 3.30.3-'s-Hertogenbosch
#> ...
```

This package automatically tries to detect a QGIS installation and complains if it cannot find it.[2] There are a few possible solutions when the configuration fails: you can set `options(qgisprocess.path = "path/to/your_qgis_process")`, or set

[2] You can see details of the detection process with `qgis_configure()`.

up the `R_QGISPROCESS_PATH` environment variable. The above approaches can also be used when you have more than one QGIS installation and want to decide which one to use. For more details, please refer to the **qgisprocess** 'getting started' vignette. Next, we can find which plugins (meaning different software) are available on our computer:

```
qgis_plugins()
#> # A tibble: 4 × 2
#>   name                      enabled
#>   <chr>                     <lgl>
#> 1 grassprovider             FALSE
#> 2 otbprovider               FALSE
#> 3 processing                TRUE
#> 4 processing_saga_nextgen FALSE
```

This tells us that the GRASS GIS (`grassprovider`) and SAGA (`processing_saga_nextgen`) plugins are available on the system but are not yet enabled. Since we need both later on in the chapter, let's enable them.

```
qgis_enable_plugins(c("grassprovider", "processing_saga_nextgen"),
                    quiet = TRUE)
```

Please note that aside from installing SAGA on your system, you also need to install the QGIS Python plugin Processing Saga NextGen. You can do so from within QGIS with the Plugin Manager or programmatically with the help of the Python package qgis-plugin-manager (at least on Linux).

`qgis_providers()` lists the name of the software and the corresponding count of available geoalgorithms.

```
qgis_providers()
#> # A tibble: 7 × 3
#>   provider provider_title    algorithm_count
#>   <chr>    <chr>                       <int>
#> 1 gdal     GDAL                           56
#> 2 grass    GRASS                         306
#> 3 qgis     QGIS                           50
#> 4 3d       QGIS (3D)                       1
#> 5 native   QGIS (native c++)              243
#> 6 pdal     QGIS (PDAL)                    17
#> 7 sagang   SAGA Next Gen                  509
```

The output table affirms that we can use QGIS geoalgorithms (`native`, `qgis`, `3d`, `pdal`) and external ones from the third-party providers GDAL, SAGA and GRASS GIS through the QGIS interface.

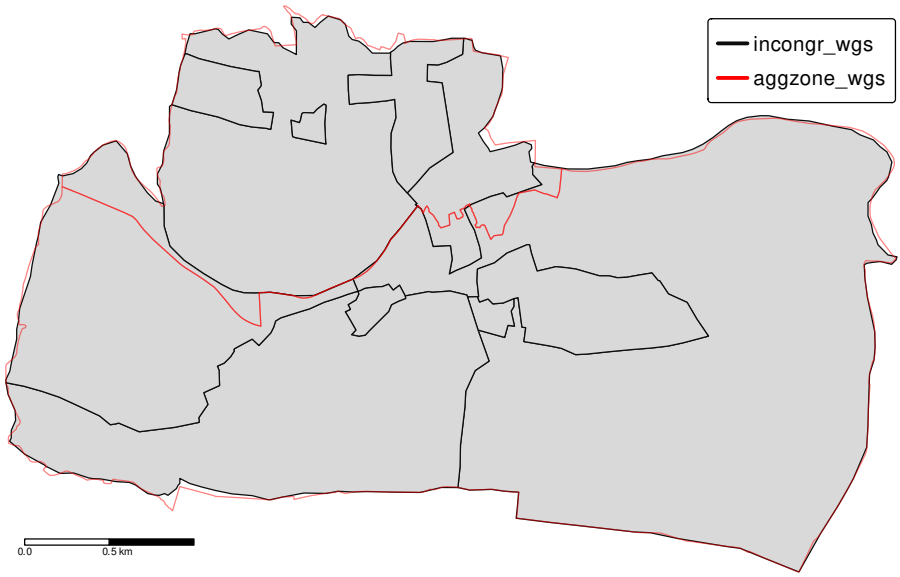

FIGURE 10.1 Two areal units: incongruent (black lines) and aggregating zones (red borders).

Now, we are ready for some geocomputation with QGIS and friends, from within R! Let's try two example case studies. The first one shows how to unite two polygonal datasets with different borders (Section 10.2.1). The second one focuses on deriving new information from a digital elevation model represented as a raster (Section 10.2.2).

10.2.1 Vector data

Consider a situation when you have two polygon objects with different spatial units (e.g., regions, administrative units). Our goal is to merge these two objects into one, containing all of the boundary lines and related attributes. We use again the incongruent polygons we have already encountered in Section 4.2.8 (Figure 10.1). Both polygon datasets are available in the **spData** package, and for both we would like to use a geographic CRS (see also Chapter 7).

```
data("incongruent", "aggregating_zones", package = "spData")
incongr_wgs = st_transform(incongruent, "EPSG:4326")
aggzone_wgs = st_transform(aggregating_zones, "EPSG:4326")
```

The first step is to find an algorithm that can merge two vector objects. To list all of the available algorithms, we can use the `qgis_algorithms()` function.

This function returns a data frame containing all of the available providers and the algorithms they contain.[3]

```
# output not shown
qgis_algorithms()
```

To find an algorithm, we can use the `qgis_search_algorithms()` function. Assuming that the short description of the function contains the word "union", we can run the following code to find the algorithm of interest:

```
qgis_search_algorithms("union")
#> # A tibble: 2 × 5
#>   provider provider_title    group          algorithm         algorithm_title
#>   <chr>    <chr>             <chr>          <chr>             <chr>
#> 1 native   QGIS (native c++) Vector overlay native:multiunion Union (multiple)
#> 2 native   QGIS (native c++) Vector overlay native:union      Union
```

One of the algorithms on the above list, `"native:union"`, sounds promising. The next step is to find out what this algorithm does and how we can use it. This is the role of the `qgis_show_help()`, which returns a short summary of what the algorithm does, its arguments, and outputs.[4] This makes its output rather long. The following command returns a data frame with each row representing an argument required by `"native:union"` and columns with the name, description, type, default value, available values, and acceptable values associated with each:

```
alg = "native:union"
union_arguments = qgis_get_argument_specs(alg)
union_arguments
#> # A tibble: 5 × 6
#>   name    description qgis_type default_value available_values acceptable_...
#>   <chr>   <chr>       <chr>     <list>        <list>           <list>
#> 1 INPUT   Input layer source    <NULL>        <NULL>           <chr [1]>
#> 2 OVERLAY Overlay la… source    <NULL>        <NULL>           <chr [1]>
#> 3 OVERLA… Overlay fi… string    <NULL>        <NULL>           <chr [3]>
#> 4 OUTPUT  Union       sink      <NULL>        <NULL>           <chr [1]>
#> 5 GRID_S… Grid size   number    <NULL>        <NULL>           <chr [3]>

#> [[1]]
#> [1] "A numeric value"
```

[3] Therefore, if you cannot see an expected provider, it is probably because you still need to install some external GIS software.

[4] We can also extract some of information independently with `qgis_get_description()`, `qgis_get_argument_specs()`, and `qgis_get_output_specss()`.

```
#> [2] "field:FIELD_NAME to use a data defined value taken from the FIELD_NAME
#>      field"
#> [3] "expression:SOME EXPRESSION to use a data defined value calculated using
#>      a custom QGIS expression"
```

The arguments, contained in `union_arguments$name`, are `INPUT`, `OVERLAY`, `OVERLAY_FIELDS_PREFIX`, and `OUTPUT`. `union_arguments$acceptable_values` contains a list with the possible input values for each argument. Many functions require inputs representing paths to a vector layer; **qgisprocess** functions accept `sf` objects for such arguments. Objects from the **terra** and **stars** package can be used when a "path to a raster layer" is expected. This can be very convenient, but we recommend providing the path to your spatial data on disk when you only read it in to submit it to a **qgisprocess** algorithm: the first thing **qgisprocess** does when executing a geoalgorithm is to export the spatial data living in your R session back to disk in a format known to QGIS such as .gpkg or .tif files. This can increase algorithm runtimes.

The main function of **qgisprocess** is `qgis_run_algorithm()`, which sends inputs to QGIS and returns the outputs. It accepts the algorithm name and a set of named arguments shown in the help list, and it performs expected calculations. In our case, three arguments seem important: `INPUT`, `OVERLAY`, and `OUTPUT`. The first one, `INPUT`, is our main vector object `incongr_wgs`, while the second one, `OVERLAY`, is `aggzone_wgs`. The last argument, `OUTPUT`, is an output file name, which **qgisprocess** will automatically choose and create in `tempdir()` if none is provided.

```
union = qgis_run_algorithm(alg,
  INPUT = incongr_wgs, OVERLAY = aggzone_wgs
)
union
#>  $ OUTPUT: 'qgis_outputVector' chr "/tmp/...gpkg"
```

Running the above line of code will save our two input objects into temporary .gpkg files, run the selected algorithm on them, and return a temporary .gpkg file as the output. The **qgisprocess** package stores the `qgis_run_algorithm()` result as a list containing, in this case, a path to the output file. We can either read this file back into R with `read_sf()` (e.g., `union_sf = read_sf(union[[1]])`) or directly with `st_as_sf()`:

```
union_sf = st_as_sf(union)
```

Note that the QGIS union operation merges the two input layers into one layer by using the intersection and the symmetrical difference of the two input layers (which, by the way, is also the default when doing a union operation in GRASS

GIS and SAGA). This is **not** the same as `st_union(incongr_wgs, aggzone_wgs)` (see the Exercises)!

The result, `union_sf`, is a multipolygon with a larger number of features than two input objects. Notice, however, that many of these polygons are small and do not represent real areas but are rather a result of our two datasets having a different level of detail. These artifacts of error are called sliver polygons (see red-colored polygons in the left panel of Figure 10.2). One way to identify slivers is to find polygons with comparatively very small areas, here, e.g., 25000 m^2, and next remove them. Let's search for an appropriate algorithm.

```
qgis_search_algorithms("clean")
#> # A tibble: 1 × 5
#>   provider provider_title group        algorithm      algorithm_title
#>   <chr>    <chr>          <chr>        <chr>          <chr>
#> 1 grass    GRASS          Vector (v.*) grass:v.clean v.clean
```

This time the found algorithm, `v.clean`, is not included in QGIS, but GRASS GIS. GRASS GIS's `v.clean` is a powerful tool for cleaning topology of spatial vector data. Importantly, we can use it through **qgisprocess**.

The GRASS GIS provider in QGIS was called `grass7` until QGIS version 3.34. Thus, if you have an older QGIS version, you must prefix the algorithms with `grass7` instead of `grass`.

Similar to the previous step, we should start by looking at this algorithm's help.

```
qgis_show_help("grass:v.clean")
```

We have omitted the output here, because the help text is quite long and contains a lot of arguments.[5] This is because `v.clean` is a multi-tool – it can clean different types of geometries and solve different types of topological problems. For this example, let's focus on just a few arguments, however, we encourage you to visit this algorithm's documentation to learn more about `v.clean` capabilities.

```
qgis_get_argument_specs("grass:v.clean") |>
  select(name, description) |>
  slice_head(n = 4)
#> # A tibble: 4 × 2
#>   name      description
```

[5] Also note that these arguments, contrary to the QGIS's ones, are in lowercase.

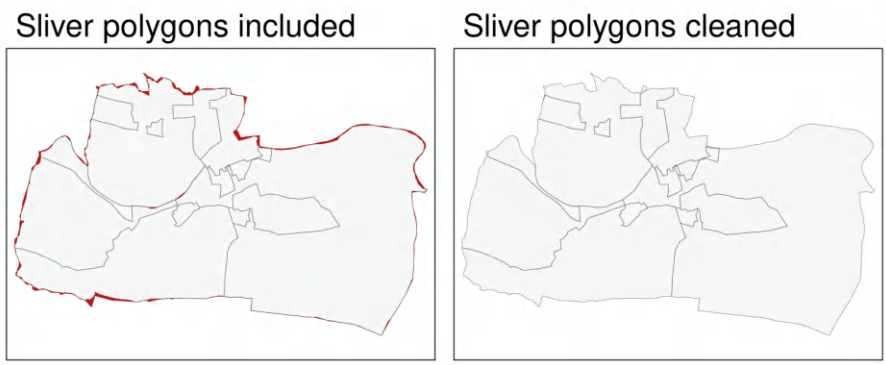

FIGURE 10.2 Sliver polygons colored in red (left panel). Cleaned polygons (right panel).

```
#>   <chr>     <chr>
#> 1 input     Layer to clean
#> 2 type      Input feature type
#> 3 tool      Cleaning tool
#> 4 threshold Threshold (comma separated for each tool)
```

The main argument for this algorithm is `input` – our vector object. Next, we need to select a tool – a cleaning method. [6] About a dozen tools exist in `v.clean` allowing to remove duplicate geometries, remove small angles between lines, or remove small areas, among others. In this case, we are interested in the latter tool, `rmarea`. Several of the tools, `rmarea` included, expect an additional argument `threshold`, whose behavior depends on the selected tool. In our case, the `rmarea` tool removes all areas smaller or equal to a provided `threshold`. Note that the threshold must be specified in square meters regardless of the coordinate reference system of the input layer.

Let's run this algorithm and convert its output into a new `sf` object `clean_sf`.

```
clean = qgis_run_algorithm("grass:v.clean",
  input = union_sf,
  tool = "rmarea", threshold = 25000
)
clean_sf = st_as_sf(clean)
```

The result, the right panel of Figure 10.2, looks as expected – sliver polygons are now removed.

[6] It is also possible to select several tools, which will then be executed sequentially.

10.2.2 Raster data

Digital elevation models (DEMs) contain elevation information for each raster cell. They are used for many purposes, including satellite navigation, water flow models, surface analysis, or visualization. Here, we are interested in deriving new information from a DEM raster that could be used as predictors for statistical learning. Various terrain parameters can be helpful, for example, for the prediction of landslides (see Chapter 12).

For this section, we will use `dem.tif` – a digital elevation model of the Mongón study area (downloaded from the Land Process Distributed Active Archive Center, see also `?dem.tif`). It has a resolution of about 30 x 30 meters and uses a projected CRS.

```
library(qgisprocess)
library(terra)
dem = system.file("raster/dem.tif", package = "spDataLarge")
```

The **terra** package's `terrain()` command already allows the calculation of several fundamental topographic characteristics such as slope, aspect, TPI (*Topographic Position Index*), TRI (*Topographic Ruggedness Index*), roughness, and flow directions. However, GIS programs offer many more terrain characteristics, some of which can be more suitable in certain contexts. For example, the topographic wetness index (TWI) was found useful in studying hydrological and biological processes (Sørensen et al., 2006). Let's search the algorithm list for this index using `"wetness"` as keyword.

```
qgis_search_algorithms("wetness") |>
  dplyr::select(provider_title, algorithm) |>
  head(2)
#> # A tibble: 2 × 2
#>   provider_title algorithm
#>   <chr>          <chr>
#> 1 SAGA Next Gen  sagang:sagawetnessindex
#> 2 SAGA Next Gen  sagang:topographicwetnessindexonestep
```

An output of the above code suggests that the desired algorithm exists in the SAGA software.[7] Though SAGA is a hybrid GIS, its main focus has been on raster processing, and here, particularly on digital elevation models (soil properties, terrain attributes, climate parameters). Hence, SAGA is especially good at the fast processing of large (high-resolution) raster datasets (Conrad et al., 2015).

[7] TWI can be also calculated using the `r.topidx` GRASS GIS function.

The "sagang:sagawetnessindex" algorithm is actually a modified TWI, that results in a more realistic soil moisture potential for the cells located in valley floors (Böhner and Selige, 2006).

```
qgis_show_help("sagang:sagawetnessindex")
```

Here, we stick with the default values for all arguments. Therefore, we only have to specify one argument – the input DEM. Of course, when applying this algorithm you should make sure that the parameter values are in correspondence with your study aim.[8]

Before running the SAGA algorithm from within QGIS, we change the default raster output format from .tif to SAGA's native raster format .sdat. Hence, all output rasters that we do not specify ourselves will from now on be written to the .sdat format. Depending on the software versions (SAGA, GDAL) you are using, this might not be necessary, but often enough this will save you trouble when trying to read-in output rasters created with SAGA.

```
options(qgisprocess.tmp_raster_ext = ".sdat")
dem_wetness = qgis_run_algorithm("sagang:sagawetnessindex",
  DEM = dem
)
```

"sagang:sagawetnessindex" returns not one but four rasters – catchment area, catchment slope, modified catchment area, and topographic wetness index. We can read a selected output by providing an output name in the qgis_as_terra() function. And since we are done with the SAGA processing from within QGIS, we change the raster output format back to .tif.

```
dem_wetness_twi = qgis_as_terra(dem_wetness$TWI)
# plot(dem_wetness_twi)
options(qgisprocess.tmp_raster_ext = ".tif")
```

You can see the TWI map in the left panel of Figure 10.3. The topographic wetness index is unitless: its low values represent areas that will not accumulate water, while higher values show areas that will accumulate water at increasing levels.

Information from digital elevation models can also be categorized, for example, to geomorphons – the geomorphological phenotypes consisting of ten classes that represent terrain forms, such as slopes, ridges, or valleys (Jasiewicz and Stepinski, 2013). These phenotypes are used in many studies, including landslide susceptibility, ecosystem services, human mobility, and digital soil mapping.

[8] The additional arguments of "sagang:sagawetnessindex" are well-explained at https://gis.stackexchange.com/a/323454/20955.

The original implementation of the geomorphons' algorithm was created in GRASS GIS, and we can find it in the **qgisprocess** list as `"grass:r.geomorphon"`:

```
qgis_search_algorithms("geomorphon")
#> [1] "grass:r.geomorphon" "sagang:geomorphons"
qgis_show_help("grass:r.geomorphon")
# output not shown
```

Calculation of geomorphons requires an input DEM (`elevation`) and can be customized with a set of optional arguments. It includes, `search` – a length for which the line-of-sight is calculated, and `-m` – a flag specifying that the search value will be provided in meters (and not the number of cells). More information about additional arguments can be found in the original paper and the GRASS GIS documentation.

```
dem_geomorph = qgis_run_algorithm("grass:r.geomorphon",
  elevation = dem,
  `-m` = TRUE, search = 120
)
```

Our output, `dem_geomorph$forms`, contains a raster file with ten categories – each representing a terrain form. We can read it into R with `qgis_as_terra()`, and then visualize it (Figure 10.3, right panel) or use it in our subsequent calculations.

```
dem_geomorph_terra = qgis_as_terra(dem_geomorph$forms)
```

Interestingly, there are connections between some geomorphons and the TWI values, as shown in Figure 10.3. The largest TWI values mostly occur in valleys and hollows, while the lowest values are seen, as expected, on ridges.

10.3 SAGA

The System for Automated Geoscientific Analyses (SAGA; Table 10.1) provides the possibility to execute SAGA modules via the command-line interface (`saga_cmd.exe` under Windows and just `saga_cmd` under Linux) (see the SAGA wiki on modules). In addition, there is a Python interface (SAGA Python API). **Rsagacmd** uses the former to run SAGA from within R.

We will use **Rsagacmd** in this section to delineate areas with similar values of the normalized difference vegetation index (NDVI) of the Mongón study area

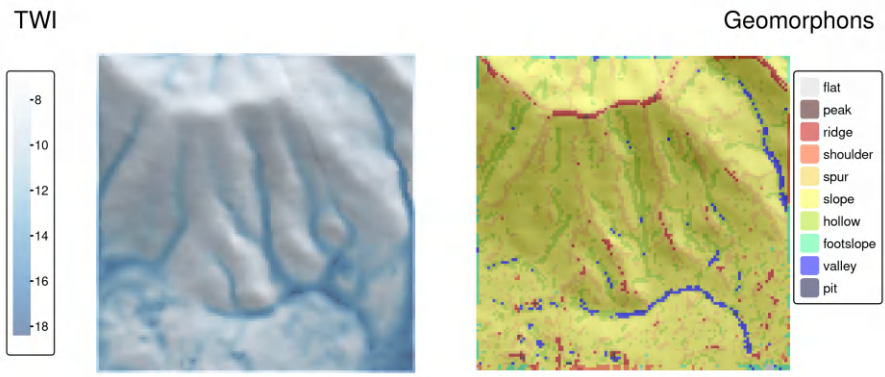

FIGURE 10.3 Topographic wetness index (TWI, left panel) and geomorphons (right panel) derived for the Mongón study area.

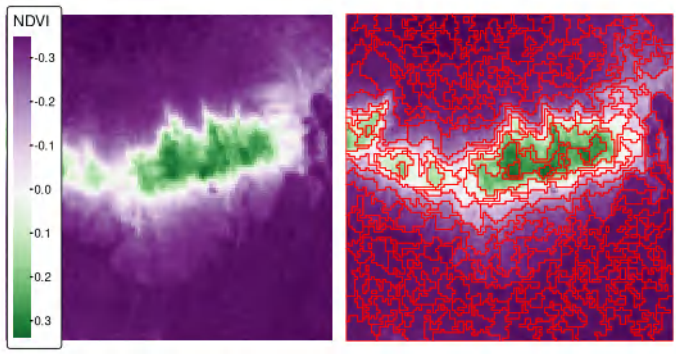

FIGURE 10.4 Normalized difference vegetation index (NDVI, left panel) and NDVI-based segments derived using the seeded region growing algorithm for the Mongón study area.

in Peru from September in the year 2000 (Figure 10.4, left panel) by using a seeded region growing algorithm from SAGA.[9]

```
ndvi = rast(system.file("raster/ndvi.tif", package = "spDataLarge"))
```

To start using **Rsagacmd**, we need to run the `saga_gis()` function. It serves two main purposes:

[9]Read Section 4.3.3 on details of how to calculate NDVI from a remote-sensing image.

- It dynamically[10] creates a new object that contains links to all valid SAGA libraries and tools
- It sets up general package options, such as `raster_backend` (R package to use for handling raster data), `vector_backend` (R package to use for handling vector data), and `cores` (a maximum number of CPU cores used for processing, default: all)

```
library(Rsagacmd)
saga = saga_gis(raster_backend = "terra", vector_backend = "sf")
```

Our `saga` object contains connections to all of the available SAGA tools. It is organized as a list of libraries (groups of tools), and inside of a library it has a list of tools. We can access any tool with the `$` sign (remember to use TAB for autocompletion).

The seeded region growing algorithm works in two main steps (Adams and Bischof, 1994; Böhner et al., 2006). First, initial cells ("seeds") are generated by finding the cells with the smallest variance in local windows of a specified size. Second, the region growing algorithm is used to merge neighboring pixels of the seeds to create homogeneous areas.

```
sg = saga$imagery_segmentation$seed_generation
```

In the above example, we first pointed to the `imagery_segmentation` library and then its `seed_generation` tool. We also assigned it to the `sg` object, not to retype the whole tool code in our next steps.[11] If we just type `sg`, we will get a quick summary of the tool and a data frame with its parameters, descriptions, and defaults. You may also use `tidy(sg)` to extract just the parameters' table. The `seed_generation` tool takes a raster dataset as its first argument (`features`); optional arguments include `band_width` that specifies the size of initial polygons.

```
ndvi_seeds = sg(ndvi, band_width = 2)
# plot(ndvi_seeds$seed_grid)
```

Our output is a list of three objects: `variance` – a raster map of local variance, `seed_grid` – a raster map with the generated seeds, and `seed_points` – a spatial vector object with the generated seeds.

The second SAGA tool we use is `seeded_region_growing`.[12] The `seeded_region_growing` tool requires two inputs: our `seed_grid` calculated

[10] This means that the available libraries will depend on the installed SAGA version.

[11] You can read more about the tool at https://saga-gis.sourceforge.io/saga_tool_doc/8.3.0/imagery_segmentation_2.html.

[12] You can read more about the tool at https://saga-gis.sourceforge.io/saga_tool_doc/8.3.0/imagery_segmentation_3.html.

in the previous step and the `ndvi` raster object. Additionally, we can specify several parameters, such as `normalize` to standardize the input features, `neighbour` (4- or 8-neighborhood), and `method`. The last parameter can be set to either `0` or `1` (region growing is based on raster cells' values and their positions or just the values). For a more detailed description of the method, see Böhner et al. (2006).

Here, we will only change `method` to `1`, meaning that our output regions will be created only based on the similarity of their NDVI values.

```
srg = saga$imagery_segmentation$seeded_region_growing
ndvi_srg = srg(ndvi_seeds$seed_grid, ndvi, method = 1)
plot(ndvi_srg$segments)
```

The tool returns a list of three objects: `segments`, `similarity`, `table`. The `similarity` object is a raster showing similarity between the seeds and the other cells, and `table` is a data frame storing information about the input seeds. Finally, `ndvi_srg$segments` is a raster with our resulting areas (Figure 10.4, right panel). We can convert it into polygons with `as.polygons()` and `st_as_sf()` (Section 6.5).

```
ndvi_segments = ndvi_srg$segments |>
  as.polygons() |>
  st_as_sf()
```

The resulting polygons (segments) represent areas with similar values. They can also be further aggregated into larger polygons using various techniques, such as clustering (e.g., k-means), regionalization (e.g., SKATER) or supervised classification methods. You can try to do it in the Exercises.

R also has other tools to achieve the goal of creating polygons with similar values (so-called segments). It includes the **SegOptim** package (Gonçalves et al., 2019) that allows running several image segmentation algorithms and **supercells** package (Nowosad and Stepinski, 2022) that implements superpixels algorithm SLIC to work with geospatial data.

10.4 GRASS GIS

The U.S. Army - Construction Engineering Research Laboratory (USA-CERL) created the core of the Geographical Resources Analysis Support System (GRASS GIS) (Table 10.1; Neteler and Mitasova (2008)) from 1982 to 1995. Academia continued this work since 1997. Similar to SAGA, GRASS GIS focused on raster processing in the beginning while, only later since GRASS GIS 6.0, adding advanced vector functionality (Bivand et al., 2013).

GRASS GIS stores the input data in an internal database. With regard to vector data, GRASS GIS is by default a topological GIS, i.e., it only stores the geometry of adjacent features once. SQLite is the default database driver for vector attribute management, and attributes are linked to the geometry, i.e., to the GRASS GIS database, via keys (GRASS GIS vector management).

Before one can use GRASS GIS, one has to set up the GRASS GIS database (also from within R), and users might find this process a bit intimidating in the beginning. First of all, the GRASS GIS database requires its own directory, which, in turn, contains a location (see the GRASS GIS Database help pages at grass.osgeo.org for further information). The location stores the geodata for one project or one area. Within one location, several mapsets can exist that typically refer to different users or different tasks. Each location also has a PERMANENT mapset – a mandatory mapset that is created automatically. In order to share geographic data with all users of a project, the database owner can add spatial data to the PERMANENT mapset. In addition, the PERMANENT mapset stores the projection, the spatial extent and the default resolution for raster data. So, to sum it all up – the GRASS GIS database may contain many locations (all data in one location have the same CRS), and each location can store many mapsets (groups of datasets). Please refer to Neteler and Mitasova (2008) and the GRASS GIS quick start for more information on the GRASS GIS spatial database system. To quickly use GRASS GIS from within R, we will use the **link2GI** package, however, one can also set up the GRASS GIS database step-by-step. See GRASS within R for how to do so. Please note that the code instructions in the following paragraphs might be hard to follow when using GRASS GIS for the first time but by running through the code line-by-line and by examining the intermediate results, the reasoning behind it should become even clearer.

Here, we introduce **rgrass** with one of the most interesting problems in GI-Science: the traveling salesman problem. Suppose a traveling salesman would like to visit 24 customers. Additionally, the salesman would like to start and finish the journey at home which makes a total of 25 locations while covering the shortest distance possible. There is a single best solution to this problem; however, to check all of the possible solutions, it is (mostly) impossible for modern computers (Longley, 2015). In our case, the number of possible solutions correspond to `(25 - 1)! / 2`, i.e., the factorial of 24 divided by 2 (since we do not differentiate between forward or backward direction). Even if one iteration can be done in a nanosecond, this still corresponds to 9837145 years. Luckily, there are clever, almost optimal solutions which run in a tiny fraction of this inconceivable amount of time. GRASS GIS provides one of these solutions (for more details, see v.net.salesman). In our use case, we would like to find the shortest path between the first 25 bicycle stations (instead of customers) on London's streets (and we simply assume that the first bike station corresponds to the home of our traveling salesman).

```
data("cycle_hire", package = "spData")
points = cycle_hire[1:25, ]
```

Aside from the cycle hire points data, we need a street network for this area. We can download it from OpenStreetMap with the help of the **osmdata** package (see also Section 8.5). To do this, we constrain the query of the street network (in OSM language called "highway") to the bounding box of `points`, and attach the corresponding data as an `sf`-object. `osmdata_sf()` returns a list with several spatial objects (points, lines, polygons, etc.), but here, we only keep the line objects with their related ids.[13]

```
library(osmdata)
b_box = st_bbox(points)
london_streets = opq(b_box) |>
  add_osm_feature(key = "highway") |>
  osmdata_sf()
london_streets = london_streets[["osm_lines"]]
london_streets = select(london_streets, osm_id)
```

Now that we have the data, we can go on and initiate a GRASS GIS session. Luckily, `linkGRASS()` of the **link2GI** packages lets one set up the GRASS GIS environment with just one line of code. The only thing you need to provide is a spatial object which determines the projection and the extent of the spatial database. First, `linkGRASS()` finds all GRASS GIS installations on your computer. Since we have set `ver_select` to `TRUE`, we can interactively choose one of the found GRASS GIS-installations. If there is just one installation, the `linkGRASS()` automatically chooses it. Second, `linkGRASS()` establishes a connection to GRASS GIS.

```
library(rgrass)
link2GI::linkGRASS(london_streets, ver_select = TRUE)
```

Before we can use GRASS GIS geoalgorithms, we also need to add data to GRASS GIS's spatial database. Luckily, the convenience function `write_VECT()` does this for us. (Use `write_RAST()` for raster data.) In our case, we add the street and cycle hire point data while using only the first attribute column, and name them as `london_streets` and `points` in GRASS GIS.

```
write_VECT(terra::vect(london_streets), vname = "london_streets")
write_VECT(terra::vect(points[, 1]), vname = "points")
```

[13] As a convenience to the reader, one can attach `london_streets` to the global environment using `data("london_streets", package = "spDataLarge")`.

The **rgrass** package expects its inputs and gives its outputs as **terra** objects. Therefore, we need to convert our `sf` spatial vectors to **terra**'s `SpatVector`s using the `vect()` function to be able to use `write_VECT()`.[14]

Now, both datasets exist in the GRASS GIS database. To perform our network analysis, we need a topologically clean street network. GRASS GIS's `"v.clean"` takes care of the removal of duplicates, small angles and dangles, among others. Here, we break lines at each intersection to ensure that the subsequent routing algorithm can actually turn right or left at an intersection, and save the output in a GRASS GIS object named `streets_clean`.

```
execGRASS(
  cmd = "v.clean", input = "london_streets", output = "streets_clean",
  tool = "break", flags = "overwrite"
)
```

To learn about the possible arguments and flags of the GRASS GIS modules, you can use the `help` flag. For example, try `execGRASS("g.region", flags = "help")`.

It is likely that a few of our cycling station points will not lie exactly on a street segment. However, to find the shortest route between them, we need to connect them to the nearest streets segment. `"v.net"`'s connect-operator does exactly this. We save its output in `streets_points_con`.

```
execGRASS(
  cmd = "v.net", input = "streets_clean", output = "streets_points_con",
  points = "points", operation = "connect", threshold = 0.001,
  flags = c("overwrite", "c")
)
```

The resulting clean dataset serves as input for the `"v.net.salesman"` algorithm, which finally finds the shortest route between all cycle hire stations. One of its arguments is `center_cats`, which requires a numeric range as input. This range represents the points for which a shortest route should be calculated. Since we would like to calculate the route for all cycle stations, we set it to `1-25`. To access the GRASS GIS help page of the traveling salesman algorithm, run `execGRASS("g.manual", entry = "v.net.salesman")`.

[14] You can learn more how to convert between spatial classes in R by reading the (Conversions between different spatial classes in R)[https://geocompx.org/post/2021/spatial-classes-conversion/] blog post and the (Coercion between object formats)[https://CRAN.R-project.org/package=rgrass/vignettes/coerce.html] vignette.

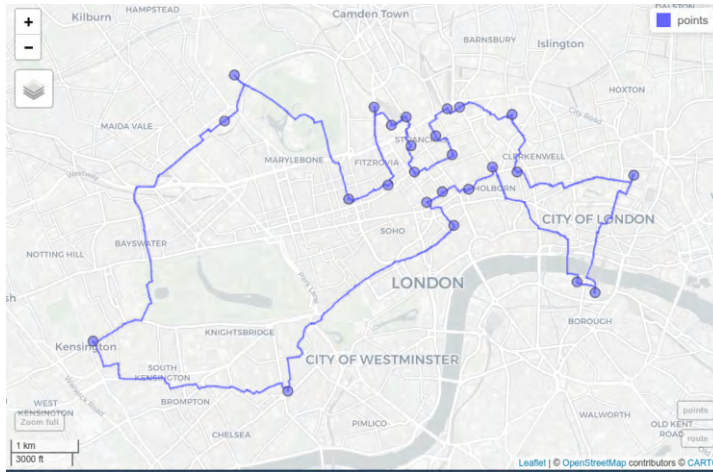

FIGURE 10.5 Shortest route (blue line) between 24 cycle hire stations (blue dots) on the OSM street network of London.

```
execGRASS(
  cmd = "v.net.salesman", input = "streets_points_con",
  output = "shortest_route", center_cats = paste0("1-", nrow(points)),
  flags = "overwrite"
)
```

To see our result, we read the result into R, convert it into an `sf`-object keeping only the geometry, and visualize it with the help of the **mapview** package (Figure 10.5 and Section 9.4).

```
route = read_VECT("shortest_route") |>
  st_as_sf() |>
  st_geometry()
mapview::mapview(route) + points
```

There are a few important considerations to note in the process:

- We could have used GRASS GIS's spatial database which allows faster processing. That means we have only exported geographic data at the beginning. Then we created new objects but only imported the final result back into R. To find out which datasets are currently available, run `execGRASS("g.list", type = "vector,raster", flags = "p")`.
- We could have also accessed an already existing GRASS GIS spatial database from within R. Prior to importing data into R, you might want to perform some (spatial) subsetting. Use `"v.select"` and `"v.extract"` for vector data.

`"db.select"` lets you select subsets of the attribute table of a vector layer without returning the corresponding geometry.
- You can also start R from within a running GRASS GIS session (for more information, please refer to Bivand et al., 2013).
- Refer to the excellent GRASS GIS online help or `execGRASS("g.manual", flags = "i")` for more information on each available GRASS GIS geoalgorithm.

10.5 When to use what?

To recommend a single R-GIS interface is hard since the usage depends on personal preferences, the tasks at hand, and your familiarity with different GIS software packages which in turn probably depends on your domain. As mentioned previously, SAGA is especially good at the fast processing of large (high-resolution) raster datasets and frequently used by hydrologists, climatologists and soil scientists (Conrad et al., 2015). GRASS GIS, on the other hand, is the only GIS presented here supporting a topologically based spatial database which is especially useful for network analyses but also simulation studies. QGISS is much more user-friendly compared to GRASS GIS and SAGA, especially for first-time GIS users, and probably the most popular open-source GIS. Therefore, **qgisprocess** is an appropriate choice for most use cases. Its main advantages are:

- A unified access to several GIS, and therefore the provision of >1000 geoalgorithms (Table 10.1) including duplicated functionality, e.g., you can perform overlay-operations using QGIS-, SAGA- or GRASS GIS-geoalgorithms
- Automatic data format conversions (SAGA uses `.sdat` grid files and GRASS GIS uses its own database format, but QGIS will handle the corresponding conversions)
- Its automatic passing of geographic R objects to QGIS geoalgorithms and back into R
- Convenience functions to support named arguments and automatic default value retrieval (as inspired by **rgrass**)

By all means, there are use cases when you certainly should use one of the other R-GIS bridges. Though QGIS is the only GIS providing a unified interface to several GIS software packages, it only provides access to a subset of the corresponding third-party geoalgorithms (for more information, please refer to Muenchow et al. (2017)). Therefore, to use the complete set of SAGA and GRASS GIS functions, stick with **Rsagacmd** and **rgrass**. In addition, if you would like to run simulations with the help of a geodatabase (Krug et al., 2010), use **rgrass** directly since **qgisprocess** always starts a new GRASS GIS session for each call. Finally, if you need topological correct data and/or spatial

database management functionality such as multi-user access, we recommend the usage of GRASS GIS.

Please note that there are a number of further GIS software packages that have a scripting interface but for which there is no dedicated R package that accesses these: gvSig, OpenJump, and the Orfeo Toolbox.[15]

10.6 Bridges to GDAL

As discussed in Chapter 8, GDAL is a low-level library that supports many geographic data formats. GDAL is so effective that most GIS programs use GDAL in the background for importing and exporting geographic data, rather than reinventing the wheel and using bespoke read-write code. But GDAL offers more than data I/O. It has geoprocessing tools for vector and raster data, functionality to create tiles for serving raster data online, and rapid rasterization of vector data. Since GDAL is a command-line tool, all its commands can be accessed from within R via the `system()` command.

The code chunk below demonstrates this functionality: `linkGDAL()` searches the computer for a working GDAL installation and adds the location of the executable files to the PATH variable, allowing GDAL to be called (usually only needed under Windows).

```
link2GI::linkGDAL()
```

Now we can use the `system()` function to call any of the GDAL tools. For example, `ogrinfo` provides metadata of a vector dataset. Here we will call this tool with two additional flags: `-al` to list all features of all layers and `-so` to get a summary only (and not a complete geometry list):

```
our_filepath = system.file("shapes/world.gpkg", package = "spData")
cmd = paste("ogrinfo -al -so", our_filepath)
system(cmd)
#> INFO: Open of `.../spData/shapes/world.gpkg'
#>       using driver `GPKG' successful.
#>
#> Layer name: world
#> Geometry: Multi Polygon
#> Feature Count: 177
```

[15] Please note that **link2GI** provides a partial integration with the Orfeo Toolbox and that you can also access the Orfeo Toolbox geoalgorithms via **qgisprocess**. Note also that TauDEM can be accessed from R with package **traudem**.

```
#> Extent: (-180.000000, -89.900000) - (179.999990, 83.645130)
#> Layer SRS WKT:
#> ...
```

Other commonly used GDAL tools include:

- `gdalinfo`: provides metadata of a raster dataset
- `gdal_translate`: converts between different raster file formats
- `ogr2ogr`: converts between different vector file formats
- `gdalwarp`: reprojects, transforms, and clips raster datasets
- `gdaltransform`: transforms coordinates

Visit https://gdal.org/programs/ to see the complete list of GDAL tools and to read their help files.

The 'link' to GDAL provided by **link2GI** could be used as a foundation for doing more advanced GDAL work from the R or system CLI. TauDEM (https://hydrology.usu.edu/taudem/) and the Orfeo Toolbox (https://www.orfeo-toolbox.org/) are other spatial data processing libraries/programs offering a command-line interface – the above example shows how to access these libraries from the system command line via R. This in turn could be the starting point for creating a proper interface to these libraries in the form of new R packages.

Before diving into a project to create a new bridge, however, it is important to be aware of the power of existing R packages and that `system()` calls may not be platform-independent (they may fail on some computers). On the other hand, **sf** and **terra** brings most of the power provided by GDAL, GEOS and PROJ to R via the R/C++ interface provided by **Rcpp**, which avoids `system()` calls.[16]

10.7 Bridges to spatial databases

Spatial database management systems (spatial DBMSs) store spatial and non-spatial data in a structured way. They can organize large collections of data into related tables (entities) via unique identifiers (primary and foreign keys) and implicitly via space (think for instance of a spatial join). This is useful because geographic datasets tend to become big and messy quite quickly. Databases enable storing and querying large datasets efficiently based on spatial and non-spatial fields, and provide multi-user access and topology support.

[16] There are also **vapour** and **gdalraster** that provide low-level interfaces to GDAL.

The most important open source spatial database is PostGIS (Obe and Hsu, 2015).[17] R bridges to spatial DBMSs such as PostGIS are important, allowing access to huge data stores without loading several gigabytes of geographic data into RAM, and likely crashing the R session. The remainder of this section shows how PostGIS can be called from R, based on "Hello real-world" from *PostGIS in Action, Second Edition* (Obe and Hsu, 2015).[18]

The subsequent code requires a working internet connection, since we are accessing a PostgreSQL/PostGIS database which is living in the QGIS Cloud (https://qgiscloud.com/).[19] Our first step here is to create a connection to a database by providing its name, host name, and user information.

```
library(RPostgreSQL)
conn = dbConnect(
  drv = PostgreSQL(),
  dbname = "rtafdf_zljbqm", host = "db.qgiscloud.com",
  port = "5432", user = "rtafdf_zljbqm", password = "d3290ead"
)
```

Our new object, `conn`, is just an established link between our R session and the database. It does not store any data.

Often the first question is, 'which tables can be found in the database?'. This can be answered with `dbListTables()` as follows:

```
dbListTables(conn)
#> [1] "spatial_ref_sys" "topology"        "layer"           "restaurants"
#> [5] "highways"
```

The answer is five tables. Here, we are only interested in the `restaurants` and the `highways` tables. The former represents the locations of fast-food restaurants in the US, and the latter are principal US highways. To find out about attributes available in a table, we can run `dbListFields`:

```
dbListFields(conn, "highways")
#> [1] "qc_id"        "wkb_geometry" "gid"          "feature"
#> [5] "name"         "state"
```

[17] SQLite/SpatiaLite are certainly also important, but implicitly we have already introduced this approach since GRASS GIS is using SQLite in the background (see Section 10.4).

[18] Thanks to Manning Publications, Regina Obe and Leo Hsu for permission to use this example.

[19] QGIS Cloud lets you store geographic data and maps in the cloud. In the background, it uses QGIS Server and PostgreSQL/PostGIS. This way, the reader can follow the PostGIS example without the need to have PostgreSQL/PostGIS installed on a local machine. Thanks to the QGIS Cloud team for hosting this example.

Now, as we know the available datasets, we can perform some queries – ask the database some questions. The query needs to be provided in a language understandable by the database – usually, it is SQL. The first query will select `US Route 1` in the state of Maryland (`MD`) from the `highways` table. Note that `read_sf()` allows us to read geographic data from a database if it is provided with an open connection to a database and a query. Additionally, `read_sf()` needs to know which column represents the geometry (here: `wkb_geometry`).

```
query = paste(
  "SELECT *",
  "FROM highways",
  "WHERE name = 'US Route 1' AND state = 'MD';"
)
us_route = read_sf(conn, query = query, geom = "wkb_geometry")
```

This results in an **sf**-object named `us_route` of type `MULTILINESTRING`.

As we mentioned before, it is also possible to not only ask non-spatial questions, but also query datasets based on their spatial properties. To show this, the next example adds a 35-kilometer (35,000 m) buffer around the selected highway (Figure 10.6).

```
query = paste(
  "SELECT ST_Union(ST_Buffer(wkb_geometry, 35000))::geometry",
  "FROM highways",
  "WHERE name = 'US Route 1' AND state = 'MD';"
)
buf = read_sf(conn, query = query)
```

Note that this was a spatial query using functions (`ST_Union()`, `ST_Buffer()`) you should be already familiar with. You find them also in the **sf**-package, though here they are written in lowercase characters (`st_union()`, `st_buffer()`). In fact, function names of the **sf** package largely follow the PostGIS naming conventions.[20]

The last query will find all Hardee's restaurants (`HDE`) within the 35-km buffer zone (Figure 10.6).

```
query = paste(
  "SELECT *",
  "FROM restaurants r",
  "WHERE EXISTS (",
  "SELECT gid",
```

[20] The prefix `st` stands for space/time.

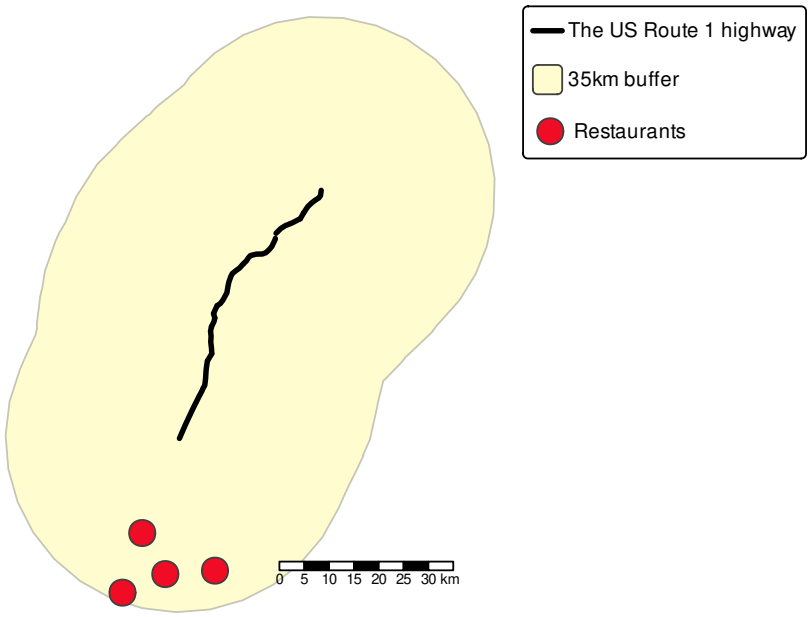

FIGURE 10.6 Visualization of the output of previous PostGIS commands showing the highway (black line), a buffer (light yellow) and four restaurants (red points) within the buffer.

```
  "FROM highways",
  "WHERE",
  "ST_DWithin(r.wkb_geometry, wkb_geometry, 35000) AND",
  "name = 'US Route 1' AND",
  "state = 'MD' AND",
  "r.franchise = 'HDE');"
)
hardees = read_sf(conn, query = query)
```

Please refer to Obe and Hsu (2015) for a detailed explanation of the spatial SQL query. Finally, it is good practice to close the database connection as follows:[21]

```
RPostgreSQL::postgresqlCloseConnection(conn)
```

Unlike PostGIS, **sf** only supports spatial vector data. To query and manipulate raster data stored in a PostGIS database, use the **rpostgis** package (Bucklin

[21] It is important to close the connection here because QGIS Cloud (free version) allows only ten concurrent connections.

and Basille, 2018) and/or use command line tools such as `rastertopgsql` which comes as part of the PostGIS installation.

This subsection is only a brief introduction to PostgreSQL/PostGIS. Nevertheless, we would like to encourage the practice of storing geographic and non-geographic data in a spatial DBMS while only attaching those subsets to R's global environment which are needed for further (geo-)statistical analysis. Please refer to Obe and Hsu (2015) for a more detailed description of the SQL queries presented and a more comprehensive introduction to PostgreSQL/PostGIS in general. PostgreSQL/PostGIS is a formidable choice as an open-source spatial database. But the same is true for the lightweight SQLite/SpatiaLite database engine and GRASS GIS which uses SQLite in the background (see Section 10.4).

If your datasets are too big for PostgreSQL/PostGIS and you require massive spatial data management and query performance, it may be worth exploring large-scale geographic querying on distributed computing systems. Such systems are outside the scope of this book, but it is worth mentioning that open source software providing this functionality exists. Prominent projects in this space include GeoMesa and Apache Sedona. The **apache.sedona** package provides an interface to the latter.

10.8 Bridges to cloud technologies and services

In recent years, cloud technologies have become more and more prominent on the internet. This also includes their use to store and process spatial data. Major cloud computing providers (Amazon Web Services, Microsoft Azure / Planetary Computer, Google Cloud Platform, and others) offer vast catalogs of open Earth observation data, such as the complete Sentinel-2 archive, on their platforms. We can use R and directly connect to and process data from these archives, ideally from a machine in the same cloud and region.

Three promising developments that make working with such image archives on cloud platforms *easier* and *more efficient* are the SpatioTemporal Asset Catalog (STAC), the cloud-optimized GeoTIFF (COG) image file format, and the concept of data cubes. Section 10.8.1 introduces these individual developments and briefly describes how they can be used from R.

Besides hosting large data archives, numerous cloud-based services to process Earth observation data have been launched during the last few years. It includes the OpenEO initiative – a unified interface between programming languages (including R) and various cloud-based services. You can find more information about OpenEO in Section 10.8.2.

10.8.1 STAC, COGs, and data cubes in the cloud

The SpatioTemporal Asset Catalog (STAC) is a general description format for spatiotemporal data that is used to describe a variety of datasets on cloud platforms including imagery, synthetic aperture radar (SAR) data, and point clouds. Besides simple static catalog descriptions, STAC-API presents a web service to query items (e.g., images) of catalogs by space, time, and other properties. In R, the **rstac** package (Simoes et al., 2021b) allows to connect to STAC-API endpoints and search for items. In the example below, we request all images from the Sentinel-2 Cloud-Optimized GeoTIFF (COG) dataset on Amazon Web Services that intersect with a predefined area and time of interest. The result contains all found images and their metadata (e.g., cloud cover) and URLs pointing to actual files on AWS.

```
library(rstac)
# Connect to the STAC-API endpoint for Sentinel-2 data
# and search for images intersecting our AOI
s = stac("https://earth-search.aws.element84.com/v0")
items = s |>
  stac_search(collections = "sentinel-s2-l2a-cogs",
              bbox = c(7.1, 51.8, 7.2, 52.8),
              datetime = "2020-01-01/2020-12-31") |>
  post_request() |>
  items_fetch()
```

Cloud storage differs from local hard disks and traditional image file formats do not perform well in cloud-based geoprocessing. Cloud-optimized GeoTIFF makes reading rectangular subsets of an image or reading images at lower resolution much more efficient. As an R user, you do not have to install anything to work with COGs because GDAL (and any package using it) can already work with COGs. However, keep in mind that the availability of COGs is a big plus while browsing through catalogs of data providers.

For larger areas of interest, requested images are still relatively difficult to work with: they may use different map projections, may spatially overlap, and their spatial resolution often depends on the spectral band. The **gdalcubes** package (Appel and Pebesma, 2019) can be used to abstract from individual images and to create and process image collections as four-dimensional data cubes.

The code below shows a minimal example to create a lower resolution (250 m) maximum NDVI composite from the Sentinel-2 images returned by the previous STAC-API search.

```
library(gdalcubes)
# Filter images by cloud cover and create an image collection object
cloud_filter = function(x) {
    x[["eo:cloud_cover"]] < 10
}
collection = stac_image_collection(items$features,
                                   property_filter = cloud_filter)
# Define extent, resolution (250m, daily) and CRS of the target data cube
v = cube_view(srs = "EPSG:3857", extent = collection, dx = 250, dy = 250,
              dt = "P1D") # "P1D" is an ISO 8601 duration string
# Create and process the data cube
cube = raster_cube(collection, v) |>
  select_bands(c("B04", "B08")) |>
  apply_pixel("(B08-B04)/(B08+B04)", "NDVI") |>
  reduce_time("max(NDVI)")
# gdalcubes_options(parallel = 8)
# plot(cube, zlim = c(0, 1))
```

To filter images by cloud cover, we provide a property filter function that is applied on each STAC result item while creating the image collection. The function receives available metadata of an image as input list and returns a single logical value such that only images for which the function yields TRUE will be considered. In this case, we ignore images with 10% or more cloud cover. For more details, please refer to this tutorial presented at OpenGeoHub summer school 2021.[22]

The combination of STAC, COGs, and data cubes forms a cloud-native workflow to analyze (large) collections of satellite imagery in the cloud. These tools already form a backbone, for example, of the **sits** package, which allows land use and land cover classification of big Earth observation data in R. The package builds EO data cubes from image collections available in cloud services and performs land classification of data cubes using various machine and deep learning algorithms. For more information about **sits**, visit https://e-sensing.github.io/sitsbook/ or read the related article (Simoes et al., 2021a).

10.8.2 openEO

OpenEO (Schramm et al., 2021) is an initiative to support interoperability among cloud services by defining a common language for processing the data. The initial idea has been described in an r-spatial.org blog post and aims at making it possible for users to change between cloud services easily with

[22] Another tool for working with STAC data in R is the **rsi** package that allows to get spatial data from STAC for a given time and location.

as little code changes as possible. The standardized processes use a multidimensional data cube model as an interface to the data. Implementations are available for eight different backends (see https://hub.openeo.org) to which users can connect with R, Python, JavaScript, QGIS, or a web editor and define (and chain) processes on collections. Since the functionality and data availability differs among the backends, the **openeo** R package (Lahn, 2021) dynamically loads available processes and collections from the connected backend. Afterwards, users can load image collections, apply and chain processes, submit jobs, and explore and plot results.

The following code will connect to the openEO platform backend, request available datasets, processes, and output formats, define a process graph to compute a maximum NDVI image from Sentinel-2 data, and finally execute the graph after logging in to the backend. The openEO platform backend includes a free tier, and registration is possible from existing institutional or internet platform accounts.

```
library(openeo)
con = connect(host = "https://openeo.cloud")
p = processes() # load available processes
collections = list_collections() # load available collections
formats = list_file_formats() # load available output formats
# Load Sentinel-2 collection
s2 = p$load_collection(id = "SENTINEL2_L2A",
                       spatial_extent = list(west = 7.5, east = 8.5,
                                             north = 51.1, south = 50.1),
                       temporal_extent = list("2021-01-01", "2021-01-31"),
                       bands = list("B04", "B08"))
# Compute NDVI vegetation index
compute_ndvi = p$reduce_dimension(data = s2, dimension = "bands",
                                  reducer = function(data, context) {
                                      (data[2] - data[1]) / (data[2] + data[1])
                                  })
# Compute maximum over time
reduce_max = p$reduce_dimension(data = compute_ndvi, dimension = "t",
                                reducer = function(x, y) {
                                    max(x)
                                })
# Export as GeoTIFF
result = p$save_result(reduce_max, formats$output$GTiff)
# Login, see https://docs.openeo.cloud/getting-started/r/#authentication
login(login_type = "oidc", provider = "egi",
      config = list(client_id = "...", secret = "..."))
# Execute processes
compute_result(graph = result, output_file = tempfile(fileext = ".tif"))
```

10.9 Exercises

E1. Compute global solar irradiation for an area of `system.file("raster/dem.tif", package = "spDataLarge")` for March 21 at 11:00 am using the `r.sun` GRASS GIS through **qgisprocess**.

E2. Compute catchment area and catchment slope of `system.file("raster/dem.tif", package = "spDataLarge")` using **Rsagacmd**.

E3. Continue working on the `ndvi_segments` object created in the SAGA section. Extract average NDVI values from the `ndvi` raster and group them into six clusters using `kmeans()`. Visualize the results.

E4. Attach `data(random_points, package = "spDataLarge")` and read `system.file("raster/dem.tif", package = "spDataLarge")` into R. Select a point randomly from `random_points` and find all `dem` pixels that can be seen from this point (hint: viewshed can be calculated using GRASS GIS). Visualize your result. For example, plot a hillshade, the digital elevation model, your viewshed output, and the point. Additionally, give `mapview` a try.

E5. Use `gdalinfo` via a system call for a raster file stored on a disk of your choice. What kind of information can you find there?

E6. Use `gdalwarp` to decrease the resolution of your raster file (for example, if the resolution is 0.5, change it into 1). Note: `-tr` and `-r` flags will be used in this exercise.

E7. Query all Californian highways from the PostgreSQL/PostGIS database living in the QGIS Cloud introduced in this chapter.

E8. The `ndvi.tif` raster (`system.file("raster/ndvi.tif", package = "spDataLarge")`) contains NDVI calculated for the Mongón study area based on Landsat data from September 22, 2000. Use **rstac**, **gdalcubes**, and **terra** to download Sentinel-2 images for the same area from 2020-08-01 to 2020-10-31, calculate its NDVI, and then compare it with the results from `ndvi.tif`.

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

11

Scripts, algorithms and functions

Prerequisites

This chapter has minimal software prerequisites, as it primarily uses base R. Only, the **sf** package is used to check the results of an algorithm we will develop to calculate the area of polygons. In terms of prior knowledge, this chapter assumes you have an understanding of the geographic classes introduced in Chapter 2 and how they can be used to represent a wide range of input file formats (see Chapter 8).

11.1 Introduction

Chapter 1 established that geocomputation is not only about using existing tools, but developing new ones, "in the form of shareable R scripts and functions". This chapter teaches these building blocks of reproducible code. It also introduces low-level geometric algorithms, of the type used in Chapter 10. Reading it should help you to understand how such algorithms work and to write code that can be used many times, by many people, on multiple datasets. The chapter cannot, by itself, make you a skilled programmer. Programming is hard and requires plenty of practice (Abelson et al., 1996):

> To appreciate programming as an intellectual activity in its own right you must turn to computer programming; you must read and write computer programs — many of them.

There are strong reasons for learning to program. Although this chapter does not teach programming itself — see resources such as Wickham (2019),

Gillespie and Lovelace (2016), and Xiao (2016) which teach programming in R and other languages — it does provide some starting points, focused on geometry data, that could form a good foundation for developing programming skills.

The chapter also demonstrates and highlights the importance of reproducibility. The advantages of reproducibility go beyond allowing others to replicate your work: reproducible code is often better in every way than code written to be run only once, including in terms of computational efficiency, 'scalability' (the capability of code to run on large datasets) and ease of adapting and maintaining it.

Scripts are the basis of reproducible R code, a topic covered in Section 11.2. Algorithms are recipes for modifying inputs using a series of steps, resulting in an output, as described in Section 11.3. To ease sharing and reproducibility, algorithms can be placed into functions. That is the topic of Section 11.4. The example of finding the centroid of a polygon will be used to tie these concepts together. Chapter 5 already introduced a centroid function `st_centroid()`, but this example highlights how seemingly simple operations are the result of comparatively complex code, affirming the following observation (Wise, 2001):

> One of the most intriguing things about spatial data problems is that things which appear to be trivially easy to a human being can be surprisingly difficult on a computer.

The example also reflects a secondary aim of the chapter which, following Xiao (2016), is "not to duplicate what is available out there, but to show how things out there work".

11.2 Scripts

If functions distributed in packages are the building blocks of R code, scripts are the glue that holds them together. Scripts should be stored and executed in a logical order to create reproducible workflows, manually or with workflow automation tools such as **targets** (Landau, 2021). If you are new to programming, scripts may seem intimidating when you first encounter them, but they are simply plain text files. Scripts are usually saved as a file with an extension representing the language they contain, such as `.py` for scripts written in Python or `.rs` for scripts written in Rust. R scripts should be saved with

a `.R` extension and named to reflect what they do. An example is `11-hello.R`, a script file stored in the `code` folder of the book's repository. `11-hello.R` is a simple script containing only two lines of code, one of which is a comment:

```
# Aim: provide a minimal R script
print("Hello geocompr")
```

The contents of this script are not particularly exciting, but they demonstrate the point: scripts do not need to be complicated. Saved scripts can be called and executed in their entirety from the R command line with the `source()` function, as demonstrated below. The output of this command shows that the comment is ignored but `print()` command is executed:

```
source("code/11-hello.R")
#> [1] "Hello geocompr"
```

You can also call R scripts from system shells such as `bash` and `PowerShell` as follows:

```
Rscript code/11-hello.R
```

If your `RScript` executable is configured so it is available, the above command will print `Hello geocompr` in the system shell. There are no strict rules on what can and cannot go into script files and nothing to prevent you from saving broken, non-reproducible code, so testing is important. Lines of code that do not contain valid R should be commented out, by adding a `#` to the start of the line, to prevent errors, as shown in line 1 of the `11-hello.R` script. It is worth following some basic rules:

- Write the script in order: just like the script of a film, scripts should have a clear order such as 'setup', 'data processing' and 'save results' (roughly equivalent to 'beginning', 'middle' and 'end' in a film).
- Add comments to the script so other people (and your future self) can understand it: at a minimum, a comment should state the purpose of the script (see Figure 11.1) and (for long scripts) divide it into sections. This can be done in RStudio, for example, with the shortcut `Ctrl+Shift+R`, which creates 'foldable' code section headings
- Above all, scripts should be reproducible: self-contained scripts that will work on any computer are more useful than scripts that only run on your computer, on a good day. This involves attaching required packages at the beginning, reading-in data from persistent sources (such as a reliable website) and ensuring that previous steps have been taken.[1]

[1] Prior steps can be referred to with a comment or with an if statement such as `if (!exists("x")) source("x.R")` (which would run the script file `x.R` if the object `x` is missing).

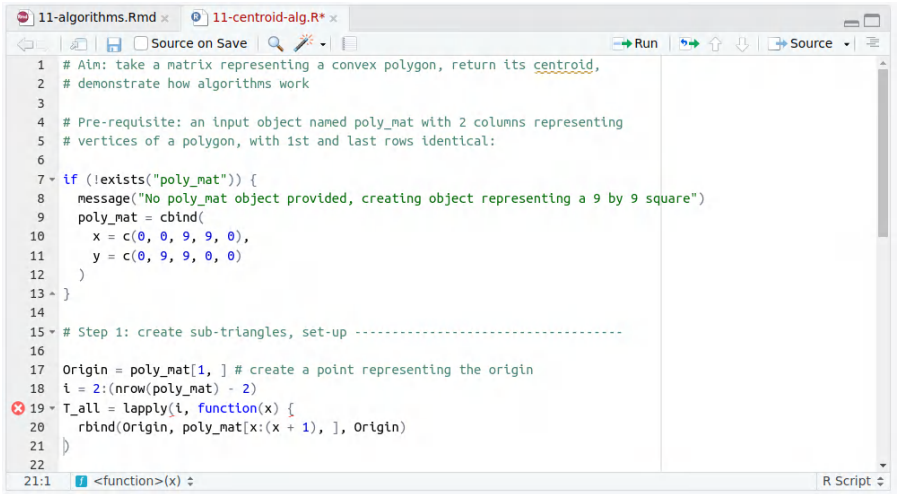

FIGURE 11.1 Code checking in RStudio. This example, from the script 11-centroid-alg.R, highlights an unclosed curly bracket on line 19.

Unless you organise your code into a package, it is hard to enforce reproducibility in R scripts, but there are tools that can help. By default, RStudio 'code checks' R scripts and underlines faulty code with a red wavy line, as shown in Figure 11.1. The **repex** package is another tool that can help with reproducibility.

A useful tool for reproducibility is the **reprex** package. Its main function `reprex()` tests lines of R code to check if they are reproducible, and it generates markdown output to facilitate communication on sites such as GitHub. See the webpage reprex.tidyverse.org for details.

The contents of this section apply to any type of R script. A particular consideration with scripts for geocomputation is that they tend to have external dependencies, such as the GDAL dependency needed for core R packages for working with geographic data, which we made heavy use of in Chapter 8 for data import and export. GIS software dependencies may be needed to run more specialist geoalgorithms, as outlined in Chapter 10. Scripts for working with geographic data also often require input datasets to be available in specific formats. Such dependencies should be mentioned as comments or suitable place in the project of which it is a part, or described as dependencies with tools such as the **renv** package or Docker.

'Defensive' programming techniques and good error messages can save time by checking for dependencies and communicating with users if certain requirements are not met. If statements, implemented with `if ()` in R, can be used to

send messages or run lines of code if, and only if, certain conditions are met. The following lines of code, for example, send a message to users if a certain file is missing:

```
if (!file.exists("required_geo_data.gpkg")) {
  message("No file, required_geo_data.gpkg is missing!")
}
#> No file, required_geo_data.gpkg is missing!
```

The work undertaken by the `11-centroid-alg.R` script is demonstrated in the reproducible example below, which creates a prerequisite object named `poly_mat`, representing a square with sides 9 units in length. This example shows that `source()` works with URLs, assuming you have an internet connection. If you do not, the same script can be called with `source("code/11-centroid-alg.R")`, assuming that you have previously downloaded the github.com/geocompx/geocompr repository and that you are running R from the `geocompr` folder.

```
poly_mat = cbind(
  x = c(0, 9, 9, 0, 0),
  y = c(0, 0, 9, 9, 0)
)
# Short URL to code/11-centroid-alg.R in the geocompr repo
source("https://t.ly/0nzj")


#> [1] "The area is: 81"
#> [1] "The coordinates of the centroid are: 4.5, 4.5"
```

11.3 Geometric algorithms

Algorithms can be understood as the computing equivalent of a baking recipe. They are a complete set of instructions which, when undertaken on the inputs result in useful/tasty outcomes. Inputs are ingredients such as flour and sugar in the case of baking, data and input parameters in the case of algorithms. And while tasty cakes may result from a baking recipe, successful algorithms should have computational outcomes with environmental/social/other benefits. Before diving into a reproducible example, the brief history below shows how algorithms relate to scripts (covered in Section 11.2) and functions (which can be used to generalize algorithms and make them more portable and easy-to-use, as we'll see in Section 11.4).

The word "algorithm" originated with the publication of an early math textbook in Baghdad in the year 825. The book was translated into Latin and

became so popular that the author's last name, al-Khwārizmī, "was immortalized as a scientific term: Al-Khwarizmi became Alchoarismi, Algorismi and, eventually, algorithm" (Bellos, 2011). In the computing age, algorithm refers to a series of steps that solves a problem, resulting in a predefined output. Inputs must be formally defined in a suitable data structure (Wise, 2001). Algorithms often start as flow charts or pseudocode showing the aim of the process before being implemented in code. To ease usability, common algorithms are often packaged inside functions, which may hide some or all of the steps taken (unless you look at the function's source code, see Section 11.4).

Geoalgorithms, such as those we encountered in Chapter 10, are algorithms that take geographic data in and, generally, return geographic results (alternative terms for the same thing include *GIS algorithms* and *geometric algorithms*). That may sound simple, but it is a deep subject with an entire academic field, *Computational Geometry*, dedicated to their study (de Berg et al., 2008) and numerous books on the subject. O'Rourke (1998), for example, introduces the subject with a range of progressively harder geometric algorithms using reproducible and freely available C code.

An example of a geometric algorithm is one that finds the centroid of a polygon. There are many approaches to centroid calculation, some of which work only on specific types of spatial data. For the purposes of this section, we choose an approach that is easy to visualize: breaking the polygon into many triangles and finding the centroid of each of these, an approach discussed by Kaiser and Morin (1993) alongside other centroid algorithms (and mentioned briefly in O'Rourke, 1998). It helps to further break down this approach into discrete tasks before writing any code (subsequently referred to as step 1 to step 4, these could also be presented as a schematic diagram or pseudocode):

1. Divide the polygon into contiguous triangles
2. Find the centroid of each triangle
3. Find the area of each triangle
4. Find the area-weighted mean of triangle centroids

These steps may sound straightforward, but converting words into working code requires some work and plenty of trial-and-error, even when the inputs are constrained: The algorithm will only work for *convex polygons*, which contain no internal angles greater than 180°, no star shapes allowed (packages **decido** and **sfdct** can triangulate non-convex polygons using external libraries, as shown in the algorithm vignette hosted at geocompx.org).

The simplest data structure representing a polygon is a matrix of x and y coordinates in which each row represents a vertex tracing the polygon's border in order where the first and last rows are identical (Wise, 2001). In this case, we will create a polygon with five vertices in base R, building on an example

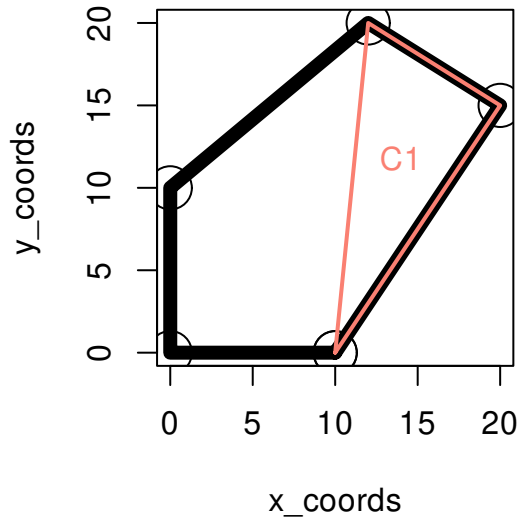

FIGURE 11.2 Polygon centroid calculation problem.

from *GIS Algorithms* (Xiao, 2016, see github.com/gisalgs for Python code), as illustrated in Figure 11.2:

```
# generate a simple matrix representation of a polygon:
x_coords = c(10, 20, 12, 0, 0, 10)
y_coords = c(0, 15, 20, 10, 0, 0)
poly_mat = cbind(x_coords, y_coords)
```

Now that we have an example dataset, we are ready to undertake step 1 outlined above. The code below shows how this can be done by creating a single triangle (`T1`), that demonstrates the method; it also demonstrates step 2 by calculating its centroid based on the formula $1/3(a+b+c)$ where a to c are coordinates representing the triangle's vertices:

```
# create a point representing the origin:
Origin = poly_mat[1, ]
# create 'triangle matrix':
T1 = rbind(Origin, poly_mat[2:3, ], Origin)
C1 = (T1[1,] + T1[2,] + T1[3,]) / 3
```

Step 3 is to find the area of each triangle, so a *weighted mean* accounting for the disproportionate impact of large triangles is accounted for. The formula to calculate the area of a triangle is as follows (Kaiser and Morin, 1993):

$$\frac{A_x(B_y - C_y) + B_x(C_y - A_y) + C_x(A_y - B_y)}{2}$$

Where A to C are the triangle's three points and x and y refer to the x and y dimensions. A translation of this formula into R code that works with the data in the matrix representation of a triangle `T1` is as follows (the function `abs()` ensures a positive result):

```
# calculate the area of the triangle represented by matrix T1:
abs(T1[1, 1] * (T1[2, 2] - T1[3, 2]) +
    T1[2, 1] * (T1[3, 2] - T1[1, 2]) +
    T1[3, 1] * (T1[1, 2] - T1[2, 2])) / 2
#> [1] 85
```

This code chunk outputs the correct result.[2] The problem is that code is clunky and must by retyped if we want to run it on another triangle matrix. To make the code more generalizable, we will see how it can be converted into a function in Section 11.4.

Step 4 requires steps 2 and 3 to be undertaken not just on one triangle (as demonstrated above) but on all triangles. This requires *iteration* to create all triangles representing the polygon, as illustrated in Figure 11.3. `lapply()` and `vapply()` are used to iterate over each triangle here because they provide a concise solution in base R:[3]

```
i = 2:(nrow(poly_mat) - 2)
T_all = lapply(i, function(x) {
  rbind(Origin, poly_mat[x:(x + 1), ], Origin)
})

C_list = lapply(T_all,  function(x) (x[1, ] + x[2, ] + x[3, ]) / 3)
C = do.call(rbind, C_list)

A = vapply(T_all, function(x) {
  abs(x[1, 1] * (x[2, 2] - x[3, 2]) +
        x[2, 1] * (x[3, 2] - x[1, 2]) +
        x[3, 1] * (x[1, 2] - x[2, 2]) ) / 2
  }, FUN.VALUE = double(1))
```

We are now in a position to complete step 4 to calculate the total area with `sum(A)` and the centroid coordinates of the polygon with `weighted.mean(C[, 1], A)`

[2] The result can be verified with the following formula (which assumes a horizontal base): area is half of the base width times height, $A = B * H/2$. In this case $10 * 10/2 = 50$.

[3] See `?lapply` for documentation and Chapter 14 for more on iteration.

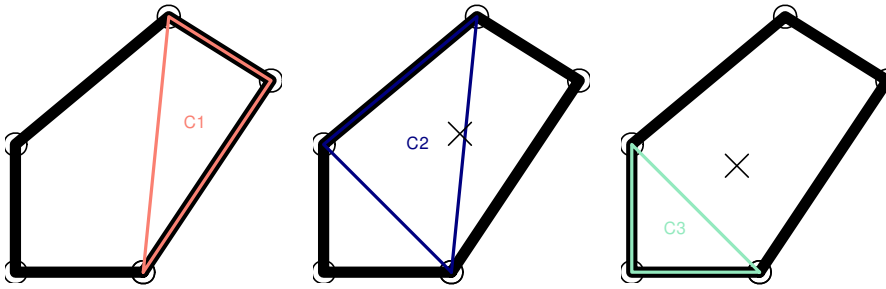

FIGURE 11.3 Iterative centroid algorithm with triangles. The X represents the area-weighted centroid in iterations 2 and 3.

and `weighted.mean(C[, 2], A)` (exercise for alert readers: verify these commands work). To demonstrate the link between algorithms and scripts, the contents of this section have been condensed into `11-centroid-alg.R`. We saw at the end of Section 11.2 how this script can calculate the centroid of a square. The great thing about *scripting* the algorithm is that it works on the new `poly_mat` object (see exercises below to verify these results with reference to `st_centroid()`):

```
source("code/11-centroid-alg.R")
#> [1] "The area is: 245"
#> [1] "The coordinates of the centroid are: 8.83, 9.22"
```

The example above shows that low-level geographic operations *can* be developed from first principles with base R. It also shows that if a tried-and-tested solution already exists, it may not be worth reinventing the wheel: if we aimed only to find the centroid of a polygon, it would have been quicker to represent `poly_mat` as an **sf** object and use the preexisting `sf::st_centroid()` function instead. However, the great benefit of writing algorithms from first principles is that you will understand every step of the process, something that cannot be guaranteed when using other peoples' code. A further consideration is performance: R may be slow compared with low-level languages such as C++ for number crunching (see Section 1.4) and optimization is difficult. If the aim is to develop new methods, computational efficiency should not be prioritized. This is captured in the saying "premature optimization is the root of all evil (or at least most of it) in programming" (Knuth, 1974).

Algorithm development is hard. This should be apparent from the amount of work that has gone into developing a centroid algorithm in base R that is just one, rather inefficient, approach to the problem with limited real-world applications (convex polygons are uncommon in practice). The experience should lead to an appreciation of low-level geographic libraries such as GEOS and CGAL (Computational Geometry Algorithms Library) which not only run fast but work on a wide range of input geometry types. A great

advantage of the open source nature of such libraries is that their source code is readily available for study, comprehension and (for those with the skills and confidence) modification.[4]

11.4 Functions

Like algorithms, functions take an input and return an output. Functions, however, refer to the implementation in a particular programming language, rather than the 'recipe' itself. In R, functions are objects in their own right, that can be created and joined together in a modular fashion. We can, for example, create a function that undertakes step 2 of our centroid generation algorithm as follows:

```
t_centroid = function(x) {
  (x[1, ] + x[2, ] + x[3, ]) / 3
}
```

The above example demonstrates two key components of functions: (1) the function *body*, the code inside the curly brackets that define what the function does with the inputs; and (2) the *arguments*, the list of arguments the function works with — `x` in this case (the third key component, the environment, is beyond the scope of this section). By default, functions return the last object that has been calculated (the coordinates of the centroid in the case of `t_centroid()`).[5]

The function now works on any inputs you pass it, as illustrated in the below command which calculates the area of the first triangle from the example polygon in the previous section (see Figure 11.3).

```
t_centroid(T1)
#> x_coords y_coords
#>     14.0     11.7
```

We can also create a function to calculate a triangle's area, which we will name `t_area()`:

[4]The CGAL function `CGAL::centroid()` is in fact composed of seven sub-functions as described at https://doc.cgal.org/latest/Kernel_23/group___centroid___grp.html allowing it to work on a wide range of input data types, whereas the solution we created works only on a very specific input data type. The source code underlying GEOS function `Centroid::getCentroid()` can be found at https://github.com/libgeos/geos/search?q=getCentroid.

[5]You can also explicitly set the output of a function by adding `return(output)` into the body of the function, where `output` is the result to be returned.

```
t_area = function(x) {
  abs(
    x[1, 1] * (x[2, 2] - x[3, 2]) +
    x[2, 1] * (x[3, 2] - x[1, 2]) +
    x[3, 1] * (x[1, 2] - x[2, 2])
  ) / 2
}
```

Note that after the function's creation, a triangle's area can be calculated in a single line of code, avoiding duplication of verbose code: functions are a mechanism for *generalizing* code. The newly created function `t_area()` takes any object `x`, assumed to have the same dimensions as the 'triangle matrix' data structure we've been using, and returns its area, as illustrated on `T1` as follows:

```
t_area(T1)
#> [1] 85
```

We can test the generalizability of the function by using it to find the area of a new triangle matrix, which has a height of 1 and a base of 3:

```
t_new = cbind(x = c(0, 3, 3, 0),
              y = c(0, 0, 1, 0))
t_area(t_new)
#>   x
#> 1.5
```

A useful feature of functions is that they are modular. Provided that you know what the output will be, one function can be used as the building block of another. Thus, the functions `t_centroid()` and `t_area()` can be used as subcomponents of a larger function to do the work of the script `11-centroid-alg.R`: calculate the area of any convex polygon. The code chunk below creates the function `poly_centroid()` to mimic the behavior of `sf::st_centroid()` for convex polygons.[6]

```
poly_centroid = function(poly_mat) {
  Origin = poly_mat[1, ] # create a point representing the origin
  i = 2:(nrow(poly_mat) - 2)
  T_all = lapply(i, function(x) {rbind(Origin, poly_mat[x:(x + 1), ], Origin)})
  C_list = lapply(T_all, t_centroid)
```

[6] Note that the functions we created are called iteratively in `lapply()` and `vapply()` function calls.

```
  C = do.call(rbind, C_list)
  A = vapply(T_all, t_area, FUN.VALUE = double(1))
  c(weighted.mean(C[, 1], A), weighted.mean(C[, 2], A))
}


poly_centroid(poly_mat)
#> [1] 8.83 9.22
```

Functions, such as `poly_centroid()`, can further be extended to provide different types of output. To return the result as an object of class `sfg`, for example, a 'wrapper' function can be used to modify the output of `poly_centroid()` before returning the result:

```
poly_centroid_sfg = function(x) {
  centroid_coords = poly_centroid(x)
  sf::st_point(centroid_coords)
}
```

We can verify that the output is the same as the output from `sf::st_centroid()` as follows:

```
poly_sfc = sf::st_polygon(list(poly_mat))
identical(poly_centroid_sfg(poly_mat), sf::st_centroid(poly_sfc))
#> [1] TRUE
```

11.5 Programming

In this chapter we have moved quickly, from scripts to functions via the tricky topic of algorithms. Not only have we discussed them in the abstract, but we have also created working examples of each to solve a specific problem:

- The script `11-centroid-alg.R` was introduced and demonstrated on a 'polygon matrix'
- The individual steps that allowed this script to work were described as an algorithm, a computational recipe
- To generalize the algorithm, we converted it into modular functions which were eventually combined to create the function `poly_centroid()` in the previous section

Each of these may seem straightforward. However, skillful programming is complex and involves *combining* each element — scripts, algorithms and functions — into a *system*, with efficiency and style. The outcome should be robust and user-friendly tools that other people can use. If you are new to programming, as we expect most people reading this book will be, being able to follow and reproduce the results in the preceding sections is a major achievement. Programming takes many hours of dedicated study and practice before you become proficient.

The challenge facing developers aiming to implement new algorithms in an efficient way is put in perspective by considering the amount of work that has gone into creating a simple function that is not intended for use in production: in its current state, `poly_centroid()` fails on most (non-convex) polygons! This raises the question: how to generalize the function? Two options are (1) to find ways to triangulate non-convex polygons (a topic covered in the online Algorithms Extended article hosted at geocompx.github.io/geocompkg/articles/) and (2) to explore other centroid algorithms that do not rely on triangular meshes.

A wider question is: is it worth programming a solution at all when high performance algorithms have already been implemented and packaged in functions such as `st_centroid()`? The reductionist answer in this specific case is 'no'. In the wider context, and considering the benefits of learning to program, the answer is 'it depends'. With programming, it's easy to waste hours trying to implement a method, only to find that someone has already done the hard work. You can understand this chapter as a stepping stone towards geometric algorithm programming wizardry. However, it can also be seen as a lesson in when to try to program a generalized solution, and when to use existing higher-level solutions. There will surely be occasions when writing new functions is the best way forward, but there will also be times when using functions that already exist is the best way forward.

"Do not reinvent the wheel" applies as much, if not more, to programming than to other walks of life. A bit of research and thinking at the outset of a project can help decide where programming time is best spent. Three principles can also help maximize use of your effort when writing code, whether it's a simple script or package that is composed of hundreds of functions:

1. DRY (don't repeat yourself): minimize repetition of code and aim to use fewer lines of code to solve a particular problem. This principle is explained with reference to the use of functions to reduce code repetition in the Functions chapter of R for Data Science (Grolemund and Wickham, 2016).
2. KISS (keep it simple stupid): this principle suggests that simple solutions should be tried first and preferred over complex solutions, using dependencies where needed, and aiming to keep scripts

concise. This principle is the computing analogy of the quote "things should be made as simple as possible, but no simpler".
3. Modularity: your code will be easier to maintain if it's divided into well-defined pieces. A function should do only one thing, but do this really well. If you function is becoming too long, think about splitting it into multiple small functions, each of which could be reused for other purposes, supporting DRY and KISS principles.

We cannot guarantee that this chapter will instantly enable you to create perfectly formed functions for your work. We are, however, confident that its contents will help you decide when is an appropriate time to try (when no other existing functions solve the problem, when the programming task is within your capabilities and when the benefits of the solution are likely to outweigh the time costs of developing it). By using the principles above, in combination with the practical experience of working through the examples above, you will build your scripting, package-writing and programming skills. First steps towards programming can be slow (the exercises below should not be rushed), but the long-term rewards can be large.

11.6 Exercises

E1. Read the script `11-centroid-alg.R` in the `code` folder of the book's GitHub repository.

- Which of the best practices covered in Section 11.2 does it follow?
- Create a version of the script on your computer in an IDE such as RStudio (preferably by typing the script line-by-line, in your own coding style and with your own comments, rather than copy-pasting — this will help you learn how to type scripts). Using the example of a square polygon (e.g., created with `poly_mat = cbind(x = c(0, 9, 9, 0, 0), y = c(0, 0, 9, 9, 0))`) execute the script line-by-line.
- What changes could be made to the script to make it more reproducible?
- How could the documentation be improved?

E2. In the geometric algorithms section, we calculated that the area of the polygon `poly_mat` was 245 units squared and that its centroid as at the coordinates (8.8, 9.2).

- Reproduce the results on your own computer with reference to the script `11-centroid-alg.R`, an implementation of this algorithm (bonus: type the commands - try to avoid copy-pasting).

- Are the results correct? Verify them by converting `poly_mat` into an `sfc` object (named `poly_sfc`) with `st_polygon()` (hint: this function takes objects of class `list()`) and then using `st_area()` and `st_centroid()`.

E3. It was stated that the algorithm we created only works for *convex hulls*. Define convex hulls (see the geometry operations chapter) and test the algorithm on a polygon that is *not* a convex hull.

- Bonus 1: Think about why the method only works for convex hulls and note changes that would need to be made to the algorithm to make it work for other types of polygon.
- Bonus 2: Building on the contents of `11-centroid-alg.R`, write an algorithm only using base R functions that can find the total length of linestrings represented in matrix form.

E4. In the functions section, we created different versions of the `poly_centroid()` function that generated outputs of class `sfg` (`poly_centroid_sfg()`) and type-stable `matrix` outputs (`poly_centroid_type_stable()`). Further extend the function by creating a version (e.g., called `poly_centroid_sf()`) that is type stable (only accepts inputs of class `sf`) *and* returns `sf` objects (hint: you may need to convert the object `x` into a matrix with the command `sf::st_coordinates(x)`).

- Verify if it works by running `poly_centroid_sf(sf::st_sf(sf::st_sfc(poly_sfc)))`
- What error message do you get when you try to run `poly_centroid_sf(poly_mat)`?

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

12

Statistical learning

Prerequisites

This chapter assumes proficiency with geographic data analysis, for example gained by studying the contents and working through the exercises in Chapters 2 to 7. A familiarity with Generalized Linear Models (GLM) and machine learning is highly recommended (for example Zuur et al., 2009, and James et al. (2013)).

The chapter uses the following packages:[1]

```
library(sf)
library(terra)
library(dplyr)
library(future)              # parallel processing
library(lgr)                 # logging framework for R
library(mlr3)                # unified interface to machine learning algorithms
library(mlr3learners)        # most important machine learning algorithms
library(mlr3extralearners)   # access to even more learning algorithms
library(mlr3proba)           # here needed for mlr3extralearners::list_learners()
library(mlr3spatiotempcv)    # spatiotemporal resampling strategies
library(mlr3tuning)          # hyperparameter tuning
library(mlr3viz)             # plotting functions for mlr3 objects
library(progressr)           # report progress updates
library(pROC)                # compute roc values
```

Required data will be attached in due course.

[1] Packages **GGally**, **lgr**, **kernlab**, **mlr3measures**, **paradox**, **pROC**, **progressr** and **spDataLarge** must also be installed, although these do not need to be attached.

12.1 Introduction

Statistical learning is concerned with the use of statistical and computational models for identifying patterns in data and predicting from these patterns. Due to its origins, statistical learning is one of R's great strengths (see Section 1.4).[2] Statistical learning combines methods from statistics and machine learning and can be categorized into supervised and unsupervised techniques. Both are increasingly used in disciplines ranging from physics, biology and ecology to geography and economics (James et al., 2013).

This chapter focuses on supervised techniques in which there is a training dataset, as opposed to unsupervised techniques such as clustering. Response variables can be binary (such as landslide occurrence), categorical (land use), integer (species richness count) or numeric (soil acidity measured in pH). Supervised techniques model the relationship between such responses — which are known for a sample of observations — and one or more predictors.

The primary aim of much machine learning research is to make good predictions. Machine learning thrives in the age of 'big data' because its methods make few assumptions about input variables and can handle huge datasets. Machine learning is conducive to tasks such as the prediction of future customer behavior, recommendation services (music, movies, what to buy next), face recognition, autonomous driving, text classification and predictive maintenance (infrastructure, industry).

This chapter is based on a case study: modeling the occurrence of landslides. This application links to the applied nature of geocomputation, defined in Chapter 1, and illustrates how machine learning borrows from the field of statistics when the sole aim is prediction. Therefore, this chapter first introduces modeling and cross-validation concepts with the help of a GLM (Zuur et al., 2009). Building on this, the chapter implements a more typical machine learning algorithm, namely a Support Vector Machine (SVM). The models' **predictive performance** will be assessed using spatial cross-validation (CV), which accounts for the fact that geographic data is special.

CV determines a model's ability to generalize to new data, by splitting a dataset (repeatedly) into training and test sets. It uses the training data to fit the model and checks its performance when predicting against the test data. CV helps to detect overfitting, since models that predict the training data too closely (noise) will tend to perform poorly on the test data.

[2] Applying statistical techniques to geographic data has been an active topic of research for many decades in the fields of geostatistics, spatial statistics and point pattern analysis (Diggle and Ribeiro, 2007; Gelfand et al., 2010; Baddeley et al., 2015).

TABLE 12.1 Structure of the lsl dataset.

	x	y	lslpts	slope	cplan	cprof	elev	log10_carea
1	713888	9558537	FALSE	34	0.023	0.003	2400	2.8
2	712788	9558917	FALSE	39	-0.039	-0.017	2100	4.1
350	713826	9559078	TRUE	35	0.020	-0.003	2400	3.2

Randomly splitting spatial data can lead to training points that are neighbors in space with test points. Due to spatial autocorrelation, test and training datasets would not be independent in this scenario, with the consequence that CV fails to detect a possible overfitting. Spatial CV alleviates this problem and is the **central** theme in this chapter.

Hence, and to emphasize it again, this chapter is focusing on the **predictive performance** of models. It **does not** teach how to do predictive mapping. This will be the topic of Chapter 15.

12.2 Case study: Landslide susceptibility

This case study is based on a dataset of landslide locations in Southern Ecuador, illustrated in Figure 12.1 and described in detail in Muenchow et al. (2012). A subset of the dataset used in that paper is provided in the **spDataLarge** package, which can be loaded as follows:

```
data("lsl", "study_mask", package = "spDataLarge")
ta = terra::rast(system.file("raster/ta.tif", package = "spDataLarge"))
```

The above code loads three objects: a `data.frame` named `lsl`, an `sf` object named `study_mask` and a `SpatRaster` (see Section 2.3.4) named `ta` containing terrain attribute rasters. `lsl` contains a factor column `lslpts` where `TRUE` corresponds to an observed landslide 'initiation point', with the coordinates stored in columns `x` and `y`.[3] There are 175 landslide and 175 non-landslide points, as shown by `summary(lsl$lslpts)`. The 175 non-landslide points were sampled randomly from the study area, with the restriction that they must fall outside a small buffer around the landslide polygons.

The first three rows of `lsl`, rounded to two significant digits, can be found in Table 12.1.

[3]The landslide initiation point is located in the scarp of a landslide polygon. See Muenchow et al. (2012) for further details.

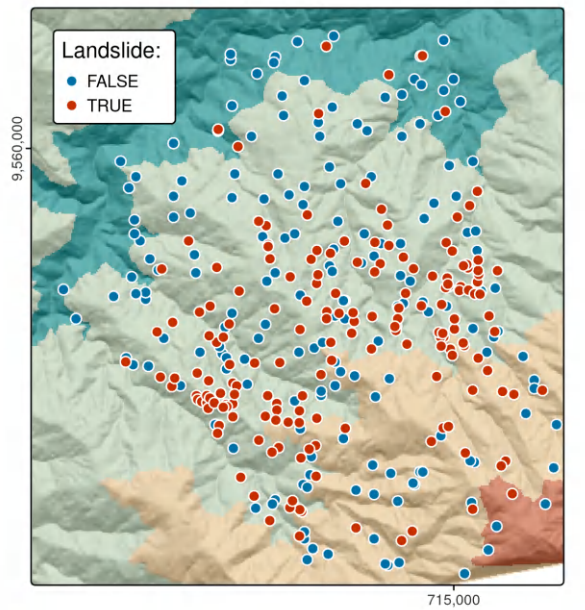

FIGURE 12.1 Landslide initiation points (red) and points unaffected by landsliding (blue) in Southern Ecuador.

To model landslide susceptibility, we need some predictors. Since terrain attributes are frequently associated with landsliding (Muenchow et al., 2012), we have already extracted following terrain attributes from `ta` to `lsl`:

- `slope`: slope angle (°)
- `cplan`: plan curvature (rad m^{-1}) expressing the convergence or divergence of a slope and thus water flow
- `cprof`: profile curvature (rad m^{-1}) as a measure of flow acceleration, also known as downslope change in slope angle
- `elev`: elevation (m a.s.l.) as the representation of different altitudinal zones of vegetation and precipitation in the study area
- `log10_carea`: the decadic logarithm of the catchment area (log10 m^2) representing the amount of water flowing toward a location

It might be a worthwhile exercise to compute the terrain attributes with the help of R-GIS bridges (see Chapter 10) and extract them to the landslide points (see Exercise section at the end of this chapter).

12.3 Conventional modeling approach in R

Before introducing the **mlr3** package, an umbrella package providing a unified interface to dozens of learning algorithms (Section 12.5), it is worth taking a look at the conventional modeling interface in R. This introduction to supervised statistical learning provides the basis for doing spatial CV, and contributes to a better grasp on the **mlr3** approach presented subsequently.

Supervised learning involves predicting a response variable as a function of predictors (Section 12.4). In R, modeling functions are usually specified using formulas (see `?formula` for more details on R formulas). The following command specifies and runs a generalized linear model:

```
fit = glm(lslpts ~ slope + cplan + cprof + elev + log10_carea,
          family = binomial(),
          data = lsl)
```

It is worth understanding each of the three input arguments:

- A formula, which specifies landslide occurrence (`lslpts`) as a function of the predictors
- A family, which specifies the type of model, in this case `binomial` because the response is binary (see `?family`)
- The data frame which contains the response and the predictors (as columns)

The results of this model can be printed as follows (`summary(fit)` provides a more detailed account of the results):

```
class(fit)
#> [1] "glm" "lm"
fit
#>
#> Call:  glm(formula = lslpts ~ slope + cplan + cprof + elev + log10_carea,
#>     family = binomial(), data = lsl)
#>
#> Coefficients:
#> (Intercept)        slope        cplan        cprof         elev  log10_carea
#>    2.51e+00     7.90e-02    -2.89e+01    -1.76e+01     1.79e-04    -2.27e+00
#>
#> Degrees of Freedom: 349 Total (i.e. Null);  344 Residual
#> Null Deviance:       485
#> Residual Deviance: 373   AIC: 385
```

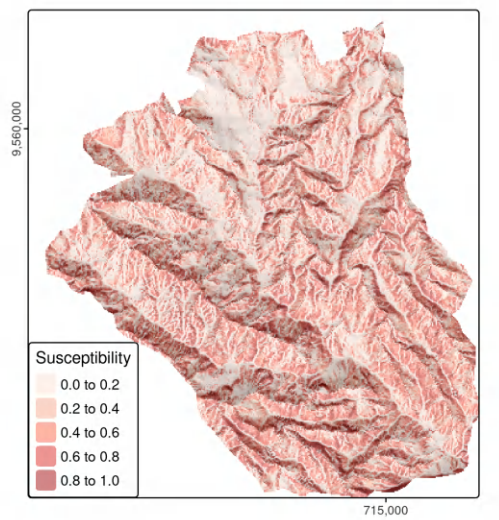

FIGURE 12.2 Spatial distribution mapping of landslide susceptibility using a GLM.

The model object `fit`, of class `glm`, contains the coefficients defining the fitted relationship between response and predictors. It can also be used for prediction. This is done with the generic `predict()` method, which in this case calls the function `predict.glm()`. Setting `type` to `response` returns the predicted probabilities (of landslide occurrence) for each observation in `lsl`, as illustrated below (see `?predict.glm`).

```
pred_glm = predict(object = fit, type = "response")
head(pred_glm)
#>      1      2      3      4      5      6
#> 0.1901 0.1172 0.0952 0.2503 0.3382 0.1575
```

Spatial distribution maps can be made by applying the coefficients to the predictor rasters. This can be done manually or with `terra::predict()`. In addition to a model object (`fit`), the latter function also expects a `SpatRaster` with the predictors (raster layers) named as in the model's input data frame (Figure 12.2).

```
# making the prediction
pred = terra::predict(ta, model = fit, type = "response")
```

Here, when making predictions, we neglect spatial autocorrelation since we assume that on average the predictive accuracy remains the same with or without spatial autocorrelation structures. However, it is possible to include

spatial autocorrelation structures into models as well as into predictions. Though, this is beyond the scope of this book, we give the interested reader some pointers where to look it up:

1. The predictions of regression kriging combines the predictions of a regression with the kriging of the regression's residuals (Goovaerts, 1997; Hengl, 2007; Bivand et al., 2013).
2. One can also add a spatial correlation (dependency) structure to a generalized least squares model (`nlme::gls()`, Zuur et al., 2009, 2017).

3. One can also use mixed-effect modeling approaches. Basically, a random effect imposes a dependency structure on the response variable which in turn allows for observations of one class to be more similar to each other than to those of another class (Zuur et al., 2009). Classes can be, for example, bee hives, owl nests, vegetation transects or an altitudinal stratification. This mixed modeling approach assumes normal and independent distributed random intercepts. This can even be extended by using a random intercept that is normal and spatially dependent. For this, however, you will have to resort most likely to Bayesian modeling approaches since frequentist software tools are rather limited in this respect especially for more complex models (Blangiardo and Cameletti, 2015; Zuur et al., 2017).

Spatial distribution mapping is one very important outcome of a model (Figure 12.2). Even more important is how good the underlying model is at making them since a prediction map is useless if the model's predictive performance is bad. One of the most popular measures to assess the predictive performance of a binomial model is the Area Under the Receiver Operator Characteristic Curve (AUROC). This is a value between 0.5 and 1.0, with 0.5 indicating a model that is no better than random and 1.0 indicating perfect prediction of the two classes. Thus, the higher the AUROC, the better the model's predictive power. The following code chunk computes the AUROC value of the model with `roc()`, which takes the response and the predicted values as inputs. `auc()` returns the area under the curve.

```
pROC::auc(pROC::roc(lsl$lslpts, fitted(fit)))
#> Area under the curve: 0.8216
```

An AUROC value of 0.82 represents a good fit. However, this is an overoptimistic estimation since we have computed it on the complete dataset. To derive a biased-reduced assessment, we have to use cross-validation and in the case of spatial data should make use of spatial CV.

12.4 Introduction to (spatial) cross-validation

Cross-validation belongs to the family of resampling methods (James et al., 2013). The basic idea is to split (repeatedly) a dataset into training and test sets whereby the training data is used to fit a model which then is applied to the test set. Comparing the predicted values with the known response values from the test set (using a performance measure such as the AUROC in the binomial case) gives a bias-reduced assessment of the model's capability to generalize the learned relationship to independent data. For example, a 100-repeated 5-fold cross-validation means to randomly split the data into five partitions (folds) with each fold being used once as a test set (see upper row of Figure 12.3). This guarantees that each observation is used once in one of the test sets, and requires the fitting of five models. Subsequently, this procedure is repeated 100 times. Of course, the data splitting will differ in each repetition. Overall, this sums up to 500 models, whereas the mean performance measure (AUROC) of all models is the model's overall predictive power.

However, geographic data is special. As we will see in Chapter 13, the 'first law' of geography states that points close to each other are, generally, more similar than points further away (Miller, 2004). This means these points are not statistically independent because training and test points in conventional CV are often too close to each other (see first row of Figure 12.3). 'Training' observations near the 'test' observations can provide a kind of 'sneak preview': information that should be unavailable to the training dataset. To alleviate this problem, 'spatial partitioning' is used to split the observations into spatially disjointed subsets (using the observations' coordinates in a k-means clustering; Brenning (2012b); second row of Figure 12.3). This partitioning strategy is the **only** difference between spatial and conventional CV. As a result, spatial CV leads to a bias-reduced assessment of a model's predictive performance, and hence helps to avoid overfitting.

12.5 Spatial CV with mlr3

There are dozens of packages for statistical learning, as described for example in the CRAN machine learning task view. Getting acquainted with each of these packages, including how to undertake cross-validation and hyperparameter tuning, can be a time-consuming process. Comparing model results from different packages can be even more laborious. The **mlr3** package and ecosystem was developed to address these issues. It acts as a 'meta-package', providing a unified interface to popular supervised and unsupervised statistical learning techniques including classification, regression, survival analysis

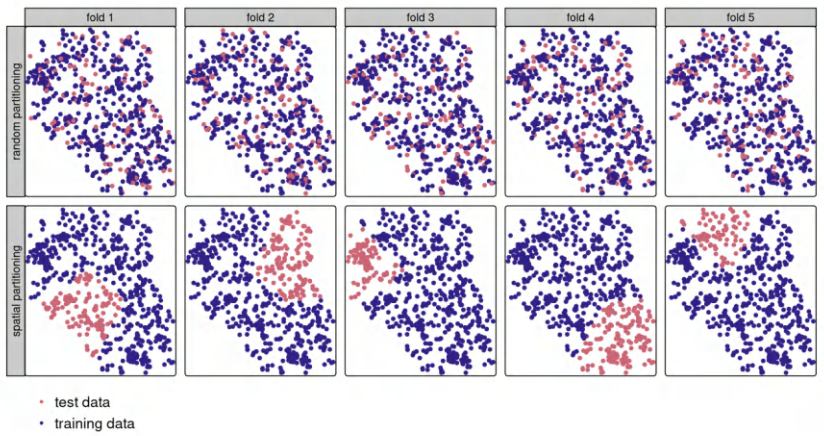

FIGURE 12.3 Spatial visualization of selected test and training observations for cross-validation of one repetition. Random (upper row) and spatial partitioning (lower row).

and clustering (Lang et al., 2019; Bischl et al., 2024). The standardized **mlr3** interface is based on eight 'building blocks'. As illustrated in Figure 12.4, these have a clear order.

The **mlr3** modeling process consists of three main stages. First, a **task** specifies the data (including response and predictor variables) and the model type (such as regression or classification). Second, a **learner** defines the specific learning algorithm that is applied to the created task. Third, the **resampling** approach assesses the predictive performance of the model, i.e., its ability to generalize to new data (see also Section 12.4).

12.5.1 Generalized linear model

To use a GLM in **mlr3**, we must create a **task** containing the landslide data. Since the response is binary (two-category variable) and has a spatial dimension, we create a classification task with `as_task_classif_st()` of the **mlr3spatiotempcv** package (Schratz et al., 2021, for non-spatial tasks, use `mlr3::as_task_classif()` or `mlr3::as_task_regr()` for regression tasks, see `?Task` for other task types).[4] The first essential argument of these `as_task_` functions is `x`. `x` expects that the input data includes the response and predictor variables. The `target` argument indicates the name of a response variable (in our case

[4]The **mlr3** ecosystem makes use of **data.table** and **R6** classes. And though you might use **mlr3** without knowing the specifics of **data.table** or **R6**, it might be rather helpful. To learn more about **data.table**, please refer to https://rdatatable.gitlab.io/data.table/. To learn more about **R6**, we recommend Chapter 14 of the *Advanced R* book (Wickham, 2019).

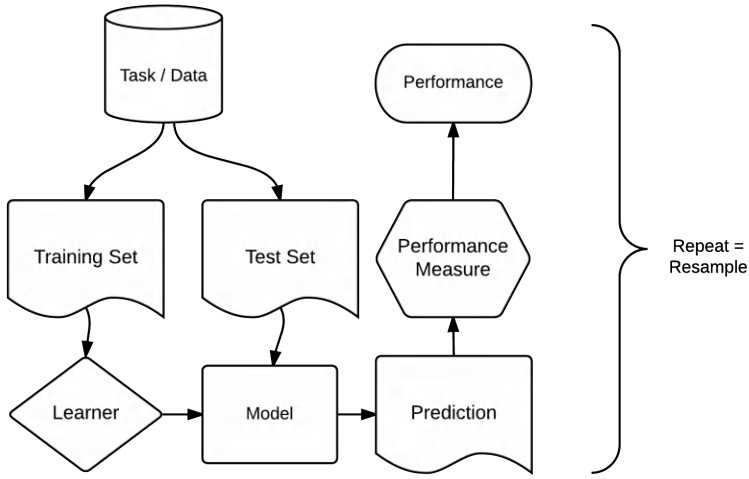

FIGURE 12.4 Basic building blocks of the mlr3 package (Bischl et al., 2024). Permission to reuse this figure was kindly granted.

this is `lslpts`) and `positive` determines which of the two factor levels of the response variable indicate the landslide initiation point (in our case this is `TRUE`). All other variables of the `lsl` dataset will serve as predictors. For spatial CV, we need to provide a few extra arguments. The `coordinate_names` argument expects the names of the coordinate columns (see Section 12.4 and Figure 12.3). Additionally, we should indicate the used CRS (`crs`) and decide if we want to use the coordinates as predictors in the modeling (`coords_as_features`).

```
# 1. create task
task = mlr3spatiotempcv::as_task_classif_st(
  mlr3::as_data_backend(lsl),
  target = "lslpts",
  id = "ecuador_lsl",
  positive = "TRUE",
  coordinate_names = c("x", "y"),
  crs = "EPSG:32717",
  coords_as_features = FALSE
  )
```

Note that `mlr3spatiotempcv::as_task_classif_st()` also accepts an `sf`-object as input for the `backend` parameter. In this case, you might only want to additionally specify the `coords_as_features` argument. We did not convert `lsl` into

TABLE 12.2 Sample of available learners for binomial tasks in the mlr3 package.

Class	Name	Short name	Package
classif.adaboostm1	ada Boosting M1	adaboostm1	RWeka
classif.binomial	Binomial Regression	binomial	stats
classif.featureless	Featureless classifier	featureless	mlr
classif.fnn	Fast k-Nearest Neighbour	fnn	FNN
classif.gausspr	Gaussian Processes	gausspr	kernlab
classif.IBk	k-Nearest Neighbours	ibk	RWeka

an `sf`-object because `as_task_classif_st()` would just turn it back into a non-spatial `data.table` object in the background.

For a short data exploration, the `autoplot()` function of the **mlr3viz** package might come in handy since it plots the response against all predictors and all predictors against all predictors (not shown).

```
# plot response against each predictor
mlr3viz::autoplot(task, type = "duo")
# plot all variables against each other
mlr3viz::autoplot(task, type = "pairs")
```

Having created a task, we need to choose a **learner** that determines the statistical learning method to use. All classification **learners** start with `classif.` and all regression learners with `regr.` (see `?Learner` for details). `mlr3extralearners::list_mlr3learners()` lists all available learners and from which package **mlr3** imports them (Table 12.2). To find out about learners that are able to model a binary response variable, we can run:

```
mlr3extralearners::list_mlr3learners(
  filter = list(class = "classif", properties = "twoclass"),
  select = c("id", "mlr3_package", "required_packages")) |>
  head()
```

This yields all learners able to model two-class problems (landslide yes or no). We opt for the binomial classification method used in Section 12.3 and implemented as `classif.log_reg` in **mlr3learners**. Additionally, we need to specify the `predict.type` which determines the type of the prediction with `prob` resulting in the predicted probability for landslide occurrence between 0 and 1 (this corresponds to `type = response` in `predict.glm()`).

```
# 2. specify learner
learner = mlr3::lrn("classif.log_reg", predict_type = "prob")
```

To access the help page of the learner and find out from which package it was taken, we can run:

```
learner$help()
```

The setup steps for modeling with **mlr3** may seem tedious. But remember, this single interface provides access to the 130+ learners shown by `mlr3extralearners::list_mlr3learners()`; it would be far more tedious to learn the interface for each learner! Further advantages are simple parallelization of resampling techniques and the ability to tune machine learning hyperparameters (see Section 12.5.2). Most importantly, (spatial) resampling in **mlr3spatiotempcv** (Schratz et al., 2021) is straightforward, requiring only two more steps: specifying a resampling method and running it. We will use a 100-repeated 5-fold spatial CV: five partitions will be chosen based on the provided coordinates in our `task` and the partitioning will be repeated 100 times:[5]

```
# 3. specify resampling
resampling = mlr3::rsmp("repeated_spcv_coords", folds = 5, repeats = 100)
```

To execute the spatial resampling, we run `resample()` using the previously specified task, learner, and resampling strategy. This takes some time (around 15 seconds on a modern laptop) because it computes 500 resampling partitions and 500 models. Again, we choose the AUROC as performance measure. To retrieve it, we use the `score()` method of the resampling result output object (`score_spcv_glm`). This returns a `data.table` object with 500 rows – one for each model.

```
# reduce verbosity
lgr::get_logger("mlr3")$set_threshold("warn")
# run spatial cross-validation and save it to resample result glm (rr_glm)
rr_spcv_glm = mlr3::resample(task = task,
                             learner = learner,
                             resampling = resampling)
```

[5]Note that package **sperrorest** initially implemented spatial cross-validation in R (Brenning, 2012b). In the meantime, its functionality was integrated into the **mlr3** ecosystem which is the reason why we are using **mlr3** (Schratz et al., 2019). The **tidymodels** framework is another umbrella package for streamlined modeling in R; however, it only recently integrated support for spatial cross-validation via **spatialsample**, which so far only supports one spatial resampling method.

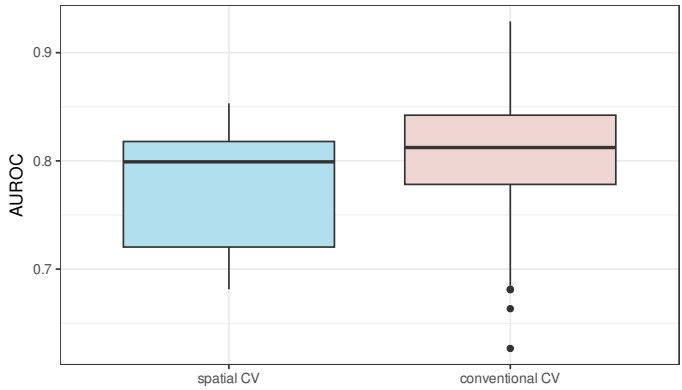

FIGURE 12.5 Boxplot showing the difference in GLM AUROC values on spatial and conventional 100-repeated 5-fold cross-validation.

```
# compute the AUROC as a data.table
score_spcv_glm = rr_spcv_glm$score(measure = mlr3::msr("classif.auc"))
# keep only the columns you need
score_spcv_glm = dplyr::select(score_spcv_glm, task_id, learner_id,
                               resampling_id, classif.auc)
```

The output of the preceding code chunk is a bias-reduced assessment of the model's predictive performance. We have saved it as `extdata/12-bmr_score.rds` in the book's GitHub repository. If required, you can read it in as follows:

```
score = readRDS("extdata/12-bmr_score.rds")
score_spcv_glm = dplyr::filter(score, learner_id == "classif.log_reg",
                               resampling_id == "repeated_spcv_coords")
```

To compute the mean AUROC over all 500 models, we run:

```
mean(score_spcv_glm$classif.auc) |>
  round(2)
#> [1] 0.77
```

To put these results in perspective, let us compare them with AUROC values from a 100-repeated 5-fold non-spatial cross-validation (Figure 12.5; the code for the non-spatial cross-validation is not shown here but will be explored in the Exercise section). As expected (see Section 12.4), the spatially cross-validated result yields lower AUROC values on average than the conventional cross-validation approach, underlining the over-optimistic predictive performance of the latter due to its spatial autocorrelation.

12.5.2 Spatial tuning of machine-learning hyperparameters

Section 12.4 introduced machine learning as part of statistical learning. To recap, we adhere to the following definition of machine learning by Jason Brownlee:

> Machine learning, more specifically the field of predictive modeling, is primarily concerned with minimizing the error of a model or making the most accurate predictions possible, at the expense of explainability. In applied machine learning we will borrow, reuse and steal algorithms from many different fields, including statistics and use them towards these ends.

In Section 12.5.1 a GLM was used to predict landslide susceptibility. This section introduces support vector machines (SVMs) for the same purpose. Random forest models might be more popular than SVMs; however, the positive effect of tuning hyperparameters on model performance is much more pronounced in the case of SVMs (Probst et al., 2018). Since (spatial) hyperparameter tuning is the major aim of this section, we will use an SVM. For those wishing to apply a random forest model, we recommend to read this chapter, and then proceed to Chapter 15 in which we will apply the currently covered concepts and techniques to make spatial distribution maps based on a random forest model.

SVMs search for the best possible 'hyperplanes' to separate classes (in a classification case) and estimate 'kernels' with specific hyperparameters to create non-linear boundaries between classes (James et al., 2013). Machine learning algorithms often feature hyperparameters and parameters. Parameters can be estimated from the data, while hyperparameters are set before the learning begins (see also the machine mastery blog and the hyperparameter optimization chapter of the mlr3 book). The optimal hyperparameter configuration is usually found within a specific search space and determined with the help of cross-validation methods. This is called hyperparameter tuning and the main topic of this section.

Some SVM implementations such as that provided by **kernlab** allow hyperparameters to be tuned automatically, usually based on random sampling (see upper row of Figure 12.3). This works for non-spatial data but is of less use for spatial data where 'spatial tuning' should be undertaken.

Before defining spatial tuning, we will set up the **mlr3** building blocks, introduced in Section 12.5.1, for the SVM. The classification task remains the same,

hence, we can simply reuse the `task` object created in Section 12.5.1. Learners implementing SVM can be found using the `list_mlr3learners()` command of the **mlr3extralearners**.

```
mlr3_learners = mlr3extralearners::list_mlr3learners()
#> This will take a few seconds.
mlr3_learners |>
  dplyr::filter(class == "classif" & grepl("svm", id)) |>
  dplyr::select(id, class, mlr3_package, required_packages)
#>               id   class      mlr3_package              required_packages
#>           <char>  <char>            <char>                         <list>
#> 1:  classif.ksvm classif mlr3extralearners mlr3,mlr3extralearners,kernlab
#> 2: classif.lssvm classif mlr3extralearners mlr3,mlr3extralearners,kernlab
#> 3:   classif.svm classif      mlr3learners        mlr3,mlr3learners,e1071
```

Of the options, we will use `ksvm()` from the **kernlab** package (Karatzoglou et al., 2004). To allow for non-linear relationships, we use the popular radial basis function (or Gaussian) kernel (`"rbfdot"`) which is also the default of `ksvm()`. Setting the `type` argument to `"C-svc"` makes sure that `ksvm()` is solving a classification task. To make sure that the tuning does not stop because of one failing model, we additionally define a fallback learner (for more information please refer to https://mlr3book.mlr-org.com/chapters/chapter10/advanced_technical_aspects_of_mlr3.html#sec-fallback).

```
lrn_ksvm = mlr3::lrn("classif.ksvm", predict_type = "prob", kernel = "rbfdot",
                     type = "C-svc")
lrn_ksvm$encapsulate(method = "try",
                     fallback = lrn("classif.featureless",
                                    predict_type = "prob"))
```

The next stage is to specify a resampling strategy. Again we will use a 100-repeated 5-fold spatial CV.

```
# performance estimation level
perf_level = mlr3::rsmp("repeated_spcv_coords", folds = 5, repeats = 100)
```

Note that this is the exact same code as used for the resampling for the GLM in Section 12.5.1; we have simply repeated it here as a reminder.

So far, the process has been identical to that described in Section 12.5.1. The next step is new, however: to tune the hyperparameters. Using the same data for the performance assessment and the tuning would potentially lead to overoptimistic results (Cawley and Talbot, 2010). This can be avoided using nested spatial CV.

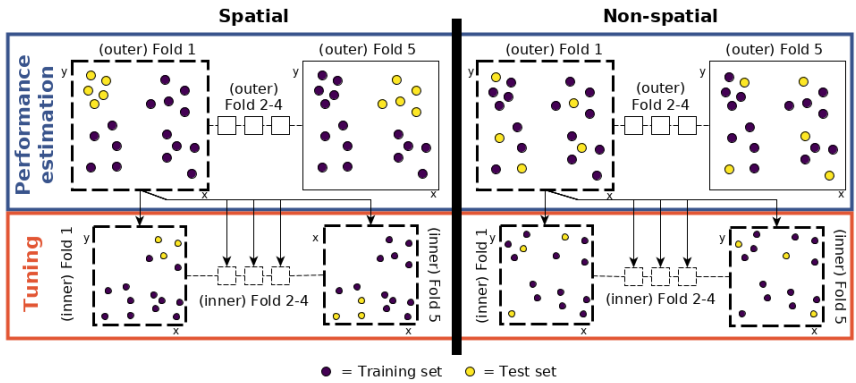

FIGURE 12.6 Schematic of hyperparameter tuning and performance estimation levels in CV. (Figure was taken from Schratz et al. (2019). Permission to reuse it was kindly granted.)

This means that we split each fold again into five spatially disjoint subfolds which are used to determine the optimal hyperparameters (`tune_level` object in the code chunk below; see Figure 12.6 for a visual representation). The random selection of values C and Sigma is additionally restricted to a predefined tuning space (`search_space` object). The range of the tuning space was chosen with values recommended in the literature (Schratz et al., 2019). To find the optimal hyperparameter combination, we fit 50 models (`terminator` object in the code chunk below) in each of these subfolds with randomly selected values for the hyperparameters C and Sigma.

```
# five spatially disjoint partitions
tune_level = mlr3::rsmp("spcv_coords", folds = 5)
# define the outer limits of the randomly selected hyperparameters
search_space = paradox::ps(
  C = paradox::p_dbl(lower = -12, upper = 15, trafo = function(x) 2^x),
  sigma = paradox::p_dbl(lower = -15, upper = 6, trafo = function(x) 2^x)
)
# use 50 randomly selected hyperparameters
terminator = mlr3tuning::trm("evals", n_evals = 50)
tuner = mlr3tuning::tnr("random_search")
```

The next stage is to modify the learner `lrn_ksvm` in accordance with all the characteristics defining the hyperparameter tuning with `auto_tuner()`.

```
at_ksvm = mlr3tuning::auto_tuner(
  learner = lrn_ksvm,
  resampling = tune_level,
  measure = mlr3::msr("classif.auc"),
  search_space = search_space,
  terminator = terminator,
  tuner = tuner
)
```

The tuning is now set up to fit 250 models to determine optimal hyperparameters for one fold. Repeating this for each fold, we end up with 1,250 (250 * 5) models for each repetition. Repeated 100 times means fitting a total of 125,000 models to identify optimal hyperparameters (Figure 12.3). These are used in the performance estimation, which requires the fitting of another 500 models (5 folds * 100 repetitions; see Figure 12.3). To make the performance estimation processing chain even clearer, let us write down the commands we have given to the computer:

1. Performance level (upper left part of Figure 12.6): split the dataset into five spatially disjoint (outer) subfolds.
2. Tuning level (lower left part of Figure 12.6): use the first fold of the performance level and split it again spatially into five (inner) subfolds for the hyperparameter tuning. Use the 50 randomly selected hyperparameters in each of these inner subfolds, i.e., fit 250 models.
3. Performance estimation: use the best hyperparameter combination from the previous step (tuning level) and apply it to the first outer fold in the performance level to estimate the performance (AUROC).
4. Repeat steps 2 and 3 for the remaining four outer folds.
5. Repeat steps 2 to 4, 100 times.

The process of hyperparameter tuning and performance estimation is computationally intensive. To decrease model runtime, **mlr3** offers the possibility to use parallelization with the help of the **future** package. Since we are about to run a nested cross-validation, we can decide if we would like to parallelize the inner or the outer loop (see lower left part of Figure 12.6). Since the former will run 125,000 models, whereas the latter only runs 500, it is quite obvious that we should parallelize the inner loop. To set up the parallelization of the inner loop, we run:

```
library(future)
# execute the outer loop sequentially and parallelize the inner loop
```

```
future::plan(list("sequential", "multisession"),
             workers = floor(availableCores() / 2))
```

Additionally, we instructed **future** to only use half instead of all available cores (default), a setting that allows possible other users to work on the same high performance computing cluster in case one is used.

Now we are set up for computing the nested spatial CV. Specifying the `resample()` parameters follows the exact same procedure as presented when using a GLM, the only difference being the `store_models` and `encapsulate` arguments. Setting the former to `TRUE` would allow the extraction of the hyperparameter tuning results which is important if we plan follow-up analyses on the tuning. The latter ensures that the processing continues even if one of the models throws an error. This avoids the process stopping just because of one failed model, which is desirable on large model runs. Once the processing is completed, one can have a look at the failed models. After the processing, it is good practice to explicitly stop the parallelization with `future:::ClusterRegistry("stop")`. Finally, we save the output object (`result`) to disk in case we would like to use it in another R session. Before running the subsequent code, be aware that it is time-consuming since it will run the spatial cross-validation with 125,500 models. It can easily run for half a day on a modern laptop. Note that runtime depends on many aspects: CPU speed, the selected algorithm, the selected number of cores and the dataset.

```
progressr::with_progress(expr = {
  rr_spcv_svm = mlr3::resample(task = task,
                               learner = at_ksvm,
                               # outer resampling (performance level)
                               resampling = perf_level,
                               store_models = FALSE,
                               encapsulate = "evaluate")
})
# stop parallelization
future:::ClusterRegistry("stop")
# compute the AUROC values
score_spcv_svm = rr_spcv_svm$score(measure = mlr3::msr("classif.auc"))
# keep only the columns you need
score_spcv_svm = dplyr::select(score_spcv_svm, task_id, learner_id,
                               resampling_id, classif.auc)
```

In case you do not want to run the code locally, we have saved score_svm in the book's GitHub repository. They can be loaded as follows:

```
score = readRDS("extdata/12-bmr_score.rds")
score_spcv_svm = dplyr::filter(score, learner_id == "classif.ksvm.tuned",
                               resampling_id == "repeated_spcv_coords")
```

Let's have a look at the final AUROC: the model's ability to discriminate the two classes.

```
# final mean AUROC
round(mean(score_spcv_svm$classif.auc), 2)
#> [1] 0.74
```

It appears that the GLM (aggregated AUROC was 0.77) is slightly better than the SVM in this specific case. To guarantee an absolute fair comparison, one should also make sure that the two models use the exact same partitions – something we have not shown here but have silently used in the background (see `code/12_cv.R` in the book's GitHub repository for more information). To do so, **mlr3** offers the functions `benchmark_grid()` and `benchmark()` (see also https://mlr3book.mlr-org.com/chapters/chapter3/evaluation_and_benchmarking.html#sec-benchmarking, Bischl et al., 2024). We will explore these functions in more detail in the Exercises. Please note also that using more than 50 iterations in the random search of the SVM would probably yield hyperparameters that result in models with a better AUROC (Schratz et al., 2019). On the other hand, increasing the number of random search iterations would also increase the total number of models and thus runtime.

So far spatial CV has been used to assess the ability of learning algorithms to generalize to unseen data. For predictive mapping purposes, one would tune the hyperparameters on the complete dataset. This will be covered in Chapter 15.

12.6 Conclusions

Resampling methods are an important part of a data scientist's toolbox (James et al., 2013). This chapter used cross-validation to assess predictive performance of various models. As described in Section 12.4, observations with spatial coordinates may not be statistically independent due to spatial autocorrelation, violating a fundamental assumption of cross-validation. Spatial CV addresses this issue by reducing bias introduced by spatial autocorrelation.

The **mlr3** package facilitates (spatial) resampling techniques in combination with the most popular statistical learning techniques including linear regression, semi-parametric models such as generalized additive models and machine

learning techniques such as random forests, SVMs, and boosted regression trees (Bischl et al., 2016; Schratz et al., 2019). Machine learning algorithms often require hyperparameter inputs, the optimal 'tuning' of which can require thousands of model runs which require large computational resources, consuming much time, RAM and/or cores. **mlr3** tackles this issue by enabling parallelization.

Machine learning overall, and its use to understand spatial data, is a large field and this chapter has provided the basics, but there is more to learn. We recommend the following resources in this direction:

- The **mlr3 book** (Bischl et al. (2024); https://mlr3book.mlr-org.com/) and especially the chapter on the handling of spatiotemporal data
- An academic paper on hyperparameter tuning (Schratz et al., 2019)
- An academic paper on how to use **mlr3spatiotempcv** (Schratz et al., 2021)
- In case of spatiotemporal data, one should account for spatial and temporal autocorrelation when doing CV (Meyer et al., 2018)

12.7 Exercises

E1. Compute the following terrain attributes from the `elev` dataset loaded with `terra::rast(system.file("raster/ta.tif", package = "spDataLarge"))$elev` with the help of R-GIS bridges (see the bridges to GIS software chapter):

- Slope
- Plan curvature
- Profile curvature
- Catchment area

E2. Extract the values from the corresponding output rasters to the `lsl` data frame (`data("lsl", package = "spDataLarge")`) by adding new variables called `slope`, `cplan`, `cprof`, `elev` and `log_carea`.

E3. Use the derived terrain attribute rasters in combination with a GLM to make a spatial prediction map similar to that shown in Figure 12.2. Running `data("study_mask", package = "spDataLarge")` attaches a mask of the study area.

E4. Compute a 100-repeated 5-fold non-spatial cross-validation and spatial CV based on the GLM learner and compare the AUROC values from both resampling strategies with the help of boxplots.

Hint: You need to specify a non-spatial resampling strategy.

Another hint: You might want to solve Excercises 4 to 6 in one go with the help of `mlr3::benchmark()` and `mlr3::benchmark_grid()` (for more information, please refer to https://mlr3book.mlr-org.com/chapters/chapter10/advanced_techn

ical_aspects_of_mlr3.html#sec-fallback). When doing so, keep in mind that the computation can take very long, probably several days. This, of course, depends on your system. Computation time will be shorter the more RAM and cores you have at your disposal.

E5. Model landslide susceptibility using a quadratic discriminant analysis (QDA). Assess the predictive performance of the QDA. What is the a difference between the spatially cross-validated mean AUROC value of the QDA and the GLM?

E6. Run the SVM without tuning the hyperparameters. Use the `rbfdot` kernel with $\sigma = 1$ and $C = 1$. Leaving the hyperparameters unspecified in **kernlab**'s `ksvm()` would otherwise initialize an automatic non-spatial hyperparameter tuning.

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

Part III

Applications

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

13

Transportation

Prerequisites

- This chapter uses the following packages:[1]

```
library(sf)
library(dplyr)
library(spDataLarge)
library(stplanr)      # for processing geographic transport data
library(tmap)         # map-making (see Chapter 9)
library(ggplot2)      # data visualization package
library(sfnetworks)   # spatial network classes and functions
```

13.1 Introduction

In few other sectors is geographic space more tangible than transportation. The effort of moving (overcoming distance) is central to the 'first law' of geography, defined by Waldo Tobler in 1970 as follows (Tobler, 1970):

> Everything is related to everything else, but near things are more related than distant things.

This 'law' is the basis for spatial autocorrelation and other key geographic concepts. It applies to phenomena as diverse as friendship networks and ecological diversity and can be explained by the costs of transport — in terms of time,

[1] The **nabor** package must also be installed, although it does not need to be attached.

energy and money — which constitute the 'friction of distance'. From this perspective, transport technologies are disruptive, changing spatial relationships between geographic entities including mobile humans and goods: "the purpose of transportation is to overcome space" (Rodrigue et al., 2013).

Transport is an inherently spatial activity, involving moving from an origin point 'A' to a destination point 'B', through infinite localities in between. It is therefore unsurprising that transport researchers have long turned to geographic and computational methods to understand movement patterns, and how interventions can improve their performance (Lovelace, 2021).

This chapter introduces the geographic analysis of transport systems at different geographic levels:

- **Areal units**: transport patterns can be understood with reference to zonal aggregates, such as the main mode of travel (by car, bike or foot, for example), and average distance of trips made by people living in a particular zone, covered in Section 13.3
- **Desire lines**: straight lines that represent 'origin-destination' data that records how many people travel (or could travel) between places (points or zones) in geographic space, the topic of Section 13.4
- **Nodes**: these are points in the transport system that can represent common origins and destinations and public transport stations such as bus stops and rail stations, the topic of Section 13.5
- **Routes**: these are lines representing a path along the route network along the desire lines and between nodes. Routes (which can be represented as single linestrings or multiple short *segments*) and the *routing engines* that generate them, are covered in Section 13.6
- **Route networks**: these represent the system of roads, paths and other linear features in an area and are covered in Section 13.7. They can be represented as geographic features (typically short segments of road that add up to create a full network) or structured as an interconnected graph, with the level of traffic on different segments referred to as 'flow' by transport modelers (Hollander, 2016)

Another key level is **agents**, mobile entities like you and me and vehicles that enable us to move such as bikes and buses. These can be represented computationally in software such as MATSim and A/B Street, which represent the dynamics of transport systems using an agent-based modeling (ABM) framework, usually at high levels of spatial and temporal resolution (Horni et al., 2016). ABM is a powerful approach to transport research with great potential for integration with R's spatial classes (Thiele, 2014; Lovelace and Dumont, 2016), but is outside the scope of this chapter. Beyond geographic levels and agents, the basic unit of analysis in many transport models is the **trip**, a single purpose journey from an origin 'A' to a destination 'B' (Hollander, 2016). Trips join-up the different levels of transport systems and can be represented simplistically as geographic *desire lines* connecting *zone* centroids (*nodes*) or

as routes that follow the transport *route network*. In this context, *agents* are usually point entities that move within the transport network.

Transport systems are dynamic (Xie and Levinson, 2011). While the focus of this chapter is the *geographic* analysis of a transport systems, it provides insights into how the approach can be used to simulate scenarios of change, in Section 13.8. The purpose of geographic transport modeling can be interpreted as simplifying the complexity of these spatiotemporal systems in ways that capture their essence. Selecting appropriate levels of geographic analysis can help simplify this complexity without losing its most important features and variables, enabling better decision-making and more effective interventions (Hollander, 2016).

Typically, models are designed to tackle a particular problem, such as how to improve safety or the environmental performance of transport systems. For this reason, this chapter is based around a policy scenario, introduced in the next section, that asks: how to increase cycling in the city of Bristol? Chapter 14 demonstrates a related application of geocomputation: prioritizing the location of new bike shops. There is a link between the chapters: new and effectively-located cycling infrastructure can get people cycling, boosting demand for bike shops and local economic activity. This highlights an important feature of transport systems: they are closely linked to broader phenomena and land-use patterns.

13.2 A case study of Bristol

The case study used for this chapter is located in Bristol, a city in the west of England, around 30 km east of the Welsh capital Cardiff. An overview of the region's transport network is illustrated in Figure 13.1, which shows a diversity of transport infrastructure, for cycling, public transport, and private motor vehicles.

Bristol is the tenth largest city council in England, with a population of half a million people, although its travel catchment area is larger (see Section 13.3). It has a vibrant economy with aerospace, media, financial service and tourism companies, alongside two major universities. Bristol shows a high average income per person but also contains areas of severe deprivation (Bristol City Council, 2015).

In terms of transport, Bristol is well served by rail and road links, and it has a relatively high level of active travel. According to the Active People Survey, 19% of its citizens cycle and 88% walk at least once per month (the national average is 15% and 81%, respectively). 8% of the population said they cycled to work in the 2011 census, compared with only 3% nationwide.

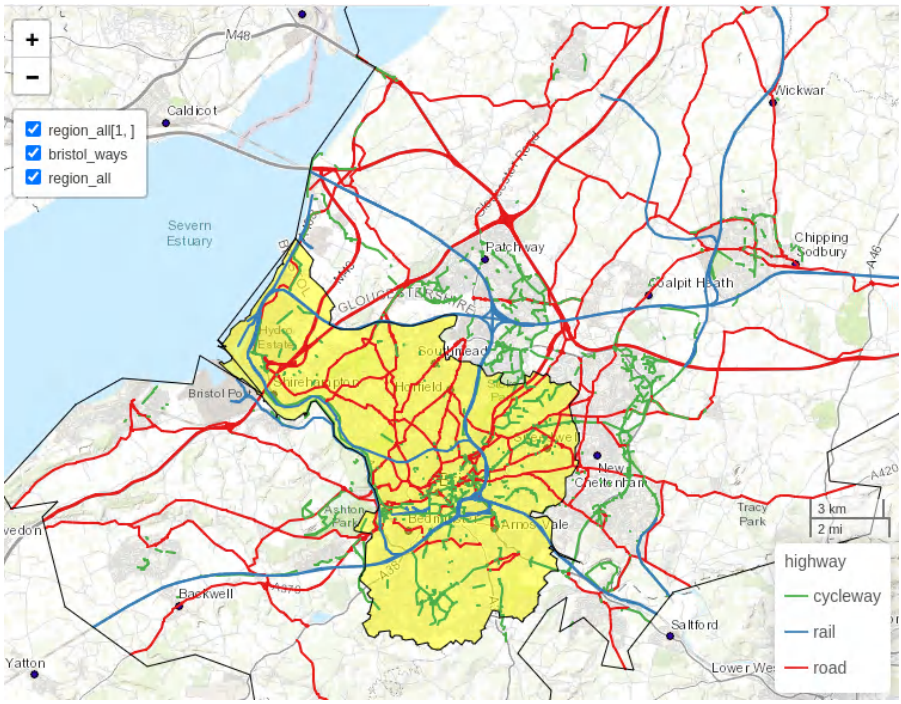

FIGURE 13.1 Bristol's transport network represented by colored lines for active (green), public (railways, blue) and private motor (red) modes of travel. Black border lines represent the inner city boundary (highlighted in yellow) and the larger Travel To Work Area (TTWA).

Like many cities, Bristol has major congestion, air quality and physical inactivity problems. Cycling can tackle all of these issues efficiently: it has a greater potential to replace car trips than walking, with typical speeds of 15-20 km/h vs. 4-6 km/h for walking. For this reason, Bristol's Transport Strategy has ambitious plans for cycling.

To highlight the importance of policy considerations in transportation research, this chapter is guided by the need to provide evidence for people (transport planners, politicians and other stakeholders) tasked with getting people out of cars and onto more sustainable modes — walking and cycling in particular. The broader aim is to demonstrate how geocomputation can support evidence-based transport planning. In this chapter you will learn how to:

- Describe the geographical patterns of transport behavior in cities
- Identify key public transport nodes supporting multi-modal trips
- Analyze travel 'desire lines' to find where many people drive short distances

- Identify cycle route locations that will encourage less car driving and more cycling

To get the wheels rolling on the practical aspects of this chapter, the next section begins by loading zonal data on travel patterns. These zone-level datasets are small but often vital for gaining a basic understanding of a settlement's overall transport system.

13.3 Transport zones

Although transport systems are primarily based on linear features and nodes — including pathways and stations — it often makes sense to start with areal data, to break continuous space into tangible units (Hollander, 2016). In addition to the boundary defining the study area (Bristol in this case), two zone types are of particular interest to transport researchers: origin and destination zones. Often, the same geographic units are used for origins and destinations. However, different zoning systems, such as 'Workplace Zones', may be appropriate to represent the increased density of trip destinations in areas with many 'trip attractors' such as schools and shops (Office for National Statistics, 2014).

The simplest way to define a study area is often the first matching boundary returned by OpenStreetMap. This can be done with a command such as `osmdata::getbb("Bristol", format_out = "sf_polygon", limit = 1)`. This returns an `sf` object (or a list of `sf` objects if `limit = 1` is not specified) representing the bounds of the largest matching city region, either a rectangular polygon of the bounding box or a detailed polygonal boundary.[2] For Bristol, a detailed polygon is returned, as represented by the `bristol_region` object in the **spDataLarge** package. See the inner blue boundary in Figure 13.1: there are a couple of issues with this approach:

- The first boundary returned by OSM may not be the official boundary used by local authorities
- Even if OSM returns the official boundary, this may be inappropriate for transport research because they bear little relation to where people travel

Travel To Work Areas (TTWAs) address these issues by creating a zoning system analogous to hydrological watersheds. TTWAs were first defined as contiguous zones within which 75% of the population travels to work (Coombes et al., 1986), and this is the definition used in this chapter. Because Bristol is a major employer attracting travel from surrounding towns, its TTWA

[2] In cases where the first match does not provide the right name, the country or region should be specified, for example `Bristol Tennessee` for a Bristol located in America.

is substantially larger than the city bounds (see Figure 13.1). The polygon representing this transport-orientated boundary is stored in the object `bristol_ttwa`, provided by the **spDataLarge** package loaded at the beginning of this chapter.

The origin and destination zones used in this chapter are the same: officially defined zones of intermediate geographic resolution (their official name is Middle layer Super Output Areas or MSOAs). Each houses around 8,000 people. Such administrative zones can provide vital context to transport analysis, such as the type of people who might benefit most from particular interventions (e.g., Moreno-Monroy et al. (2017)).

The geographic resolution of these zones is important: small zones with high geographic resolution are usually preferable, but their high number in large regions can have consequences for processing. This is especially true for origin-destination (OD) analysis in which the number of possibilities increases as a non-linear function of the number of zones (Hollander, 2016).

Another issue with small zones is related to anonymity rules. To make it impossible to infer the identity of individuals in zones, detailed socio-demographic variables are often only available at a low geographic resolution. Breakdowns of travel mode by age and sex, for example, are available at the Local Authority level in the UK, but not at the much higher Output Area level, each of which contains around 100 households. For further details, see www.ons.gov.uk/methodology/geography.

The 102 zones used in this chapter are stored in `bristol_zones`, as illustrated in Figure 13.2. Note the zones get smaller in densely populated areas: each houses a similar number of people. `bristol_zones` contains no attribute data on transport, however, only the name and code of each zone:

```
names(bristol_zones)
#> [1] "geo_code" "name"     "geometry"
```

To add travel data, we will perform an *attribute join*, a common task described in Section 3.2.4. We will use travel data from the UK's 2011 census question on travel to work, data stored in `bristol_od`, which was provided by the ons.gov.uk data portal. `bristol_od` is an OD dataset on travel to work between zones from the UK's 2011 Census (see Section 13.4). The first column is the ID of the zone of origin and the second column is the zone of destination. `bristol_od` has more rows than `bristol_zones`, representing travel *between* zones rather than the zones themselves:

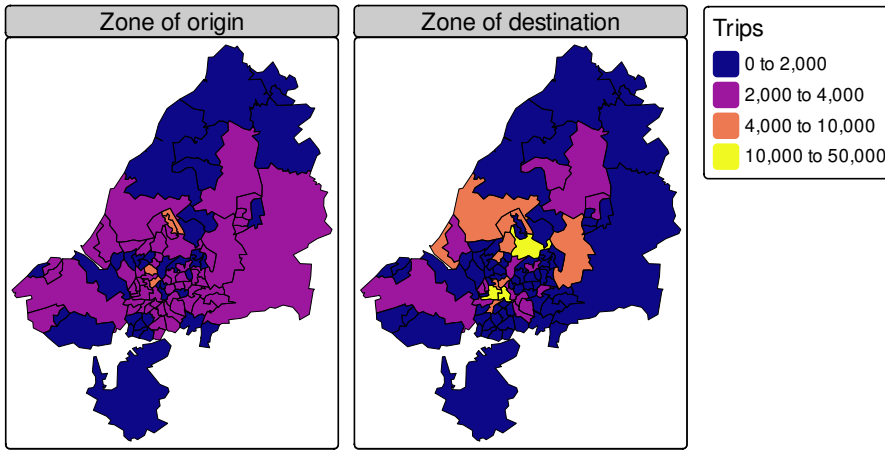

FIGURE 13.2 Number of trips (commuters) living and working in the region. The left map shows zone of origin of commute trips; the right map shows zone of destination (generated by the script '13-zones.R').

```
nrow(bristol_od)
#> [1] 2910
nrow(bristol_zones)
#> [1] 102
```

The results of the previous code chunk shows that there are more than 10 OD pairs for every zone, meaning we will need to aggregate the origin-destination data before it is joined with `bristol_zones`, as illustrated below (OD data is described in Section 13.4).

```
zones_attr = bristol_od |>
  group_by(o) |>
  summarize(across(where(is.numeric), sum)) |>
  dplyr::rename(geo_code = o)
```

The preceding chunk:

- Grouped the data by zone of origin (contained in the column `o`)
- Aggregated the variables in the `bristol_od` dataset *if* they were numeric, to find the total number of people living in each zone by mode of transport[3]
- Renamed the grouping variable `o` so it matches the ID column `geo_code` in the `bristol_zones` object

[3]The `_if` affix requires a `TRUE`/`FALSE` question to be asked of the variables, in this case 'is it numeric?' and only variables returning true are summarized.

The resulting object `zones_attr` is a data frame with rows representing zones and an ID variable. We can verify that the IDs match those in the `zones` dataset using the `%in%` operator as follows:

```
summary(zones_attr$geo_code %in% bristol_zones$geo_code)
#>    Mode    TRUE
#> logical     102
```

The results show that all 102 zones are present in the new object and that `zone_attr` is in a form that can be joined onto the zones.[4] This is done using the joining function `left_join()` (note that `inner_join()` would produce here the same result):

```
zones_joined = left_join(bristol_zones, zones_attr, by = "geo_code")
sum(zones_joined$all)
#> [1] 238805
names(zones_joined)
#> [1] "geo_code"   "name"       "all"        "bicycle"    "foot"
#> [6] "car_driver" "train"      "geometry"
```

The result is `zones_joined`, which contains new columns representing the total number of trips originating in each zone in the study area (almost 1/4 of a million) and their mode of travel (by bicycle, foot, car and train). The geographic distribution of trip origins is illustrated in the left-hand panel in Figure 13.2. This shows that most zones have between 0 and 4,000 trips originating from them in the study area. More trips are made by people living near the center of Bristol and fewer on the outskirts. Why is this? Remember that we are only dealing with trips within the study region: low trip numbers in the outskirts of the region can be explained by the fact that many people in these peripheral zones will travel to other regions outside of the study area. Trips outside the study region can be included in a regional model by a special destination ID covering any trips that go to a zone not represented in the model (Hollander, 2016). The data in `bristol_od`, however, simply ignores such trips: it is an 'intra-zonal' model.

In the same way that OD datasets can be aggregated to the zone of origin, they can also be aggregated to provide information about destination zones. People tend to gravitate towards central places. This explains why the spatial distribution represented in the right panel in Figure 13.2 is relatively uneven, with the most common destination zones concentrated in Bristol city center.

[4] It would also be important to check that IDs match in the opposite direction on real data. This could be done by changing the order of the IDs in the `summary()` command — `summary(bristol_zones$geo_code %in% zones_attr$geo_code)` — or by using `setdiff()` as follows: `setdiff(bristol_zones$geo_code, zones_attr$geo_code)`.

The result is `zones_od`, which contains a new column reporting the number of trip destinations by any mode, and it is created as follows:

```
zones_destinations = bristol_od |>
  group_by(d) |>
  summarize(across(where(is.numeric), sum)) |>
  select(geo_code = d, all_dest = all)
zones_od = inner_join(zones_joined, zones_destinations, by = "geo_code")
```

A simplified version of Figure 13.2 is created with the code below (see `13-zones.R` in the `code` folder of the book's GitHub repository to reproduce the figure and Section 9.2.7 for details on faceted maps with **tmap**):

```
qtm(zones_od, c("all", "all_dest")) +
  tm_layout(panel.labels = c("Origin", "Destination"))
```

13.4 Desire lines

Desire lines connect origins and destinations, representing where people *desire* to go, typically between zones. They represent the quickest 'bee line' or 'crow flies' route between A and B that would be taken, if it were not for obstacles such as buildings and windy roads getting in the way (we will see how to convert desire lines into routes in the next section). Typically, desire lines are represented geographically as starting and ending in the geographic (or population weighted) centroid of each zone. This is the type of desire line that we will create and use in this section, although it is worth being aware of 'jittering' techniques that enable multiple start and end points to increase the spatial coverage and accuracy of analyses building on OD data (Lovelace et al., 2022).

We have already loaded data representing desire lines in the dataset `bristol_od`. This data frame represents the number of people traveling between the zone represented in `o` and `d`, as illustrated in Table 13.1. To arrange the OD data by all trips and then filter-out only the top 5, type (please refer to Chapter 3 for a detailed description of non-spatial attribute operations):

```
od_top5 = bristol_od |>
  slice_max(all, n = 5)
```

TABLE 13.1 Sample of the top 5 origin-destination pairs in the Bristol OD data frame, representing travel desire lines between zones in the study area.

o	d	all	bicycle	foot	car_driver	train
E02003043	E02003043	1493	66	1296	64	8
E02003047	E02003043	1300	287	751	148	8
E02003031	E02003043	1221	305	600	176	7
E02003037	E02003043	1186	88	908	110	3
E02003034	E02003043	1177	281	711	100	7

The resulting table provides a snapshot of Bristolian travel patterns in terms of commuting (travel to work). It demonstrates that walking is the most popular mode of transport among the top 5 OD pairs, that zone `E02003043` is a popular destination (Bristol city center, the destination of all the top 5 OD pairs), and that the *intrazonal* trips, from one part of zone `E02003043` to another (first row of Table 13.1), constitute the most traveled OD pair in the dataset. But from a policy perspective, the raw data presented in Table 13.1 is of limited use: aside from the fact that it contains only a tiny portion of the 2,910 OD pairs, it tells us little about *where* policy measures are needed, or *what proportion* of trips are made by walking and cycling. The following command calculates the percentage of each desire line that is made by these active modes:

```
bristol_od$Active = (bristol_od$bicycle + bristol_od$foot) /
  bristol_od$all * 100
```

There are two main types of OD pairs: *interzonal* and *intrazonal*. Interzonal OD pairs represent travel between zones in which the destination is different from the origin. Intrazonal OD pairs represent travel within the same zone (see the top row of Table 13.1). The following code chunk splits `od_bristol` into these two types:

```
od_intra = filter(bristol_od, o == d)
od_inter = filter(bristol_od, o != d)
```

The next step is to convert the interzonal OD pairs into an `sf` object representing desire lines that can be plotted on a map with the **stplanr** function `od2line()`.[5]

[5] `od2line()` works by matching the IDs in the first two columns of the `bristol_od` object to the `zone_code` ID column in the geographic `zones_od` object. Note that the operation emits a warning because `od2line()` works by allocating the start and end points of each origin-destination pair to the *centroid* of its zone of origin and destination. For real-world use one would use centroid values generated from projected data or, preferably, use *population-weighted* centroids (Lovelace et al., 2017).

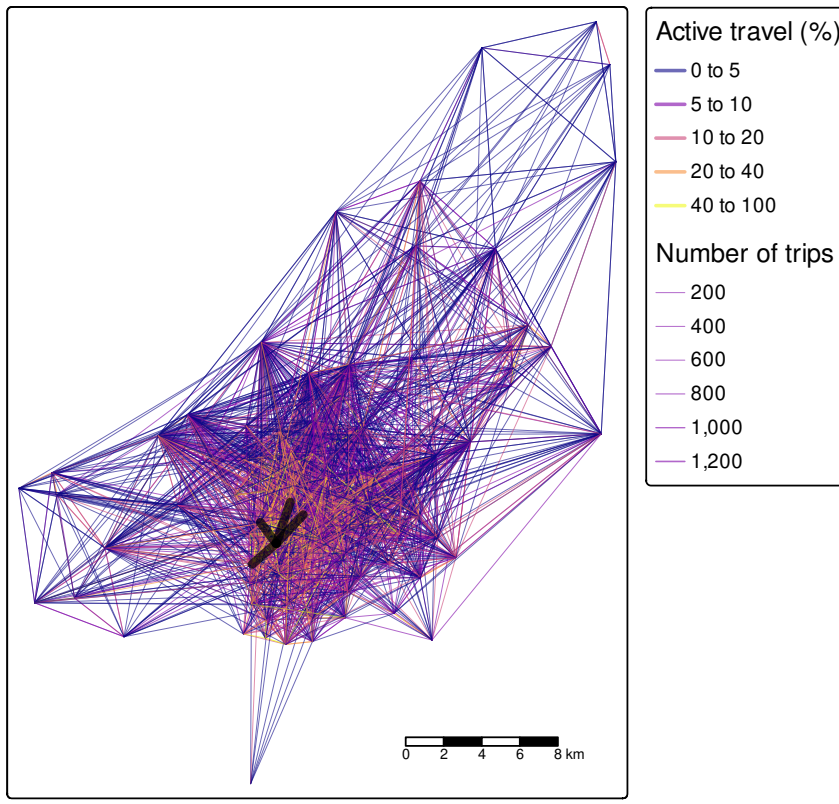

FIGURE 13.3 Desire lines representing trip patterns in Bristol, with width representing number of trips and color representing the percentage of trips made by active modes (walking and cycling). The four black lines represent the interzonal OD pairs in Table 13.1.

```
desire_lines = od2line(od_inter, zones_od)
#> Creating centroids representing desire line start and end points.
```

An illustration of the results is presented in Figure 13.3, a simplified version of which is created with the following command (see the code in `13-desire.R` to reproduce the figure exactly and Chapter 9 for details on visualization with **tmap**):

```
qtm(desire_lines, lines.lwd = "all")
```

The map shows that the city center dominates transport patterns in the region, suggesting policies should be prioritized there, although a number of

peripheral sub-centers can also be seen. Desire lines are important generalized components of transport systems. More concrete components include nodes, which have specific destinations (rather than hypothetical straight lines represented in desire lines). Nodes are covered in the next section.

13.5 Nodes

Nodes in geographic transport datasets are points among the predominantly linear features that comprise transport networks. Broadly are two main types of transport nodes:

1. Nodes not directly on the network such as zone centroids or individual origins and destinations such as houses and workplaces
2. Nodes that are a part of transport networks. Technically, a node can be located at any point on a transport network but in practice they are often special kinds of vertex such as intersections between pathways (junctions) and points for entering or exiting a transport network such as bus stops and train stations[6]

Transport networks can be represented as graphs, in which each segment is connected (via edges representing geographic lines) to one or more other edges in the network. Nodes outside the network can be added with "centroid connectors", new route segments to nearby nodes on the network (Hollander, 2016).[7] Every node in the network is then connected by one or more 'edges' that represent individual segments on the network. We will see how transport networks can be represented as graphs in Section 13.7.

Public transport stops are particularly important nodes that can be represented as either type of node: a bus stop that is part of a road, or a large rail station that is represented by its pedestrian entry point hundreds of meters from railway tracks. We will use railway stations to illustrate public transport nodes, in relation to the research question of increasing cycling in Bristol. These stations are provided by **spDataLarge** in `bristol_stations`.

A common barrier preventing people from switching away from cars for commuting to work is that the distance from home to work is too far to walk or cycle. Public transport can reduce this barrier by providing a fast and high-volume option for common routes into cities. From an active travel perspective, public transport 'legs' of longer journeys divide trips into three:

[6] The function `st_network_blend()` from the **sfnetworks** package allows new nodes to be created on the network based on proximity to arbitrary points on or off the network.

[7] The location of these connectors should be chosen carefully because they can lead to over-estimates of traffic volumes in their immediate surroundings (Jafari et al., 2015).

- The origin leg, typically from residential areas to public transport stations
- The public transport leg, which typically goes from the station nearest a trip's origin to the station nearest its destination
- The destination leg, from the station of alighting to the destination

Building on the analysis conducted in Section 13.4, public transport nodes can be used to construct three-part desire lines for trips that can be taken by bus and (the mode used in this example) rail. The first stage is to identify the desire lines with most public transport travel, which in our case is easy because our previously created dataset `desire_lines` already contains a variable describing the number of trips by train (the public transport potential could also be estimated using public transport routing services such as OpenTripPlanner). To make the approach easier to follow, we will select only the top three desire lines in terms of rails use:

```
desire_rail = top_n(desire_lines, n = 3, wt = train)
```

The challenge now is to 'break-up' each of these lines into three pieces, representing travel via public transport nodes. This can be done by converting a desire line into a multilinestring object consisting of three line geometries representing origin, public transport and destination legs of the trip. This operation can be divided into three stages: matrix creation (of origins, destinations and the 'via' points representing rail stations), identification of nearest neighbors and conversion to multilinestrings. These are undertaken by `line_via()`. This **stplanr** function takes input lines and points and returns a copy of the desire lines — see the `?line_via()` for details on how this works. The output is the same as the input line, except it has new geometry columns representing the journey via public transport nodes, as demonstrated below:

```
ncol(desire_rail)
#> [1] 9
desire_rail = line_via(desire_rail, bristol_stations)
ncol(desire_rail)
#> [1] 12
```

As illustrated in Figure 13.4, the initial `desire_rail` lines now have three additional geometry list columns representing travel from home to the origin station, from there to the destination, and finally from the destination station to the destination. In this case, the destination leg is very short (walking distance), but the origin legs may be sufficiently far to justify investment in cycling infrastructure to encourage people to cycle to the stations on the outward leg of peoples' journey to work in the residential areas surrounding the three origin stations in Figure 13.4.

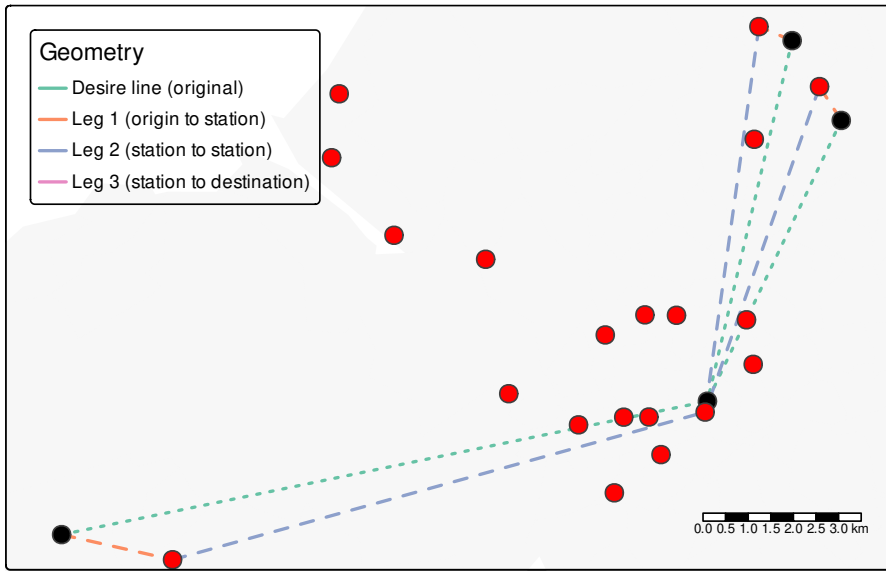

FIGURE 13.4 Station nodes (red dots) used as intermediary points that convert straight desire lines with high rail usage (thin green lines) into three legs: to the origin station (orange) via public transport (blue) and to the destination (pink, not visible because it is short).

13.6 Routes

From a geographical perspective, routes are desire lines that are no longer straight: the origin and destination points are the same as in the desire line representation of travel, but the pathway to get from A to B is more complex. The geometries of routes are typically (but not always) determined by the transport network.

While desire lines contain only two vertices (their beginning and end points), routes can contain any number of vertices, representing points between A and B joined by straight lines: the definition of a linestring geometry. Routes covering large distances or following intricate networks can have thousands of vertices; routes on grid-based or simplified road networks tend to have fewer.

Routes are generated from desire lines or, more commonly, matrices containing coordinate pairs representing desire lines. This routing process is done by a range of broadly-defined *routing engines*: software and web services that return geometries and attributes describing how to get from origins to destinations. Routing engines can be classified based on *where* they run relative to R:

- In-memory routing using R packages that enable route calculation (described in Section 13.6.2)
- Locally hosted routing engines external to R that can be called from R (Section 13.6.3)
- Remotely hosted routing engines by external entities that provide a web API that can be called from R (Section 13.6.4)

Before describing each, it is worth outlining other ways of categorizing routing engines. Routing engines can be multi-modal, meaning that they can calculate trips composed of more than one mode of transport, or not. Multi-modal routing engines can return results consisting of multiple *legs*, each one made by a different mode of transport. The optimal route from a residential area to a commercial area could involve (1) walking to the nearest bus stop, (2) catching the bus to the nearest node to the destination, and (3) walking to the destination, given a set of input parameters. The transition points between these three legs are commonly referred to as 'ingress' and 'egress', meaning getting on/off a public transport vehicle. Multi-modal routing engines such as R5 are more sophisticated and have larger input data requirements than 'uni-modal' routing engines such as the OpenStreetMap Routing Machine (OSRM), described in Section 13.6.3.

A major strength of multi-modal engines is their ability to represent 'transit' (public transport) trips by trains, buses, etc. Multi-model routing engines require input datasets representing public transport networks, typically in General Transit Feed Specification (GTFS) files, which can be processed with functions in the **tidytransit** and **gtfstools** packages (other packages and tools for working with GTFS files are available). Single mode routing engines may be sufficient for projects focused on specific (non-public) modes of transport. Another way of classifying routing engines (or settings) is by the geographic level of the outputs: routes, legs and segments.

13.6.1 Routes, legs and segments

Routing engines can generate outputs at three geographic levels of routes, legs and segments:

- **Route** level outputs contain a single feature (typically a multilinestring and associated row in the data frame representation) per origin-destination pair, meaning a single row of data per trip
- **Leg** level outputs contain a single feature and associated attributes each *mode* within each origin-destination (OD) pair, as described in Section 13.5. For trips only involving one mode (for example driving, from home to work, ignoring the short walk to the car), the leg is the same as the route: the car journey. For trips involving public transport, legs provide key information. The **r5r** function `detailed_itineraries()` returns legs which, confusingly, are sometimes referred to as 'segments'

- Segment-level outputs provide the most detailed information about routes, with records for each small section of the transport network. Typically segments are similar in length, or identical to, ways in OpenStreetMap. The **cyclestreets** function `journey()` returns data at the segment-level which can be aggregated by grouping by origin- and destination-level data returned by the `route()` function in **stplanr**

Most routing engines return route-level by default, although multi-modal engines generally provide outputs at the leg level (one feature per continuous movement by a single mode of transport). Segment-level outputs have the advantage of providing more detail. The **cyclestreets** package returns multiple 'quietness' levels per route, enabling identification of the 'weakest link' in cycle networks. Disadvantages of segment-level outputs include increased file sizes and complexities associated with the extra detail.

Route-level results can be converted into segment-level results using the function `stplanr::overline()` (Morgan and Lovelace, 2020). When working with segment- or leg-level data, route-level statistics can be returned by grouping by columns representing trip start and end points and summarizing/aggregating columns containing segment-level data.

13.6.2 In-memory routing with R

Routing engines in R enable route networks stored as R objects *in memory* to be used as the basis of route calculation. Options include **sfnetworks**, **dodgr** and **cppRouting** packages, each of which provide its own class system to represent route networks, the topic of the next section.

While fast and flexible, native R routing options are generally harder to set up than dedicated routing engines for realistic route calculation. Routing is a hard problem and many hundreds of hours have been put into open source routing engines that can be downloaded and hosted locally. On the other hand, R-based routing engines may be well suited to model experiments and the statistical analysis of the impacts of changes on the network. Changing route network characteristics (or weights associated with different route segment types), recalculating routes, and analyzing results under many scenarios in a single language have benefits for research applications.

13.6.3 Locally hosted dedicated routing engines

Locally hosted routing engines include OpenTripPlanner, Valhalla, and R5 (which are multi-modal), and the OpenStreetMap Routing Machine (OSRM) (which is 'uni-modal'). These can be accessed from R with the packages **opentripplanner**, **valhallr**, **r5r** and **osrm** (Morgan et al., 2019; Pereira et al., 2021). Locally hosted routing engines run on the user's computer but in a process separate from R. They benefit from speed of execution and control over

the weighting profile for different modes of transport. Disadvantages include the difficulty of representing complex networks locally, lack of predefined routing profiles, temporal dynamics (e.g. traffic), and the need to install specialized software.

13.6.4 Remotely hosted dedicated routing engines

Remotely hosted routing engines use a web API to send queries about origins and destinations and return results. Routing services based on open source routing engines, such as OSRM's publicly available service, work the same when called from R as locally hosted instances, simply requiring arguments specifying 'base URLs' to be updated. However, the fact that external routing services are hosted on a dedicated machine (usually funded by a commercial company with incentives to generate accurate routes) can give them advantages, including:

- Provision of routing services worldwide (or usually at least over a large region)
- Established routing services are usually updated regularly and can often respond to traffic levels
- Routing services usually run on dedicated hardware and software including systems such as load balancers to ensure consistent performance

Disadvantages of remote routing services include speed when batch jobs are not possible (they often rely on data transfer over the internet on a route-by-route basis), price (the Google routing API, for example, limits the number of free queries) and licensing issues. **googleway** and **mapbox** packages demonstrate this approach by providing access to routing services from Google and Mapbox, respectively. Free (but rate limited) routing service include OSRM and openrouteservice.org which can be accessed from R with the **osrm** and **openrouteservice** packages, the latter of which is not on CRAN. There are also more specific routing services such as that provided by CycleStreets.net, a cycle journey planner and not-for-profit transport technology company "for cyclists, by cyclists". While R users can access CycleStreets routes via the package **cyclestreets**, many routing services lack R interfaces, representing a substantial opportunity for package development: building an R package to provide an interface to a web API can be a rewarding experience.

13.6.5 Contraction hierarchies and traffic assigment

Contraction hierarchies and traffic assignment are advanced but important topics in transport modeling that are worth being aware of, especially if you want your code to scale to large networks. Calculating many routes is computationally resource intensive and can take hours, leading to the development of several algorithms to speed up routing calculations. **Contraction hierarchies** is a well-known algorithm that can lead to a substantial (1000x+ in

some cases) speed up in routing tasks, depending on network size. Contraction hierarchies are used behind the scenes in the routing engines mentioned in the previous sections.

Traffic assignment is a problem that is closely related to routing: in practice, the shortest path between two points is not always the fastest, especially if there is congestion. The process takes OD datasets, of the kind described in Section 13.4, and assigns traffic to each segment in the network, generating route networks of the kind described in Section 13.7. An established solution is Wardrop's principle of user equilibrium which shows that, to be realistic, congestion should be considered when estimating flows on the network, with reference to a mathematically defined relationship between cost and flow (Wardrop, 1952). This optimization problem can be solved by iterative algorithms which are implemented in the **cppRouting** package, which also implements contraction hierarchies for fast routing.

13.6.6 Routing: A worked example

Instead of routing *all* desire lines generated in Section 13.4, we focus on a subset that is highly policy-relevant. Running a computationally intensive operation on a subset before trying to process the whole dataset is often sensible, and this applies to routing. Routing can be time- and memory-consuming, resulting in large objects, due to the detailed geometries and extra attributes of route objects. We will therefore filter the desire lines before calculating routes in this section.

Cycling is most beneficial when it replaces car trips. Short trips (around 5 km, which can be cycled in 15 minutes at a speed of 20 km/hr) have a relatively high probability of being cycled, and the maximum distance increases when trips are made by electric bike (Lovelace et al., 2017). These considerations inform the following code chunk which filters the desire lines and returns the object `desire_lines_short` representing OD pairs between which many (100+) short (2.5 to 5 km Euclidean distance) trips are driven:

```
desire_lines$distance_km = as.numeric(st_length(desire_lines)) / 1000
desire_lines_short = desire_lines |>
  filter(car_driver >= 100, distance_km <= 5, distance_km >= 2.5)
```

In the code above, `st_length()` calculated the length of each desire line, as described in Section 4.2.3. The `filter()` function from **dplyr** filtered the `desire_lines` dataset based on the criteria outlined above, as described in Section 3.2.1. The next stage is to convert these desire lines into routes. This is done using the publicly available OSRM service with the **stplanr** functions `route()` and `route_osrm()` in the code chunk below:

```
routes_short = route(l = desire_lines_short, route_fun = route_osrm,
                     osrm.profile = "car")
```

The output is `routes_short`, an `sf` object representing routes on the transport network that are suitable for cycling (according to the OSRM routing engine at least), one for each desire line. Note: calls to external routing engines such as in the command above only work with an internet connection (and sometimes an API key stored in an environment variable, although not in this case). In addition to the columns contained in the `desire_lines` object, the new route dataset contains `distance` (referring to route distance this time) and `duration` columns (in seconds), which provide potentially useful extra information on the nature of each route. We will plot desire lines along which many short car journeys take place alongside cycling routes. Making the width of the routes proportional to the number of car journeys that could potentially be replaced provides an effective way to prioritize interventions on the road network (Lovelace et al., 2017). Figure 13.5 shows routes along which people drive short distances (see the github.com/geocompx for the source code).[8]

Visualizing the results in an interactive map shows that many short car trips take place in and around Bradley Stoke, around 10 km North of central Bristol. It is easy to find explanations for the area's high level of car-dependency: according to Wikipedia, Bradley Stoke is "Europe's largest new town built with private investment", suggesting limited public transport provision. Furthermore, the town is surrounded by large (cycling unfriendly) road structures, including the M4 and M5 motorways (Tallon, 2007).

There are many benefits of converting travel desire lines into routes. It is important to remember that we cannot be sure how many (if any) trips will follow the exact routes calculated by routing engines. However, route and street/way/segment-level results can be highly policy-relevant. Route segment results can enable the prioritization of investment where it is most needed, according to available data (Lovelace et al., 2017).

[8]Note that the red routes and black desire lines do not start at exactly the same points. This is because zone centroids rarely lie on the route network; instead the routes originate from the transport network node nearest the centroid. Note also that routes are assumed to originate in the zone centroids, a simplifying assumption which is used in transport models to reduce the computational resources needed to calculate the shortest path between all combinations of possible origins and destinations (Hollander, 2016).

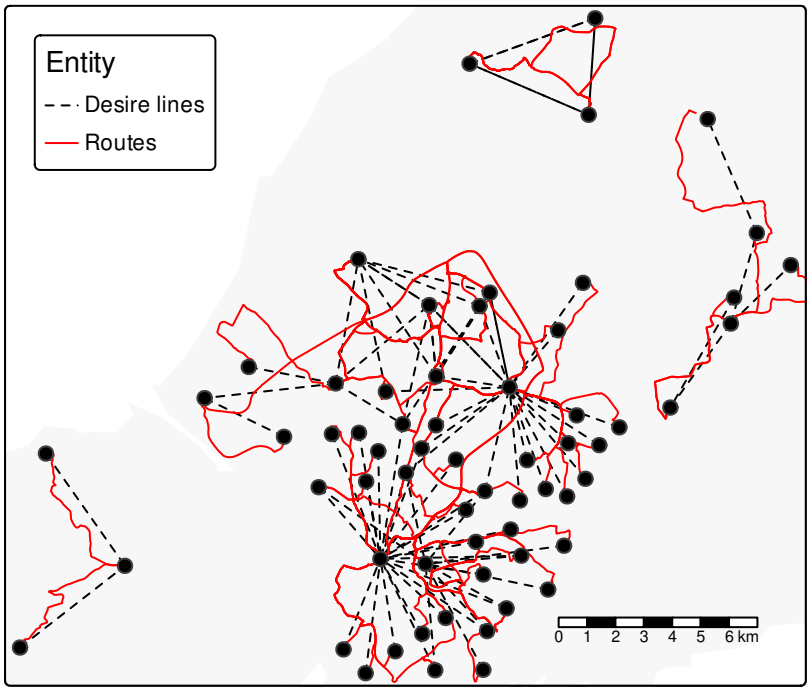

FIGURE 13.5 Routes along which many (100+) short (<5km Euclidean distance) car journeys are made (red) overlaying desire lines representing the same trips (black) and zone centroids (dots).

13.7 Route networks

While routes generally contain data on travel *behavior*, at the same geographic level as desire lines and OD pairs, route network datasets usually represent the physical transport network. Each *segment* in a route network roughly corresponds to a continuous section of street between junctions and appears only once, although the average length of segments depends on the data source (segments in the OSM-derived `bristol_ways` dataset used in this section have an average length of just over 200 m, with a standard deviation of nearly 500 m). Variability in segment lengths can be explained by the fact that in some rural locations junctions are far apart, while in dense urban areas there are crossings and other segment breaks every few meters.

Route networks can be an input into, or an output of, transport data analysis projects, or both. Any transport research that involves route calculation requires a route network dataset in the internal or external routing engines (in the latter case the route network data is not necessarily imported into R).

However, route networks are also important outputs in many transport research projects: summarizing data such as the potential number of trips made on particular segments and represented as a route network, can help prioritize investment where it is most needed.

To demonstrate how to create route networks as an output derived from route-level data, imagine a simple scenario of mode shift. Imagine that 50% of car trips between 0 to 3 km in route distance are replaced by cycling, a percentage that drops by 10 percentage points for every additional kilometer of route distance so that 20% of car trips of 6 km are replaced by cycling and no car trips that are eight km or longer are replaced by cycling. This is of course an unrealistic scenario (Lovelace et al., 2017), but is a useful starting point. In this case, we can model mode shift from cars to bikes as follows:

```
uptake = function(x) {
  case_when(
    x <= 3 ~ 0.5,
    x >= 8 ~ 0,
    TRUE ~ (8 - x) / (8 - 3) * 0.5
  )
}
routes_short_scenario = routes_short |>
  mutate(uptake = uptake(distance / 1000)) |>
  mutate(bicycle = bicycle + car_driver * uptake,
         car_driver = car_driver * (1 - uptake))
sum(routes_short_scenario$bicycle) - sum(routes_short$bicycle)
#> [1] 3252
```

Having created a scenario in which approximately 4000 trips have switched from driving to cycling, we can now model where this updated modeled cycling activity will take place. For this, we will use the function `overline()` from the **stplanr** package. The function breaks linestrings at junctions (where two or more linestring geometries meet), and calculates aggregate statistics for each unique route segment (Morgan and Lovelace, 2020), taking an object containing routes and the names of the attributes to summarize as the first and second argument:

```
route_network_scenario = overline(routes_short_scenario, attrib = "bicycle")
```

The outputs of the two preceding code chunks are summarized in Figure 13.6 below.

Transport networks with records at the segment-level, typically with attributes such as road type and width, constitute a common type of route network. Such route network datasets are available worldwide from OpenStreetMap, and can

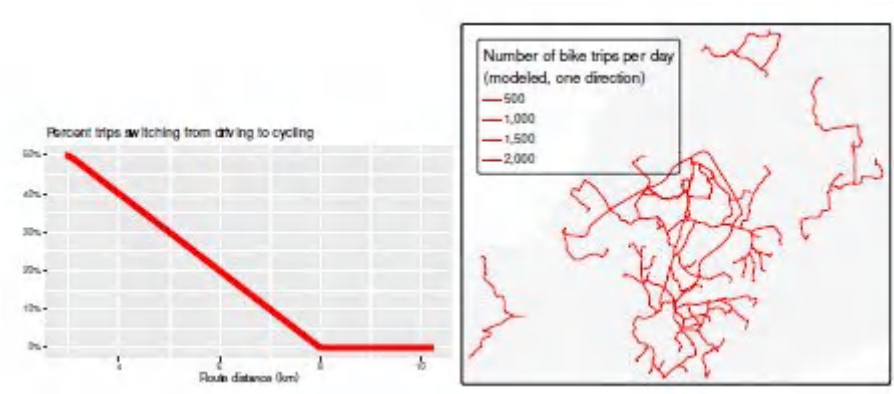

FIGURE 13.6 The percentage of car trips switching to cycling as a function of distance (left) and route network level results of this function (right).

be downloaded with packages such as **osmdata** and **osmextract**. To save time downloading and preparing OSM, we will use the `bristol_ways` object from the **spDataLarge** package, an `sf` object with LINESTRING geometries and attributes representing a sample of the transport network in the case study region (see `?bristol_ways` for details), as shown in the output below:

```
summary(bristol_ways)
#>      highway       maxspeed             ref                    geometry
#>  cycleway:1721   Length:6160        Length:6160        LINESTRING   :6160
#>  rail    :1017   Class :character   Class :character   epsg:4326    :   0
#>  road    :3422   Mode  :character   Mode  :character   +proj=long...:   0
```

The output shows that `bristol_ways` represents just over 6,000 segments on the transport network. This and other geographic networks can be represented as mathematical graphs, with nodes on the network, connected by edges. A number of R packages have been developed for dealing with such graphs, notably **igraph**. You can manually convert a route network into an `igraph` object, but the geographic attributes will be lost. To overcome this limitation of **igraph**, the **sfnetworks** package was developed (van der Meer et al., 2023). It represents networks simultaneously as graphs *and* geographic lines and has a 'tidy' syntax, as demonstrated in the following example.

```
bristol_ways$lengths = st_length(bristol_ways)
ways_sfn = as_sfnetwork(bristol_ways)
class(ways_sfn)
#> [1] "sfnetwork" "tbl_graph" "igraph"
```

```
ways_sfn
#> # A sfnetwork with 5728 nodes and 4915 edges
#> # A directed multigraph with 1013 components with spatially explicit edges
#> # Node Data:     5,728 × 1 (active)
#> # Edge Data:     4,915 × 7
#>    from    to highway maxspeed ref                              geometry lengths
#>   <int> <int> <fct>   <fct>    <fct>                     <LINESTRING [°]>     [m]
#> 1     1     2 road    <NA>     B3130 (-2.61 51.4, -2.61 51.4, -2.61 51.…    218.
#> # …
```

The output of the previous code chunk (with the final output shortened to contain only the most important eight lines due to space considerations) shows that `ways_sfn` is a composite object, containing both nodes and edges in graph and spatial form. `ways_sfn` is of class `sfnetwork`, which builds on the `igraph` class from the **igraph** package. In the example below, the 'edge betweenness', meaning the number of shortest paths passing through each edge, is calculated (see `?igraph::betweenness` for further details). The output of the edge betweenness calculation is shown in Figure 13.7, which has the cycle route network dataset calculated with the `overline()` function as an overlay for comparison. The results demonstrate that each graph edge represents a segment: the segments near the center of the road network have the highest betweenness values, whereas segments closer to central Bristol have higher cycling potential, based on these simplistic datasets.

```
ways_centrality = ways_sfn |>
  activate("edges") |>
  mutate(betweenness = tidygraph::centrality_edge_betweenness(lengths))
```

One can also find the shortest route between origins and destinations using this graph representation of the route network with the **sfnetworks** package. The methods presented in this section are relatively simple compared with what is possible. The possibilities opened-up by **sfnetworks** cannot be fully covered in this section, but it does provide a strong starting point for further exploration and research into the area. A final point is that the example dataset we used above is relatively small. It may also be worth considering how the work could adapt to larger networks: testing methods on a subset of the data, and ensuring you have enough RAM will help, although it's also worth exploring other tools that can do transport network analysis that are optimized for large networks, such as R5 (Alessandretti et al., 2022).

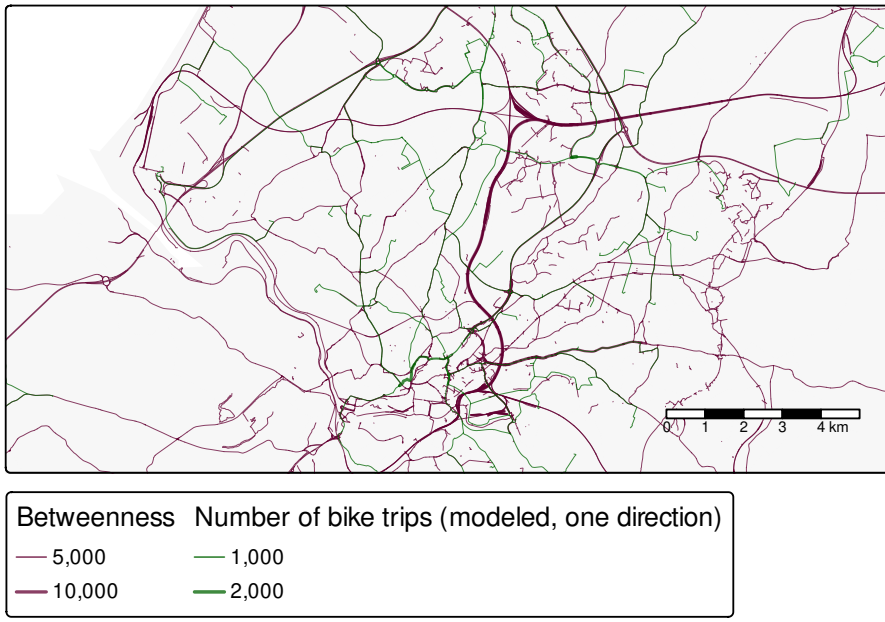

FIGURE 13.7 Route network datasets. The gray lines represent a simplified road network, with segment thickness proportional to betweenness. The green lines represent potential cycling flows (one way) calculated with the code above.

13.8 Prioritizing new infrastructure

This section demonstrates how geocomputation can create policy-relevant outcomes in the field of transport planning. We will identify promising locations for investment in sustainable transport infrastructure, using a simple approach for educational purposes.

An advantage of the data-driven approach outlined in this chapter is its modularity: each aspect can be useful on its own, and feed into wider analyses. The steps that got us to this stage included identifying short but car-dependent commuting routes (generated from desire lines) in Section 13.6 and analysis of route network characteristics with the **sfnetworks** package in Section 13.7. The final code chunk of this chapter combines these strands of analysis, by overlaying estimates of cycling potential from the previous section on top of a new dataset representing areas within a short distance of cycling infrastructure. This new dataset is created in the code chunk below which: (1) filters out the cycleway entities from the `bristol_ways` object representing the transport network,(2) 'unions' the individual LINESTRING entities of the cycleways

into a single multilinestring object (for speed of buffering),and (3) creates a 100 m buffer around them to create a polygon.

```
existing_cycleways_buffer = bristol_ways |>
  filter(highway == "cycleway") |>    # 1) filter out cycleways
  st_union() |>                       # 2) unite geometries
  st_buffer(dist = 100)               # 3) create buffer
```

The next stage is to create a dataset representing points on the network where there is high cycling potential but little provision for cycling.

```
route_network_no_infra = st_difference(
  route_network_scenario,
  route_network_scenario |> st_set_crs(st_crs(existing_cycleways_buffer)),
  existing_cycleways_buffer
)
```

The results of the preceding code chunks are shown in Figure 13.8, which shows routes with high levels of car-dependency and high cycling potential but no cycleways.

```
tmap_mode("view")
qtm(route_network_no_infra, basemaps = leaflet::providers$Esri.WorldTopoMap,
    lines.lwd = 5)
```

The method has some limitations: in reality, people do not travel to zone centroids or always use the shortest route algorithm for a particular mode. However, the results demonstrate how geographic data analysis can be used to highlight places where new investment in cycleways could be particularly beneficial, despite the simplicity of the approach. The analysis would need to be substantially expanded — including with larger input datasets — to inform transport planning design in practice.

13.9 Future directions of travel

This chapter provided a taste of the possibilities of using geocomputation for transport research, and it explored some key geographic elements that make-up a city's transport system with open data and reproducible code. The results could help plan where investment is needed.

Transport systems operate at multiple interacting levels, meaning that geocomputational methods have great potential to generate insights into how they

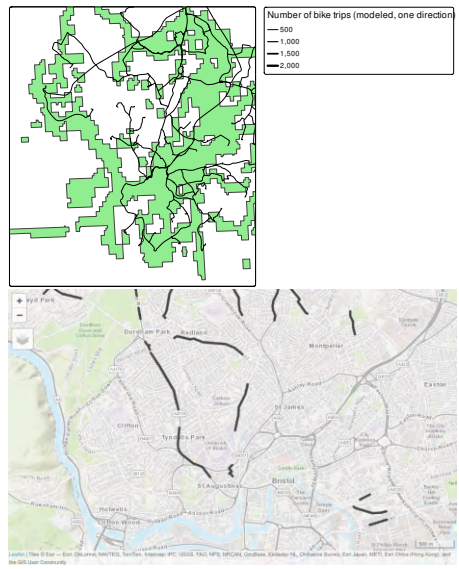

FIGURE 13.8 Potential routes along which to prioritize cycle infrastructure in Bristol to reduce car-dependency. The static map provides an overview of the overlay between existing infrastructure and routes with high car-bike switching potential (left). The screenshot the interactive map generated from the 'qtm()' function highlights Whiteladies Road as somewhere that would benefit from a new cycleway (right).

work, and the likely impacts of different interventions. There is much more that could be done in this area: it would be possible to build on the foundations presented in this chapter in many directions. Transport is the fastest growing source of greenhouse gas emissions in many countries, and it is set to become "the largest GHG emitting sector, especially in developed countries" (see EURACTIV.com). Transport-related emissions are unequally distributed across society but (unlike food and heating) are not essential for well-being. There is great potential for the sector to rapidly decarbonize through demand reduction, electrification of the vehicle fleet and the uptake of active travel modes such as walking and cycling. New technologies can reduce car-dependency by enabling more car sharing. 'Micro-mobility' systems such as dockless bike and e-scooter schemes are also emerging, creating valuable datasets in the General Bikeshare Feed Specification (GBFS) format, which can be imported and processed with the **gbfs** package. These and other changes will have large impacts on accessibility, the ability of people to reach employment and service locations that they need, something that can be quantified currently and under scenarios of change with packages such as **accessibility** packages. Further exploration of such 'transport futures' at local, regional and national levels could yield important new insights.

Methodologically, the foundations presented in this chapter could be extended by including more variables in the analysis. Characteristics of the route such as speed limits, busyness and the provision of protected cycling and walking paths could be linked to 'mode-split' (the proportion of trips made by different modes of transport). By aggregating OpenStreetMap data using buffers and geographic data methods presented in Chapters 3 and 4, for example, it would be possible to detect the presence of green space in close proximity to transport routes. Using R's statistical modeling capabilities, this could then be used to predict current and future levels of cycling, for example.

This type of analysis underlies the Propensity to Cycle Tool (PCT), a publicly accessible (see www.pct.bike) mapping tool developed in R that is being used to prioritize investment in cycling across England (Lovelace et al., 2017). Similar tools could be used to encourage evidence-based transport policies related to other topics such as air pollution and public transport access around the world.

13.10 Exercises

E1. In much of the analysis presented in the chapter, we focused on active modes, but what about driving trips?

- What proportion of trips in the `desire_lines` object are made by driving?
- What proportion of `desire_lines` have a straight line length of 5 km or more in distance?
- What proportion of trips in desire lines that are longer than 5 km in length are made by driving?
- Plot the desire lines that are both less than 5 km in length and along which more than 50% of trips are made by car.
- What do you notice about the location of these car-dependent yet short desire lines?

E2. What additional length of cycleways would be built if all the sections beyond 100 m from existing cycleways in Figure 13.8, were constructed?

E3. What proportion of trips represented in the `desire_lines` are accounted for in the `routes_short_scenario` object?

- Bonus: what proportion of all trips happen on desire lines that cross `routes_short_scenario`?

E4. The analysis presented in this chapter is designed for teaching how geocomputation methods can be applied to transport research. If you were doing this for real, in government or for a transport consultancy, what top 3 things would you do differently?

E5. Clearly, the routes identified in Figure 13.8 only provide part of the picture. How would you extend the analysis?

E6. Imagine that you want to extend the scenario by creating key *areas* (not routes) for investment in place-based cycling policies such as car-free zones, cycle parking points and reduced car parking strategy. How could raster datasets assist with this work?

- Bonus: develop a raster layer that divides the Bristol region into 100 cells (10 x 10) and estimate the average speed limit of roads in each, from the `bristol_ways` dataset (see Chapter 14).

14

Geomarketing

Prerequisites

- This chapter requires the following packages (**tmaptools** must also be installed):

```
library(sf)
library(dplyr)
library(purrr)
library(terra)
library(osmdata)
library(spDataLarge)
```

- Required data will be downloaded in due course.

As a convenience to the reader and to ensure easy reproducibility, we have made available the downloaded data in the **spDataLarge** package.

14.1 Introduction

This chapter demonstrates how the skills learned in Parts I and II can be applied to a particular domain: geomarketing (sometimes also referred to as location analysis or location intelligence). This is a broad field of research and commercial application. A typical example of geomarketing is where to locate a new shop. The aim here is to attract most visitors and, ultimately, make the most profit. There are also many non-commercial applications that can use the technique for public benefit, for example, where to locate new health services (Tomintz et al., 2008).

People are fundamental to location analysis, in particular where they are likely to spend their time and other resources. Interestingly, ecological concepts and models are quite similar to those used for store location analysis. Animals and plants can best meet their needs in certain 'optimal' locations, based on

variables that change over space (Muenchow et al. (2018); see also Chapter 15). This is one of the great strengths of geocomputation and GIScience in general: concepts and methods are transferable to other fields. Polar bears, for example, prefer northern latitudes where temperatures are lower and food (seals and sea lions) is plentiful. Similarly, humans tend to congregate in certain places, creating economic niches (and high land prices) analogous to the ecological niche of the Arctic. The main task of location analysis is to find out, based on available data, where such 'optimal locations' are for specific services. Typical research questions include:

- Where do target groups live and which areas do they frequent?
- Where are competing stores or services located?
- How many people can easily reach specific stores?
- Do existing services over- or under-utilize the market potential?
- What is the market share of a company in a specific area?

This chapter demonstrates how geocomputation can answer such questions based on a hypothetical case study and real data.

14.2 Case study: bike shops in Germany

Imagine you are starting a chain of bike shops in Germany. The stores should be placed in urban areas with as many potential customers as possible. Additionally, a hypothetical survey (invented for this chapter, not for commercial use!) suggests that single young males (aged 20 to 40) are most likely to buy your products: this is the *target audience*. You are in the lucky position to have sufficient capital to open a number of shops. But where should they be placed? Consulting companies (employing geomarketing analysts) would happily charge high rates to answer such questions. Luckily, we can do so ourselves with the help of open data and open source software. The following sections will demonstrate how the techniques learned during the first chapters of the book can be applied to undertake common steps in service location analysis:

- Tidy the input data from the German census (Section 14.3)
- Convert the tabulated census data into raster objects (Section 14.4)
- Identify metropolitan areas with high population densities (Section 14.5)
- Download detailed geographic data (from OpenStreetMap, with **osmdata**) for these areas (Section 14.6)
- Create rasters for scoring the relative desirability of different locations using map algebra (Section 14.4)

Although we have applied these steps to a specific case study, they could be generalized to many scenarios of store location or public service provision.

14.3 Tidy the input data

The German government provides gridded census data at either 1 km or 100 m resolution. The following code chunk downloads, unzips and reads in the 1 km data.

```
download.file("https://tinyurl.com/ybtpkwxz",
              destfile = "census.zip", mode = "wb")
unzip("census.zip") # unzip the files
census_de = readr::read_csv2(list.files(pattern = "Gitter.csv"))
```

Please note that `census_de` is also available from the **spDataLarge** package:

```
data("census_de", package = "spDataLarge")
```

The `census_de` object is a data frame containing 13 variables for more than 360,000 grid cells across Germany. For our work, we only need a subset of these: Easting (`x`) and Northing (`y`), number of inhabitants (population; `pop`), mean average age (`mean_age`), proportion of women (`women`) and average household size (`hh_size`). These variables are selected and renamed from German into English in the code chunk below and summarized in Table 14.1. Further, `mutate()` is used to convert values `-1` and `-9` (meaning "unknown") to `NA`.

```
# pop = population, hh_size = household size
input = select(census_de, x = x_mp_1km, y = y_mp_1km, pop = Einwohner,
                  women = Frauen_A, mean_age = Alter_D, hh_size = HHGroesse_D)
# set -1 and -9 to NA
input_tidy = mutate(input, across(.cols = c(pop, women, mean_age, hh_size),
                                  .fns =  ~ifelse(.x %in% c(-1, -9), NA, .x)))
```

14.4 Create census rasters

After the preprocessing, the data can be converted into a `SpatRaster` object (see Sections 2.3.4 and 3.3.1) with the help of the `rast()` function. When setting its `type` argument to `xyz`, the `x` and `y` columns of the input data frame should correspond to coordinates on a regular grid. All the remaining columns (here: `pop`, `women`, `mean_age`, `hh_size`) will serve as values of the raster layers (Figure 14.1; see also `code/14-location-figures.R` in our GitHub repository).

TABLE 14.1 Categories for each variable in census data from Datensatzbeschreibung...xlsx located in the downloaded file census.zip.

Class	Population	% Female	Mean Age	Household Size
1	3-250	0-40	0-40	1-2
2	250-500	40-47	40-42	2-2.5
3	500-2000	47-53	42-44	2.5-3
4	2000-4000	53-60	44-47	3-3.5
5	4000-8000	>60	>47	>3.5
6	>8000			

```
input_ras = rast(input_tidy, type = "xyz", crs = "EPSG:3035")
```

```
input_ras
#> class       : SpatRaster
#> dimensions  : 868, 642, 4  (nrow, ncol, nlyr)
#> resolution  : 1000, 1000  (x, y)
#> extent      : 4031000, 4673000, 2684000, 3552000  (xmin, xmax, ymin, ymax)
#> coord. ref. : ETRS89-extended / LAEA Europe (EPSG:3035)
#> source(s)   : memory
#> names       : pop, women, mean_age, hh_size
#> min values  :   1,     1,        1,       1
#> max values  :   6,     5,        5,       5
```

Note that we are using an equal-area projection (EPSG:3035; Lambert Equal Area Europe), i.e., a projected CRS where each grid cell has the same area, here 1000 * 1000 square meters. Since we are using mainly densities such as the number of inhabitants or the portion of women per grid cell, it is of utmost importance that the area of each grid cell is the same to avoid 'comparing apples and oranges'. Be careful with geographic CRS where grid cell areas constantly decrease in poleward directions (see also Section 2.4 and Chapter 7).

The next stage is to reclassify the values of the rasters stored in `input_ras` in accordance with the survey mentioned in Section 14.2, using the **terra** function `classify()`, which was introduced in Section 4.3.3. In the case of the population data, we convert the classes into a numeric data type using class means. Raster cells are assumed to have a population of 127 if they have a value of 1 (cells in 'class 1' contain between 3 and 250 inhabitants) and 375 if they have a value of 2 (containing 250 to 500 inhabitants), and so on (see Table

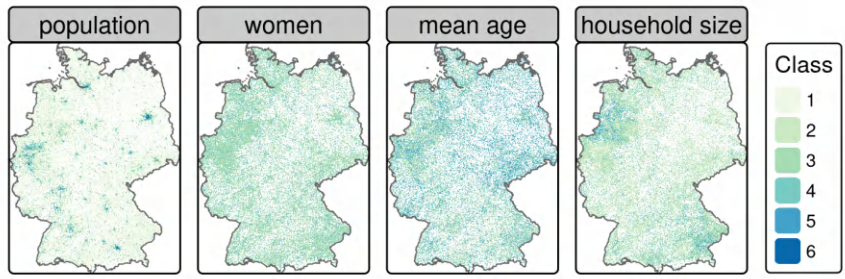

FIGURE 14.1 Gridded German census data of 2011 (see Table 14.1 for a description of the classes).

14.1). A cell value of 8000 inhabitants was chosen for 'class 6' because these cells contain more than 8000 people. Of course, these are approximations of the true population, not precise values.[1] However, the level of detail is sufficient to delineate metropolitan areas (see Section 14.5).

In contrast to the `pop` variable, representing absolute estimates of the total population, the remaining variables were reclassified as weights corresponding with weights used in the survey. Class 1 in the variable `women`, for instance, represents areas in which 0 to 40% of the population is female; these are reclassified with a comparatively high weight of 3 because the target demographic is predominantly male. Similarly, the classes containing the youngest people and highest proportion of single households are reclassified to have high weights.

```
rcl_pop = matrix(c(1, 1, 127, 2, 2, 375, 3, 3, 1250,
                   4, 4, 3000, 5, 5, 6000, 6, 6, 8000),
                 ncol = 3, byrow = TRUE)
rcl_women = matrix(c(1, 1, 3, 2, 2, 2, 3, 3, 1, 4, 5, 0),
                   ncol = 3, byrow = TRUE)
rcl_age = matrix(c(1, 1, 3, 2, 2, 0, 3, 5, 0),
                 ncol = 3, byrow = TRUE)
rcl_hh = rcl_women
rcl = list(rcl_pop, rcl_women, rcl_age, rcl_hh)
```

Note that we have made sure that the order of the reclassification matrices in the list is the same as for the elements of `input_ras`. For instance, the first element corresponds in both cases to the population. Subsequently, the `for`-loop applies the reclassification matrix to the corresponding raster layer. Finally,

[1] The potential error introduced during this reclassification stage will be explored in the exercises.

the code chunk below ensures the `reclass` layers have the same name as the layers of `input_ras`.

```
reclass = input_ras
for (i in seq_len(nlyr(reclass))) {
  reclass[[i]] = classify(x = reclass[[i]], rcl = rcl[[i]], right = NA)
}
names(reclass) = names(input_ras)


reclass # full output not shown
#> ...
#> names       :  pop, women, mean_age, hh_size
#> min values  :  127,     0,        0,       0
#> max values  : 8000,     3,        3,       3
```

14.5 Define metropolitan areas

We deliberately define metropolitan areas as pixels of 20 km^2 inhabited by more than 500,000 people. Pixels at this coarse resolution can rapidly be created using `aggregate()`, as introduced in Section 5.3.3. The command below uses the argument `fact = 20` to reduce the resolution of the result 20-fold (recall the original raster resolution was 1 km^2).

```
pop_agg = aggregate(reclass$pop, fact = 20, fun = sum, na.rm = TRUE)
summary(pop_agg)
#>       pop
#>  Min.   :    127
#>  1st Qu.:  39886
#>  Median :  66008
#>  Mean   :  99503
#>  3rd Qu.: 105696
#>  Max.   :1204870
#>  NA's   :447
```

The next stage is to keep only cells with more than half a million people.

```
pop_agg = pop_agg[pop_agg > 500000, drop = FALSE]
```

Plotting this reveals eight metropolitan regions (Figure 14.2). Each region consists of one or more raster cells. It would be nice if we could join all

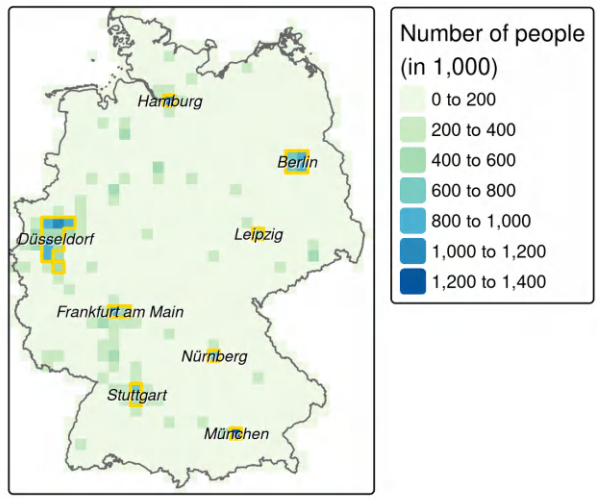

FIGURE 14.2 The aggregated population raster (resolution: 20 km) with the identified metropolitan areas (golden polygons) and the corresponding names.

cells belonging to one region. **terra**'s `patches()` command does exactly that. Subsequently, `as.polygons()` converts the raster object into spatial polygons, and `st_as_sf()` converts it into an `sf` object.

```
metros = pop_agg |>
  patches(directions = 8) |>
  as.polygons() |>
  st_as_sf()
```

The resulting eight metropolitan areas suitable for bike shops (Figure 14.2; see also `code/14-location-figures.R` for creating the figure) are still missing a name. A reverse geocoding approach can settle this problem: given a coordinate, it finds the corresponding address. Consequently, extracting the centroid coordinate of each metropolitan area can serve as an input for a reverse geocoding API. This is exactly what the `rev_geocode_OSM()` function of the **tmaptools** package expects. Setting additionally `as.data.frame` to `TRUE` will give back a `data.frame` with several columns referring to the location including the street name, house number and city. However, here, we are only interested in the name of the city.

TABLE 14.2 Result of the reverse geocoding.

City	State
Hamburg	NA
Berlin	NA
Velbert	Nordrhein-Westfalen
Leipzig	Sachsen
Frankfurt am Main	Hessen
Nürnberg	Bayern
Stuttgart	Baden-Württemberg
München	Bayern

```
metro_names = sf::st_centroid(metros, of_largest_polygon = TRUE) |>
  tmaptools::rev_geocode_OSM(as.data.frame = TRUE) |>
  select(city, town, state)
# smaller cities are returned in column town. To have all names in one column,
# we move the town name to the city column in case it is NA
metro_names = dplyr::mutate(metro_names, city = ifelse(is.na(city), town, city))
```

To make sure that the reader uses the exact same results, we have put them into **spDataLarge** as the object `metro_names`.

Overall, we are satisfied with the `City` column serving as metropolitan names (Table 14.2) apart from one exception, namely Velbert which belongs to the greater region of Düsseldorf. Hence, we replace Velbert with Düsseldorf (Figure 14.2). Umlauts like `ü` might lead to trouble further on, for example when determining the bounding box of a metropolitan area with `opq()` (see further below), which is why we avoid them.

```
metro_names = metro_names$city |>
  as.character() |>
  (\(x) ifelse(x == "Velbert", "Düsseldorf", x))() |>
  gsub("ü", "ue", x = _)
```

14.6 Points of interest

The **osmdata** package provides easy-to-use access to OSM data (see also Section 8.5). Instead of downloading shops for the whole of Germany, we restrict the query to the defined metropolitan areas, reducing computational

load and providing shop locations only in areas of interest. The subsequent code chunk does this using a number of functions including:

- `map()` (the **tidyverse** equivalent of `lapply()`), which iterates through all eight metropolitan names which subsequently define the bounding box in the OSM query function `opq()` (see Section 8.5)
- `add_osm_feature()` to specify OSM elements with a key value of `shop` (see wiki.openstreetmap.org for a list of common key:value pairs)
- `osmdata_sf()`, which converts the OSM data into spatial objects (of class `sf`)
- `while()`, which tries two more times to download the data if the download failed the first time[2]

Before running this code, please consider it will download almost two GB of data. To save time and resources, we have put the output named `shops` into **spDataLarge**. To make it available in your environment, run `data("shops", package = "spDataLarge")`.

```
shops = purrr::map(metro_names, function(x) {
  message("Downloading shops of: ", x, "\n")
  # give the server a bit time
  Sys.sleep(sample(seq(5, 10, 0.1), 1))
  query = osmdata::opq(x) |>
    osmdata::add_osm_feature(key = "shop")
  points = osmdata::osmdata_sf(query)
  # request the same data again if nothing has been downloaded
  iter = 2
  while (nrow(points$osm_points) == 0 && iter > 0) {
    points = osmdata_sf(query)
    iter = iter - 1
  }
  # return only the point features
  points$osm_points
})
```

It is highly unlikely that there are no shops in any of our defined metropolitan areas. The following `if` condition simply checks if there is at least one shop for each region. If not, we recommend to try to download the shops again for this/these specific region/s.

```
# checking if we have downloaded shops for each metropolitan area
ind = purrr::map_dbl(shops, nrow) == 0
if (any(ind)) {
  message("There are/is still (a) metropolitan area/s without any features:\n",
```

[2]The OSM-download will sometimes fail at the first attempt.

```
          paste(metro_names[ind], collapse = ", "), "\nPlease fix it!")
}
```

To make sure that each list element (an `sf` data frame) comes with the same columns[3], we only keep the `osm_id` and the `shop` columns with the help of the `map_dfr` loop which additionally combines all shops into one large `sf` object.

```
# select only specific columns
shops = purrr::map_dfr(shops, select, osm_id, shop)
```

Note: `shops` is provided in the `spDataLarge` and can be accessed as follows:

```
data("shops", package = "spDataLarge")
```

The only thing left to do is to convert the spatial point object into a raster (see Section 6.4). The `sf` object, `shops`, is converted into a raster having the same parameters (dimensions, resolution, CRS) as the `reclass` object. Importantly, the `length()` function is used here to count the number of shops in each cell.

The result of the subsequent code chunk is therefore an estimate of shop density (shops/km^2). `st_transform()` is used before `rasterize()` to ensure the CRS of both inputs match.

```
shops = sf::st_transform(shops, st_crs(reclass))
# create poi raster
poi = rasterize(x = shops, y = reclass, field = "osm_id", fun = "length")
```

As with the other raster layers (population, women, mean age, household size) the `poi` raster is reclassified into four classes (see Section 14.4). Defining class intervals is an arbitrary undertaking to a certain degree. One can use equal breaks, quantile breaks, fixed values or others. Here, we choose the Fisher-Jenks natural breaks approach which minimizes within-class variance, the result of which provides an input for the reclassification matrix.

```
# construct reclassification matrix
int = classInt::classIntervals(values(poi), n = 4, style = "fisher")
int = round(int$brks)
rcl_poi = matrix(c(int[1], rep(int[-c(1, length(int))], each = 2),
                   int[length(int)] + 1), ncol = 2, byrow = TRUE)
rcl_poi = cbind(rcl_poi, 0:3)
```

[3]This is not a given since OSM contributors are not equally meticulous when collecting data.

```
# reclassify
poi = classify(poi, rcl = rcl_poi, right = NA)
names(poi) = "poi"
```

14.7 Identify suitable locations

The only steps that remain before combining all the layers are to add `poi` to the `reclass` raster stack and remove the population layer from it. The reasoning for the latter is: First of all, we have already delineated metropolitan areas, that is areas where the population density is above average compared to the rest of Germany. Second, though it is advantageous to have many potential customers within a specific catchment area, the sheer number alone might not actually represent the desired target group. For instance, residential tower blocks are areas with a high population density but not necessarily with a high purchasing power for expensive cycle components.

```
# remove population raster and add poi raster
reclass = reclass[[names(reclass) != "pop"]] |>
  c(poi)
```

In common with other data science projects, data retrieval and 'tidying' have consumed much of the overall workload so far. With clean data, the final step — calculating a final score by summing all raster layers — can be accomplished in a single line of code.

```
# calculate the total score
result = sum(reclass)
```

For instance, a score greater than 9 might be a suitable threshold indicating raster cells where a bike shop could be placed (Figure 14.3; see also `code/14-location-figures.R`).

14.8 Discussion and next steps

The presented approach is a typical example of the normative usage of a GIS (Longley, 2015). We combined survey data with expert-based knowledge and assumptions (definition of metropolitan areas, defining class intervals,

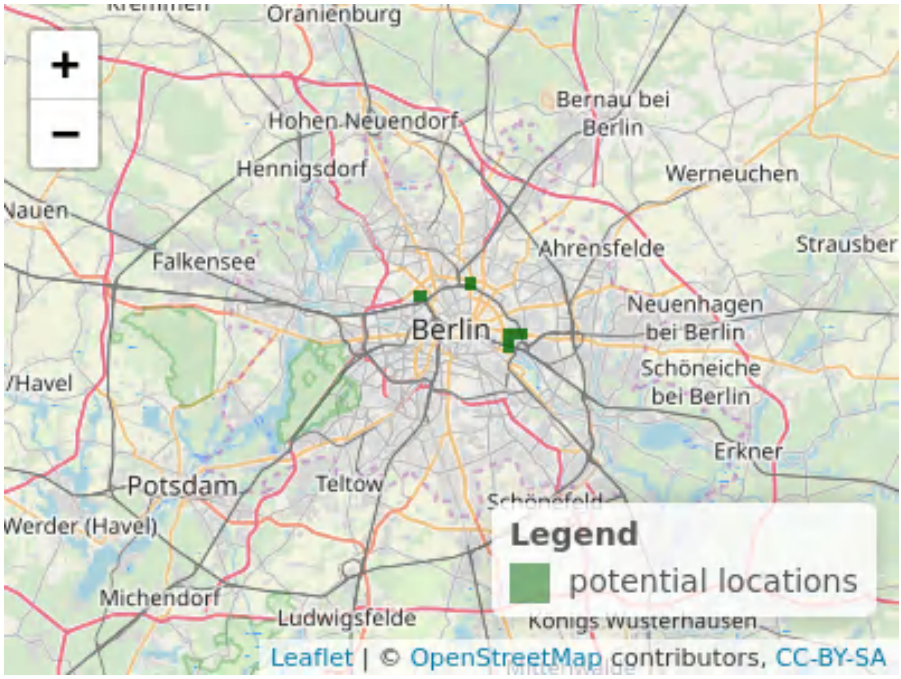

FIGURE 14.3 Suitable areas (i.e., raster cells with a score > 9) in accordance with our hypothetical survey for bike stores in Berlin.

definition of a final score threshold). This approach is less suitable for scientific research than applied analysis that provides an evidence-based indication of areas suitable for bike shops that should be compared with other sources of information. A number of changes to the approach could improve the analysis:

- We used equal weights when calculating the final scores but other factors, such as the household size, could be as important as the portion of women or the mean age
- We used all points of interest but only those related to bike shops, such as do-it-yourself, hardware, bicycle, fishing, hunting, motorcycles, outdoor and sports shops (see the range of shop values available on the OSM Wiki) may have yielded more refined results
- Data at a higher resolution may improve the output (see Exercises)
- We have used only a limited set of variables and data from other sources, such as the INSPIRE geoportal or data on cycle paths from OpenStreetMap, may enrich the analysis (see also Section 8.5)
- Interactions remained unconsidered, such as a possible relationship between the portion of men and single households

In short, the analysis could be extended in multiple directions. Nevertheless, it should have given you a first impression and understanding of how to obtain and deal with spatial data in R within a geomarketing context.

Finally, we have to point out that the presented analysis would be merely the first step of finding suitable locations. So far we have identified areas, 1 by 1 km in size, representing potentially suitable locations for a bike shop in accordance with our survey. Subsequent steps in the analysis could be taken:

- Find an optimal location based on number of inhabitants within a specific catchment area. For example, the shop should be reachable for as many people as possible within 15 minutes of traveling bike distance (catchment area routing). Thereby, we should account for the fact that the further away the people are from the shop, the more unlikely it becomes that they actually visit it (distance decay function)
- Also it would be a good idea to take into account competitors. That is, if there already is a bike shop in the vicinity of the chosen location, possible customers (or sales potential) should be distributed between the competitors (Huff, 1963; Wieland, 2017)
- We need to find suitable and affordable real estate, e.g., in terms of accessibility, availability of parking spots, desired frequency of passers-by, having big windows, etc.

14.9 Exercises

E1. Download the csv file containing inhabitant information for a 100 m cell resolution (https://www.zensus2011.de/SharedDocs/Downloads/DE/Pressemitteilung/DemografischeGrunddaten/csv_Bevoelkerung_100m_Gitter.zip?__blob=publicationFile&v=3). Please note that the unzipped file has a size of 1.23 GB. To read it into R, you can use `readr::read_csv`. This takes 30 seconds on a machine with 16 GB RAM. `data.table::fread()` might be even faster, and returns an object of class `data.table()`. Use `dplyr::as_tibble()` to convert it into a tibble. Build an inhabitant raster, aggregate it to a cell resolution of 1 km, and compare the difference with the inhabitant raster (`inh`) we have created using class mean values.

E2. Suppose our bike shop predominantly sold electric bikes to older people. Change the age raster accordingly, repeat the remaining analyses and compare the changes with our original result.

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

15

Ecology

Prerequisites

This chapter assumes you have a strong grasp of geographic data analysis and processing, covered in Chapters 2 to 5. The chapter makes use of bridges to GIS software, and spatial cross-validation, covered in Chapters 10 and 12, respectively.

The chapter uses the following packages:

```
library(sf)
library(terra)
library(dplyr)
library(data.table)        # fast data frame manipulation (used by mlr3)
library(mlr3)              # machine learning (see Chapter 12)
library(mlr3spatiotempcv)  # spatiotemporal resampling
library(mlr3tuning)        # hyperparameter tuning package
library(mlr3learners)      # interface to most important machine learning pkgs
library(paradox)           # defining hyperparameter spaces
library(ranger)            # random forest package
library(qgisprocess)       # bridge to QGIS (Chapter 10)
library(tree)              # decision tree package
library(vegan)             # community ecology package
```

15.1 Introduction

This chapter models the floristic gradient of fog oases to reveal distinctive vegetation belts that are clearly controlled by water availability. The case study provides an opportunity to bring together and extend concepts presented in previous chapters to further enhance your skills at using R for geocomputation.

Fog oases, locally called *lomas*, are vegetation formations found on mountains along the coastal deserts of Peru and Chile. Similar ecosystems can be found elsewhere, including in the deserts of Namibia and along the coasts of Yemen and Oman (Galletti et al., 2016). Despite the arid conditions and low levels of precipitation of around 30-50 mm per year on average, fog deposition increases the amount of water available to plants during austral winter, resulting in green southern-facing mountain slopes along the coastal strip of Peru. The fog, which develops below the temperature inversion caused by the cold Humboldt current in austral winter, provides the name for this habitat. Every few years, the El Niño phenomenon brings torrential rainfall to this sun-baked environment, providing tree seedlings a chance to develop roots long enough to survive the following arid conditions (Dillon et al., 2003).

Unfortunately, fog oases are heavily endangered, primarily due to agriculture and anthropogenic climate change. Evidence on the composition and spatial distribution of the native flora can support efforts to protect remaining fragments of fog oases (Muenchow et al., 2013a,b).

In this chapter you will analyze the composition and the spatial distribution of vascular plants (here referring mostly to flowering plants) on the southern slope of Mt. Mongón, a *lomas* mountain near Casma on the central northern coast of Peru (Figure 15.1). During a field study to Mt. Mongón, all vascular plants living in 100 randomly sampled 4 by 4 m^2 plots in the austral winter of 2011 were recorded (Muenchow et al., 2013a). The sampling coincided with a strong La Niña event that year, as shown in data published by the National Oceanic and Atmospheric Administration (NOAA). This led to even higher levels of aridity than usual in the coastal desert and increased fog activity on the southern slopes of Peruvian *lomas* mountains.

This chapter also demonstrates how to apply techniques covered in previous chapters to an important applied field: ecology. Specifically, we will:

- Load in needed data and compute environmental predictors (Section 15.2)
- Extract the main floristic gradient from our species composition matrix with the help of a dimension-reducing technique (ordinations; Section 15.3)
- Model the first ordination axis, i.e., the floristic gradient, as a function of environmental predictors such as altitude, slope, catchment area and NDVI (Section 15.4). For this, we will make use of a random forest model — a very popular machine learning algorithm (Breiman, 2001). To guarantee an optimal prediction, it is advisable to tune beforehand the hyperparameters with the help of spatial cross-validation (see Section 12.5.2)
- Make a spatial distribution map of the floristic composition anywhere in the study area (Section 15.4.2)

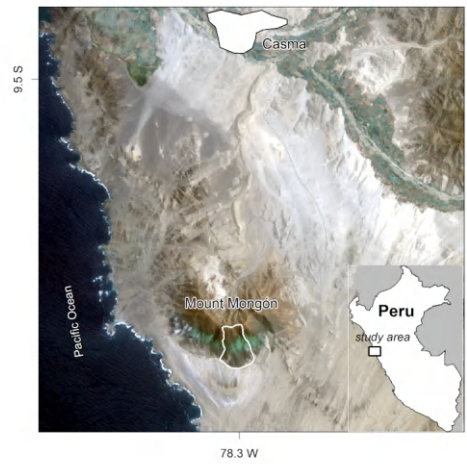

FIGURE 15.1 The Mt. Mongón study area. Figure taken from Muenchow, Schratz, and Brenning (2017).

15.2 Data and data preparation

All the data needed for the subsequent analyses is available via the **spDataLarge** package.

```
data("study_area", "random_points", "comm", package = "spDataLarge")
dem = rast(system.file("raster/dem.tif", package = "spDataLarge"))
ndvi = rast(system.file("raster/ndvi.tif", package = "spDataLarge"))
```

`study_area` is a polygon representing the outline of the study area, and `random_points` is an `sf` object containing the 100 randomly chosen sites. `comm` is a community matrix of the wide data format (Wickham, 2014) where the rows represent the visited sites in the field and the columns the observed species.[1]

```
# sites 35 to 40 and corresponding occurrences of the first five species in the
# community matrix
comm[35:40, 1:5]
#>    Alon_meri Alst_line Alte_hali Alte_porr Anth_eccr
```

[1] In statistics, this is also called a contingency table or cross-table.

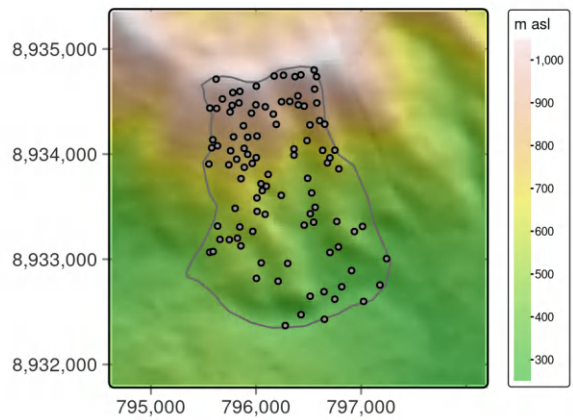

FIGURE 15.2 Study mask (polygon), location of the sampling sites (black points) and DEM in the background.

```
#> 35          0          0          0        0.0      1.000
#> 36          0          0          1        0.0      0.500
#> 37          0          0          0        0.0      0.125
#> 38          0          0          0        0.0      3.000
#> 39          0          0          0        0.0      2.000
#> 40          0          0          0        0.2      0.125
```

The values represent species cover per site, and were recorded as the area covered by a species in proportion to the site area (%; please note that one site can have >100% due to overlapping cover between individual plants). The rownames of `comm` correspond to the `id` column of `random_points`. `dem` is the digital elevation model (DEM) for the study area, and `ndvi` is the Normalized Difference Vegetation Index (NDVI) computed from the red and near-infrared channels of a Landsat scene (see Section 4.3.3 and `?spDataLarge::ndvi.tif`). Visualizing the data helps to get more familiar with it, as shown in Figure 15.2 where the `dem` is overplotted by the `random_points` and the `study_area`.

The next step is to compute variables which are not only needed for the modeling and predictive mapping (see Section 15.4.2) but also for aligning the non-metric multidimensional scaling (NMDS) axes with the main gradient in the study area, altitude and humidity, respectively (see Section 15.3).

Specifically, we compute catchment slope and catchment area from a digital elevation model using R-GIS bridges (see Chapter 10). Curvatures might also

represent valuable predictors, and in the Exercise section you can find out how they would impact the modeling result.

To compute catchment area and catchment slope, we can make use of the `sagang:sagawetnessindex` function.[2] `qgis_show_help()` returns all function parameters and default values of a specific geoalgorithm. Here, we present only a selection of the complete output.

```
# if not already done, enable the saga next generation plugin
qgisprocess::qgis_enable_plugins("processing_saga_nextgen")
# show help
qgisprocess::qgis_show_help("sagang:sagawetnessindex")
#> Saga wetness index (sagang:sagawetnessindex)
#> ...
#> ----------------
#> Arguments
#> ----------------
#>
#> DEM: Elevation
#>  Argument type:  raster
#>  Acceptable values:
#>      - Path to a raster layer
#> ...
#> SLOPE_TYPE: Type of Slope
#>  Argument type:  enum
#>  Available values:
#>      - 0: [0] local slope
#>      - 1: [1] catchment slope
#> ...
#> AREA: Catchment area
#>  Argument type:  rasterDestination
#>  Acceptable values:
#>      - Path for new raster layer
#>...
#> ----------------
#> Outputs
#> ----------------
#>
#> AREA: <outputRaster>
#>  Catchment area
#> SLOPE: <outputRaster>
```

[2] Admittedly, it is a bit unsatisfying that the only way of knowing that `sagawetnessindex` computes the desired terrain attributes is to be familiar with SAGA.

```
#>  Catchment slope
#> ...
```

Subsequently, we can specify the needed parameters using R named arguments (see Section 10.2). Remember that we can use a path to a file on disk or a `SpatRaster` living in R's global environment to specify the input raster `DEM` (see Section 10.2). Specifying 1 as the `SLOPE_TYPE` makes sure that the algorithm will return the catchment slope. The resulting rasters are saved to temporary files with an `.sdat` extension which is the native SAGA raster format.

```
# environmental predictors: catchment slope and catchment area
ep = qgisprocess::qgis_run_algorithm(
  alg = "sagang:sagawetnessindex",
  DEM = dem,
  SLOPE_TYPE = 1,
  SLOPE = tempfile(fileext = ".sdat"),
  AREA = tempfile(fileext = ".sdat"),
  .quiet = TRUE)
```

This returns a list named `ep` containing the paths to the computed output rasters. Let's read-in catchment area as well as catchment slope into a multi-layer `SpatRaster` object (see Section 2.3.4). Additionally, we will add two more raster objects to it, namely `dem` and `ndvi`.

```
# read in catchment area and catchment slope
ep = ep[c("AREA", "SLOPE")] |>
  unlist() |>
  rast()
names(ep) = c("carea", "cslope") # assign better names
origin(ep) = origin(dem) # make sure rasters have the same origin
ep = c(dem, ndvi, ep) # add dem and ndvi to the multi-layer SpatRaster object
```

Additionally, the catchment area values are highly skewed to the right (`hist(ep$carea)`). A log10-transformation makes the distribution more normal.

```
ep$carea = log10(ep$carea)
```

As a convenience to the reader, we have added `ep` to **spDataLarge**:

```
ep = rast(system.file("raster/ep.tif", package = "spDataLarge"))
```

Finally, we can extract the terrain attributes to our field observations (see also Section 6.3).

```
ep_rp = terra::extract(ep, random_points, ID = FALSE)
random_points = cbind(random_points, ep_rp)
```

15.3 Reducing dimensionality

Ordinations are a popular tool in vegetation science to extract the main information, frequently corresponding to ecological gradients, from large species-plot matrices mostly filled with 0s. However, they are also used in remote sensing, the soil sciences, geomarketing and many other fields. If you are unfamiliar with ordination techniques or in need of a refresher, have a look at Michael W. Palmer's webpage for a short introduction to popular ordination techniques in ecology and at Borcard et al. (2011) for a deeper look on how to apply these techniques in R. **vegan**'s package documentation is also a very helpful resource (`vignette(package = "vegan")`).

Principal component analysis (PCA) is probably the most famous ordination technique. It is a great tool to reduce dimensionality if one can expect linear relationships between variables, and if the joint absence of a variable in two plots (observations) can be considered a similarity. This is barely the case with vegetation data.

For one, the presence of a plant often follows a unimodal, i.e., a non-linear, relationship along a gradient (e.g., humidity, temperature or salinity) with a peak at the most favorable conditions and declining ends towards the unfavorable conditions.

Secondly, the joint absence of a species in two plots is hardly an indication for similarity. Suppose a plant species is absent from the driest (e.g., an extreme desert) and the moistest locations (e.g., a tree savanna) of our sampling. Then we really should refrain from counting this as a similarity because it is very likely that the only thing these two completely different environmental settings have in common in terms of floristic composition is the shared absence of species (except for rare ubiquitous species).

NMDS is one popular dimension-reducing technique used in ecology (von Wehrden et al., 2009). NMDS reduces the rank-based differences between distances between objects in the original matrix and distances between the ordinated objects. The difference is expressed as stress. The lower the stress value, the better the ordination, i.e., the low-dimensional representation of the original matrix. Stress values lower than 10 represent an excellent fit, stress values of around 15 are still good, and values greater than 20 represent a poor fit (McCune et al., 2002). In R, `metaMDS()` of the **vegan** package can execute a NMDS. As input, it expects a community matrix with the sites as rows and

the species as columns. Often ordinations using presence-absence data yield better results (in terms of explained variance) though the prize is, of course, a less informative input matrix (see also Exercises). `decostand()` converts numerical observations into presences and absences with 1 indicating the occurrence of a species and 0 the absence of a species. Ordination techniques such as NMDS require at least one observation per site. Hence, we need to dismiss all sites in which no species were found.

```
# presence-absence matrix
pa = vegan::decostand(comm, "pa")  # 100 rows (sites), 69 columns (species)
# keep only sites in which at least one species was found
pa = pa[rowSums(pa) != 0, ]  # 84 rows, 69 columns
```

The resulting matrix serves as input for the NMDS. `k` specifies the number of output axes, here, set to 4.[3] NMDS is an iterative procedure trying to make the ordinated space more similar to the input matrix in each step. To make sure that the algorithm converges, we set the number of steps to 500 using the `try` parameter.

```
set.seed(25072018)
nmds = vegan::metaMDS(comm = pa, k = 4, try = 500)
nmds$stress
#> ...
#> Run 498 stress 0.08834745
#> ... Procrustes: rmse 0.004100446  max resid 0.03041186
#> Run 499 stress 0.08874805
#> ... Procrustes: rmse 0.01822361  max resid 0.08054538
#> Run 500 stress 0.08863627
#> ... Procrustes: rmse 0.01421176  max resid 0.04985418
#> *** Solution reached
#> 0.08831395
```

A stress value of 9 represents a very good result, which means that the reduced ordination space represents the large majority of the variance of the input matrix. Overall, NMDS puts objects that are more similar (in terms of species composition) closer together in ordination space. However, as opposed to most other ordination techniques, the axes are arbitrary and not necessarily ordered by importance (Borcard et al., 2011). However, we already know that humidity represents the main gradient in the study area (Muenchow et al., 2013a, 2017). Since humidity is highly correlated with elevation, we rotate the NMDS axes in accordance with elevation (see also `?MDSrotate` for more details on rotating

[3] One way of choosing `k` is to try `k` values between 1 and 6 and then using the result which yields the best stress value (McCune et al., 2002).

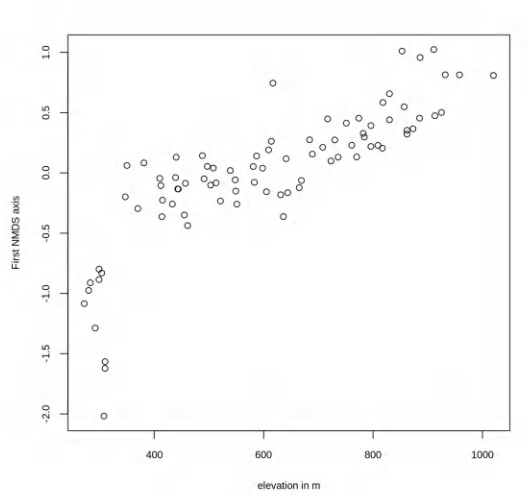

FIGURE 15.3 Plotting the first NMDS axis against altitude.

NMDS axes). Plotting the result reveals that the first axis is, as intended, clearly associated with altitude (Figure 15.3).

```
elev = dplyr::filter(random_points, id %in% rownames(pa)) |>
  dplyr::pull(dem)
# rotating NMDS in accordance with altitude (proxy for humidity)
rotnmds = vegan::MDSrotate(nmds, elev)
# extracting the first two axes
sc = vegan::scores(rotnmds, choices = 1:2, display = "sites")
# plotting the first axis against altitude
plot(y = sc[, 1], x = elev, xlab = "elevation in m",
     ylab = "First NMDS axis", cex.lab = 0.8, cex.axis = 0.8)
```

The scores of the first NMDS axis represent the different vegetation formations, i.e., the floristic gradient, appearing along the slope of Mt. Mongón. To spatially visualize them, we can model the NMDS scores with the previously created predictors (Section 15.2), and use the resulting model for predictive mapping (see Section 15.4).

15.4 Modeling the floristic gradient

To predict the floristic gradient spatially, we use a random forest model. Random forest models are frequently applied in environmental and ecological modeling, and often provide the best results in terms of predictive performance (Hengl et al., 2018; Schratz et al., 2019). Here, we shortly introduce decision trees and bagging, since they form the basis of random forests. We refer the reader to James et al. (2013) for a more detailed description of random forests and related techniques.

To introduce decision trees by example, we first construct a response-predictor matrix by joining the rotated NMDS scores to the field observations (`random_points`). We will also use the resulting data frame for the **mlr3** modeling later on.

```
# construct response-predictor matrix
# id- and response variable
rp = data.frame(id = as.numeric(rownames(sc)), sc = sc[, 1])
# join the predictors (dem, ndvi and terrain attributes)
rp = inner_join(random_points, rp, by = "id")
```

Decision trees split the predictor space into a number of regions. To illustrate this, we apply a decision tree to our data using the scores of the first NMDS axis as the response (`sc`) and altitude (`dem`) as the only predictor.

```
tree_mo = tree::tree(sc ~ dem, data = rp)
plot(tree_mo)
text(tree_mo, pretty = 0)
```

The resulting tree consists of three internal nodes and four terminal nodes (Figure 15.4). The first internal node at the top of the tree assigns all observations which are below 328.5 m to the left and all other observations to the right branch. The observations falling into the left branch have a mean NMDS score of -1.198. Overall, we can interpret the tree as follows: the higher the elevation, the higher the NMDS score becomes. This means that the simple decision tree has already revealed four distinct floristic assemblages. For a more in-depth interpretation, please refer to Section 15.4.2. Decision trees have a tendency to overfit, that is, they mirror too closely the input data including its noise which in turn leads to bad predictive performances (Section 12.4, James et al., 2013). Bootstrap aggregation (bagging) is an ensemble technique that can help to overcome this problem. Ensemble techniques simply combine the predictions of multiple models. Thus, bagging takes repeated samples from the same input data and averages the predictions. This reduces the variance

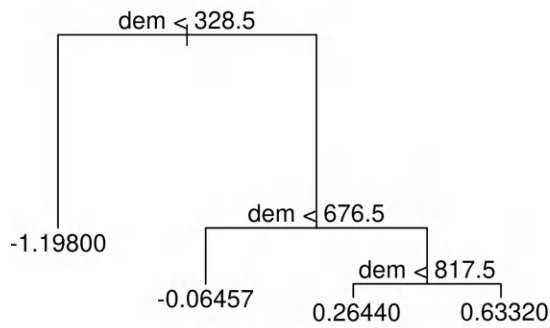

FIGURE 15.4 Simple example of a decision tree with three internal nodes and four terminal nodes.

and overfitting with the result of a much better predictive accuracy compared to decision trees. Finally, random forests extend and improve bagging by decorrelating trees which is desirable since averaging the predictions of highly correlated trees shows a higher variance and thus lower reliability than averaging predictions of decorrelated trees (James et al., 2013). To achieve this, random forests use bagging, but in contrast to the traditional bagging where each tree is allowed to use all available predictors, random forests only use a random sample of all available predictors.

15.4.1 mlr3 building blocks

The code in this section largely follows the steps we have introduced in Section 12.5.2. The only differences are the following:

1. The response variable is numeric, hence a regression task will replace the classification task of Section 12.5.2
2. Instead of the AUROC which can only be used for categorical response variables, we will use the root mean squared error (RMSE) as the performance measure
3. We use a Random Forest model instead of a Support Vector Machine which naturally goes along with different hyperparameters
4. We are leaving the assessment of a bias-reduced performance measure as an exercise to the reader (see Exercises). Instead, we show how to tune hyperparameters for (spatial) predictions

Remember that 125,500 models were necessary to retrieve bias-reduced performance estimates when using 100-repeated 5-fold spatial cross-validation and a random search of 50 iterations in Section 12.5.2. In the hyperparameter tuning level, we found the best hyperparameter combination which in turn was used in the outer performance level for predicting the test data of a specific spatial

partition (see also Figure 12.6). This was done for five spatial partitions, and repeated a 100 times yielding in total 500 optimal hyperparameter combinations. Which one should we use for making spatial distribution maps? The answer is simple: none at all. Remember, the tuning was done to retrieve a bias-reduced performance estimate, not to do the best possible spatial prediction. For the latter, one estimates the best hyperparameter combination from the complete dataset. This means, the inner hyperparameter tuning level is no longer needed, which makes perfect sense since we are applying our model to new data (unvisited field observations) for which the true outcomes are unavailable, hence testing is impossible in any case. Therefore, we tune the hyperparameters for a good spatial prediction on the complete dataset via a 5-fold spatial CV with one repetition.

Having already constructed the input variables (`rp`), we are all set for specifying the **mlr3** building blocks (task, learner, and resampling). For specifying a spatial task, we use again the **mlr3spatiotempcv** package (Schratz et al., 2021, Section 12.5), and since our response (`sc`) is numeric, we use a regression task.

```
# create task
task = mlr3spatiotempcv::as_task_regr_st(
  select(rp, -id, -spri),
  target = "sc",
  id = "mongon"
)
```

Using an `sf` object as the backend automatically provides the geometry information needed for the spatial partitioning later on. Additionally, we got rid of the columns `id` and `spri`, since these variables should not be used as predictors in the modeling. Next, we go on to construct a random forest learner from the **ranger** package (Wright and Ziegler, 2017).

```
lrn_rf = lrn("regr.ranger", predict_type = "response")
```

As opposed to, for example, Support Vector Machines (see Section 12.5.2), random forests often already show good performances when used with the default values of their hyperparameters (which may be one reason for their popularity). Still, tuning often moderately improves model results, and thus is worth the effort (Probst et al., 2018). In random forests, the hyperparameters `mtry`, `min.node.size` and `sample.fraction` determine the degree of randomness, and should be tuned (Probst et al., 2018). `mtry` indicates how many predictor variables should be used in each tree. If all predictors are used, then this corresponds in fact to bagging (see beginning of Section 15.4). The `sample.fraction` parameter specifies the fraction of observations to be used in each tree. Smaller fractions lead to greater diversity, and thus less correlated trees which often

is desirable (see above). The `min.node.size` parameter indicates the number of observations a terminal node should at least have (see also Figure 15.4). Naturally, as trees and computing time become larger, the lower the `min.node.size`.

Hyperparameter combinations will be selected randomly but should fall inside specific tuning limits (created with `paradox::ps()`). `mtry` should range between 1 and the number of predictors (4) , `sample.fraction` should range between 0.2 and 0.9 and `min.node.size` should range between 1 and 10 (Probst et al., 2018).

```
# specifying the search space
search_space = paradox::ps(
  mtry = paradox::p_int(lower = 1, upper = ncol(task$data()) - 1),
  sample.fraction = paradox::p_dbl(lower = 0.2, upper = 0.9),
  min.node.size = paradox::p_int(lower = 1, upper = 10)
)
```

Having defined the search space, we are all set for specifying our tuning via the `AutoTuner()` function. Since we deal with geographic data, we will again make use of spatial cross-validation to tune the hyperparameters (see Sections 12.4 and 12.5). Specifically, we will use a 5-fold spatial partitioning with only one repetition (`rsmp()`). In each of these spatial partitions, we run 50 models (`trm()`) while using randomly selected hyperparameter configurations (`tnr()`) within predefined limits (`seach_space`) to find the optimal hyperparameter combination (see also Section 12.5.2 and https://mlr3book.mlr-org.com/chapters/chapter4/hyperparameter_optimization.html#sec-autotuner, Bischl et al., 2024). The performance measure is the root mean squared error (RMSE).

```
autotuner_rf = mlr3tuning::auto_tuner(
  learner = lrn_rf,
  resampling = mlr3::rsmp("spcv_coords", folds = 5), # spatial partitioning
  measure = mlr3::msr("regr.rmse"), # performance measure
  terminator = mlr3tuning::trm("evals", n_evals = 50), # specify 50 iterations
  search_space = search_space, # predefined hyperparameter search space
  tuner = mlr3tuning::tnr("random_search") # specify random search
)
```

Calling the `train()`-method of the `AutoTuner`-object finally runs the hyperparameter tuning, and will find the optimal hyperparameter combination for the specified parameters.

```
# hyperparameter tuning
set.seed(24092024)
autotuner_rf$train(task)
```

```
autotuner_rf$tuning_result
#>     mtry sample.fraction min.node.size learner_param_vals  x_domain regr.rmse
#>    <int>           <num>         <int>             <list>    <list>     <num>
#> 1:     4           0.784            10          <list[4]> <list[3]>     0.382
```

15.4.2 Predictive mapping

The tuned hyperparameters can now be used for the prediction. To do so, we only need to run the `predict` method of our fitted `AutoTuner` object.

```
# predicting using the best hyperparameter combination
autotuner_rf$predict(task)
#> <PredictionRegr> for 84 observations:
#>  row_ids  truth response
#>        1 -1.084   -1.176
#>        2 -0.975   -1.176
#>        3 -0.912   -1.168
#>      ---    ---      ---
#>       82  0.814    0.594
#>       83  0.814    0.746
#>       84  0.808    0.807
```

The `predict` method will apply the model to all observations used in the modeling. Given a multi-layer `SpatRaster` containing rasters named as the predictors used in the modeling, `terra::predict()` will also make spatial distribution maps, i.e., predict to new data.

```
pred = terra::predict(ep, model = autotuner_rf, fun = predict)
```

In case, `terra::predict()` does not support a model algorithm, you can still make the predictions manually.

```
newdata = as.data.frame(as.matrix(ep))
colSums(is.na(newdata))  # 0 NAs
# but assuming there were 0s results in a more generic approach
ind = rowSums(is.na(newdata)) == 0
tmp = autotuner_rf$predict_newdata(newdata = newdata[ind, ], task = task)
newdata[ind, "pred"] = data.table::as.data.table(tmp)[["response"]]
pred_2 = ep$dem
# now fill the raster with the predicted values
pred_2[] = newdata$pred
```

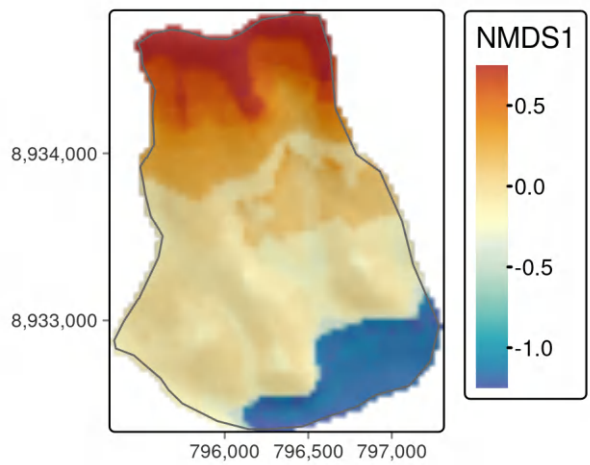

FIGURE 15.5 Predictive mapping of the floristic gradient clearly revealing distinct vegetation belts.

```
# check if terra and our manual prediction is the same
all(values(pred - pred_2) == 0)
```

The predictive mapping clearly reveals distinct vegetation belts (Figure 15.5). Please refer to Muenchow et al. (2013b) for a detailed description of vegetation belts on *lomas* mountains. The blue color tones represent the so-called *Tillandsia* belt. *Tillandsia* is a highly adapted genus especially found in high quantities at the sandy and quite desertic foot of *lomas* mountains. The yellow color tones refer to a herbaceous vegetation belt with a much higher plant cover compared to the *Tillandsia* belt. The orange colors represent the bromeliad belt, which features the highest species richness and plant cover. It can be found directly beneath the temperature inversion (ca. 750-850 m asl) where humidity due to fog is highest. Water availability naturally decreases above the temperature inversion, and the landscape becomes desertic again with only a few succulent species (succulent belt; red colors). Interestingly, the spatial prediction clearly reveals that the bromeliad belt is interrupted, which is a very interesting finding we would have not detected without the predictive mapping.

15.5 Conclusions

In this chapter we have ordinated the community matrix of the *lomas* Mt. Mongón with the help of a NMDS (Section 15.3). The first axis, representing the main floristic gradient in the study area, was modeled as a function of environmental predictors which partly were derived through R-GIS bridges (Section 15.2). The **mlr3** package provided the building blocks to spatially tune the hyperparameters `mtry`, `sample.fraction` and `min.node.size` (Section 15.4.1). The tuned hyperparameters served as input for the final model which in turn was applied to the environmental predictors for a spatial representation of the floristic gradient (Section 15.4.2). The result demonstrates spatially the astounding biodiversity in the middle of the desert. Since *lomas* mountains are heavily endangered, the prediction map can serve as basis for informed decision-making on delineating protection zones, and making the local population aware of the uniqueness found in their immediate neighborhood.

In terms of methodology, a few additional points could be addressed:

- It would be interesting to also model the second ordination axis, and to subsequently find an innovative way of visualizing jointly the modeled scores of the two axes in one prediction map
- If we were interested in interpreting the model in an ecologically meaningful way, we should probably use (semi-)parametric models (Muenchow et al., 2013a; Zuur et al., 2009, 2017). However, there are at least approaches that help to interpret machine learning models such as random forests (see, e.g., https://mlr-org.com/posts/2018-04-30-interpretable-machine-learning-iml-and-mlr/)
- A sequential model-based optimization (SMBO) might be preferable to the random search for hyperparameter optimization used in this chapter (Probst et al., 2018)

Finally, please note that random forest and other machine learning models are frequently used in a setting with much more observations and predictors than used in this Chapter, and where it is unclear which variables and variable interactions contribute to explaining the response. Additionally, the relationships might be highly non-linear. In our use case, the relationship between response and predictors is pretty clear, there is only a slight amount of non-linearity and the number of observations and predictors is low. Hence, it might be worth trying a linear model. A linear model is much easier to explain and understand than a random forest model, and therefore preferred (law of parsimony). Additionally it is computationally less demanding (see Exercises). If the linear model cannot cope with the degree of non-linearity present in the data, one could also try a generalized additive model (GAM). The point here is that the toolbox of a data scientist consists of more than one tool, and it

is your responsibility to select the tool best suited for the task or purpose at hand. Here, we wanted to introduce the reader to random forest modeling and how to use the corresponding results for predictive mapping purposes. For this purpose, a well-studied dataset with known relationships between response and predictors, is appropriate. However, this does not imply that the random forest model has returned the best result in terms of predictive performance.

15.6 Exercises

The solutions assume the following packages are attached (other packages will be attached when needed):

E1. Run a NMDS using the percentage data of the community matrix. Report the stress value and compare it to the stress value as retrieved from the NMDS using presence-absence data. What might explain the observed difference?

E2. Compute all the predictor rasters we have used in the chapter (catchment slope, catchment area), and put them into a `SpatRaster`-object. Add `dem` and `ndvi` to it. Next, compute profile and tangential curvature and add them as additional predictor rasters (hint: `grass7:r.slope.aspect`). Finally, construct a response-predictor matrix. The scores of the first NMDS axis (which were the result when using the presence-absence community matrix) rotated in accordance with elevation represent the response variable, and should be joined to `random_points` (use an inner join). To complete the response-predictor matrix, extract the values of the environmental predictor raster object to `random_points`.

E3. Retrieve the bias-reduced RMSE of a random forest and a linear model using spatial cross-validation. The random forest modeling should include the estimation of optimal hyperparameter combinations (random search with 50 iterations) in an inner tuning loop. Parallelize the tuning level. Report the mean RMSE and use a boxplot to visualize all retrieved RMSEs. Please note that this exercise is best solved using the mlr3 functions `benchmark_grid()` and `benchmark()` (see https://mlr3book.mlr-org.com/perf-eval-cmp.html#benchmarking for more information).

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

16

Conclusion

16.1 Introduction

Like the introduction, this concluding chapter contains a few code chunks. The aim is to synthesize the contents of the book, with reference to recurring themes/concepts, and to inspire future directions of application and development. The chapter has no prerequisites. However, you may get more out of it if you have read and attempted the exercises in Part I (Foundations), tried more advances approaches in Part II (Extensions), and considered how geocomputation can help you solve work, research or other problems, with reference to the chapters in Part III (Applications).

The chapter is organized as follows. Section 16.2 discusses the wide range of options for handling geographic data in R. Choice is a key feature of open source software; the section provides guidance on choosing between the various options. Section 16.3 describes gaps in the book's contents and explains why some areas of research were deliberately omitted, while others were emphasized. Next, Section 16.4 provides advice on how to ask good questions when you get stuck, and how to search for solutions online. Section 16.5 answers the following question: having read this book, where to go next? Section 16.6 returns to the wider issues raised in Chapter 1. In it we consider geocomputation as part of a wider 'open source approach' that ensures methods are publicly accessible, reproducible and supported by collaborative communities. This final section of the book also provides some pointers on how to get involved.

16.2 Package choice

A feature of R, and open source software in general, is that there are often multiple ways to achieve the same result. The code chunk below illustrates this by using three functions, covered in Chapters 3 and 5, to combine the 16 regions of New Zealand into a single geometry:

```
library(spData)
nz_u1 = sf::st_union(nz)
nz_u2 = aggregate(nz["Population"], list(rep(1, nrow(nz))), sum)
nz_u3 = dplyr::summarise(nz, t = sum(Population))
identical(nz_u1, nz_u2$geometry)
#> [1] TRUE
identical(nz_u1, nz_u3$geom)
#> [1] TRUE
```

Although the classes, attributes and column names of the resulting objects `nz_u1` to `nz_u3` differ, their geometries are identical, as verified using the base R function `identical()`.[1] Which to use? It depends: the former only processes the geometry data contained in `nz` so is faster, while the other options performed attribute operations, which may be useful for subsequent steps. Whether to use the base R function `aggregate()` or the **dplyr** function `summarise()` is a matter of preference, with the latter being more readable for many.

The wider point is that there are often multiple options to choose from when working with geographic data in R, even within a single package. The range of options grows further when more R packages are considered: you could achieve the same result using the older **sp** package, for example. However, based on our goal of providing good advice, we recommend using the more recent, more performant and future-proof **sf** package. The same applies for all packages showcased in this book, although it can be helpful (when not distracting) to be aware of alternatives and being able to justify your choice of software.

A common choice, for which there is no simple answer, is between **tidyverse** and base R for geocomputation. The following code chunk, for example, shows **tidyverse** and base R ways to extract the `Name` column from the `nz` object, as described in Chapter 3:

```
library(dplyr)                          # attach a tidyverse package
nz_name1 = nz["Name"]                   # base R approach
nz_name2 = nz |>                        # tidyverse approach
  select(Name)
identical(nz_name1$Name, nz_name2$Name) # check results
#> [1] TRUE
```

This raises the question: which to use? The answer is: it depends. Each approach has advantages: base R tends to be stable, well-known, and has

[1] The first operation, undertaken by the function `st_union()`, creates an object of class `sfc` (a simple feature column). The latter two operations create `sf` objects, each of which *contains* a simple feature column. Therefore, it is the geometries contained in simple feature columns, not the objects themselves, that are identical.

minimal dependencies, which is why it is often preferred for software (package) development. The tidyverse approach, on the other hand, is often preferred for interactive programming. Choosing between the two approaches is therefore a matter of preference and application.

While this book covers commonly needed functions — such as the base R `[` subsetting operator and the **dplyr** function `select()` demonstrated in the code chunk above — there are many other functions for working with geographic data, from other packages, that have not been mentioned. Chapter 1 mentions 20+ influential packages for working with geographic data, and only a handful of these are covered in the book. Hundreds of other packages are available for working with geographic data in R, and many more are developed each year. As of 2024, there are more than 160 packages mentioned in the Spatial Task View and countless functions for geographic data analysis are developed each year.

The rate of evolution in R's spatial ecosystem may be fast, but there are strategies to deal with the wide range of options. Our advice is to start by learning one approach *in depth* but to have a general understanding of the *breadth* of available options. This advice applies equally to solving geographic problems with R, as it does to other fields of knowledge and application. Section 16.5 covers developments in other languages.

Of course, some packages perform better than others for the *same* task, in which case it's important to know which to use. In the book we have aimed to focus on packages that are future-proof (they will work long into the future), high performance (relative to other R packages), well maintained (with user and developer communities surrounding them) and complementary. There are still overlaps in the packages we have used, as illustrated by the diversity of packages for making maps, as highlighted in Chapter 9, for example.

Overlapping functionality can be good. A new package with similar (but not identical) functionality compared to an existing package can increase resilience, performance (partly driven by friendly competition and mutual learning between developers) and choice, both of which are key benefits of doing geocomputation with open source software. In this context, deciding which combination of **sf**, **tidyverse**, **terra** and other packages to use should be made with knowledge of alternatives. The **sp** ecosystem that **sf** superseded, for example, can do many of the things covered in this book and, due to its age, is built on by many other packages. At the time of writing in 2024, 463 packages `Depend` on or `Import` **sp**, up slightly from 452 in October 2018, showing that its data structures are widely used and have been extended in many directions. The equivalent numbers for **sf** are 69 in 2018 and 431 in 2024, highlighting that the package is future-proof and has a growing user base and developer community (Bivand, 2021). Although best known for point pattern analysis, the **spatstat** package also supports raster and other vector geometries and provides powerful functionality for spatial statistics and more (Baddeley and

Turner, 2005). It may also be worth researching new alternatives that are under development if you have needs that are not met by established packages.

16.3 Gaps and overlaps

Geocomputation is a big area, so there are inevitably gaps in this book. We have been selective, deliberately highlighting certain topics, techniques and packages, while omitting others. We have tried to emphasize topics that are most commonly needed in real-world applications such as geographic data operations, basics of coordinate reference systems, read/write data operations and visualization techniques. Some topics and themes appear repeatedly, with the aim of building essential skills for geocomputation, and showing you how to go further, into more advanced topics and specific applications.

We deliberately omitted some topics that are covered in-depth elsewhere. Statistical modeling of spatial data such as point pattern analysis, spatial interpolation (e.g., kriging) and spatial regression, for example, are mentioned in the context of machine learning in Chapter 12 but not covered in detail. There are already excellent resources on these methods, including statistically orientated chapters in Pebesma and Bivand (2023b) and books on point pattern analysis (Baddeley et al., 2015), Bayesian techniques applied to spatial data (Gómez-Rubio, 2020; Moraga, 2023), and books focused on particular applications such as health (Moraga, 2019) and wildfire severity analysis (Wimberly, 2023). Other topics which received limited attention were remote sensing and using R alongside (rather than as a bridge to) dedicated GIS software. There are many resources on these topics, including a discussion on remote sensing in R, Wegmann et al. (2016) and the GIS-related teaching materials available from Marburg University.

We focused on machine learning rather than spatial statistical inference in Chapters 12 and 15 because of the abundance of quality resources on the topic. These resources include Zuur et al. (2009), Zuur et al. (2017) which focus on ecological use cases, and freely available teaching material and code on *Geostatistics & Open-source Statistical Computing* hosted at css.cornell.edu/faculty/dgr2. *R for Geographic Data Science* provides an introduction to R for geographic data science and modeling.

We have largely omitted geocomputation on 'big data' by which we mean datasets that do not fit on a high-spec laptop. This decision is justified by the fact that the majority of geographic datasets that are needed for common research or policy applications *do* fit on consumer hardware, large high-resolution remote sensing datasets being a notable exception (see Section 10.8). It is possible to get more RAM on your computer or to temporarily 'rent'

compute power available on platforms such as GitHub Codespaces, which can be used to run the code in this book. Furthermore, learning to solve problems on small datasets is a prerequisite to solving problems on huge datasets and the emphasis in this book is getting started, and the skills you learn here will be useful when you move to bigger datasets. Analysis of 'big data' often involves extracting a small amount of data from a database for a specific statistical analysis. Spatial databases, covered in Chapter 10, can help with the analysis of datasets that do not fit in memory. 'Earth observation cloud back-ends' can be accessed from R with the **openeo** package (Section 10.8.2). If you need to work with big geographic datasets, we also recommend exploring projects such as Apache Sedona and emerging file formats such as GeoParquet.

16.4 Getting help

Geocomputation is a large and challenging field, making issues and temporary blockers to work near inevitable. In many cases you may just 'get stuck' at a particular point in your data analysis workflow facing cryptic error messages that are hard to debug. Or you may get unexpected results with few clues about what is going on. This section provides pointers to help you overcome such problems, by clearly defining the problem, searching for existing knowledge on solutions and, if those approaches do not solve the problem, through the art of asking good questions.

When you get stuck at a particular point, it is worth first taking a step back and working out which approach is most likely to solve the issue. Trying each of the following steps — skipping steps already taken — provides a structured approach to problem-solving:

1. Define exactly what you are trying to achieve, starting from first principles (and often a sketch, as outlined below)
2. Diagnose exactly where in your code the unexpected results arise, by running and exploring the outputs of individual lines of code and their individual components (you can run individual parts of a complex command by selecting them with a cursor and pressing Ctrl+Enter in RStudio, for example)
3. Read the documentation of the function that has been diagnosed as the 'point of failure' in the previous step. Simply understanding the required inputs to functions, and running the examples that are often provided at the bottom of help pages, can help solve a surprisingly large proportion of issues (run the command `?terra::rast` and scroll down to the examples that are worth reproducing when getting started with the function, for example)

4. If reading R's built-in documentation, as outlined in the previous step, does not help to solve the problem, it is probably time to do a broader search online to see if others have written about the issue you're seeing. See a list of places to search for help below
5. If all the previous steps above fail, and you cannot find a solution from your online searches, it may be time to compose a question with a reproducible example and post it in an appropriate place

Steps 1 to 3 outlined above are fairly self-explanatory but, due to the vastness of the internet and multitude of search options, it is worth considering effective search strategies before deciding to compose a question.

16.4.1 Searching for solutions online

Search engines are a logical place to start for many issues. 'Googling it' can in some cases result in the discovery of blog posts, forum messages and other online content about the precise issue you're having. Simply typing in a clear description of the problem/question is a valid approach here, but it is important to be specific (e.g., with reference to function and package names and input dataset sources if the problem is dataset-specific). You can also make online searches more effective by including additional detail:

- Use quotation marks to maximize the chances that 'hits' relate to the exact issue you're having by reducing the number of results returned. For example, if you try and fail to save a GeoJSON file in a location that already exists, you will get an error containing the message "GDAL Error 6: DeleteLayer() not supported by this dataset". A specific search query such as `"GDAL Error 6" sf` is more likely to yield a solution than searching for `GDAL Error 6` without the quotation marks
- Set time restraints, for example only returning content created within the last year can be useful when searching for help on an evolving package
- Make use of additional search engine features, for example restricting searches to content hosted on CRAN with site:r-project.org

16.4.2 Places to search for (and ask) for help

In cases where online searches do not yield a solution, it is worth asking for help. There are many forums where you can do this, including:

- R's Special Interest Group on Geographic data email list (R-SIG-GEO)
- The GIS Stackexchange website at gis.stackexchange.com
- The large and general purpose programming Q&A site stackoverflow.com
- Online forums associated with a particular entity, such as the Posit Community, the rOpenSci Discuss web forum and forums associated with particular software tools such as the Stan forum

- Software development platforms such as GitHub, which hosts issue trackers for the majority of R-spatial packages and also, increasingly, built-in discussion pages such as that created to encourage discussion (not just bug reporting) around the **sfnetworks** package (see luukvdmeer/sfnetworks/discussions)
- Online chat rooms and forums associated with communities such as the rOpenSci and the geocompx community (which has a Discord server where you can ask questions), of which this book is a part

16.4.3 Reproducible examples with reprex

In terms of asking a good question, a clearly stated question supported by an accessible and fully reproducible example is key (see also https://r4ds.hadley.nz/workflow-help.html). It is also helpful, after showing the code that 'did not work' from the user's perspective, to explain what you would like to see. A very useful tool for creating reproducible examples is the **reprex** package. To highlight unexpected behavior, you can write completely reproducible code that demonstrates the issue and then use the `reprex()` function to create a copy of your code that can be pasted into a forum or other online space.

Imagine you are trying to create a map of the world with blue sea and green land. You could simply ask how to do this in one of the places outlined in the previous section. However, it is likely that you will get a better response if you provide a reproducible example of what you have tried so far. The following code creates a map of the world with blue sea and green land, but the land is not filled in:

```
library(sf)
library(spData)
plot(st_geometry(world), col = "green")
```

If you post this code in a forum, it is likely that you will get a more specific and useful response. For example, someone might respond with the following code, which demonstrably solves the problem, as illustrated in Figure 16.1:

```
library(sf)
library(spData)
# use the bg argument to fill in the land
plot(st_geometry(world), col = "green", bg = "lightblue")
```

Exercise for the reader: copy the above code, run the command `reprex::reprex()` (or paste the command into the `reprex()` function call) and paste the output into a forum or other online space.

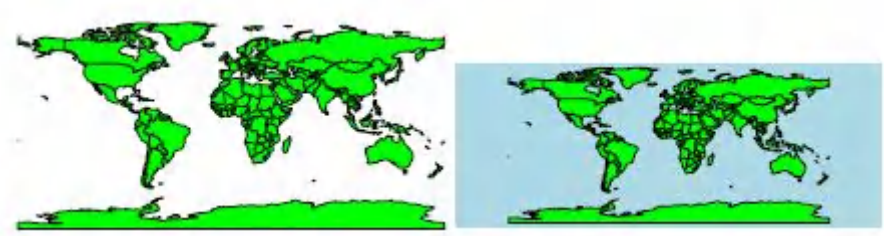

FIGURE 16.1 A map of the world with green land, illustrating a question with a reproducible example (left) and the solution (right).

A strength of open source and collaborative approaches to geocomputation is that they generate a vast and ever evolving body on knowledge, of which this book is a part. Demonstrating your own efforts to solve a problem, and providing a reproducible example of the problem, is a way of contributing to this body of knowledge.

16.4.4 Defining and sketching the problem

In some cases, you may not be able to find a solution to your problem online, or you may not be able to formulate a question that can be answered by a search engine. The best starting point in such cases, or when developing a new geocomputational methodology, may be a pen and paper (or equivalent digital sketching tools such as Excalidraw and tldraw which allow collaborative sketching and rapid sharing of ideas). During the most creative early stages of methodological development work, software *of any kind* can slow down your thoughts and direct them away from important abstract thoughts. Framing the question with mathematics is also highly recommended, with reference to a minimal example that you can sketch 'before and after' versions of numerically. If you have the skills and if the problem warrants it, describing the approach algebraically can in some cases help develop effective implementations.

16.5 Where to go next?

As indicated in Section 16.3, the book has covered only a fraction of the R's geographic ecosystem, and there is much more to discover. We have progressed quickly, from geographic data models in Chapter 2, to advanced applications in Chapter 15. Consolidation of skills learned, discovery of new packages and approaches for handling geographic data, and application of the methods to new datasets and domains are suggested future directions. This section expands on this general advice by suggesting specific 'next steps', highlighted in **bold** below.

In addition to learning about further geographic methods and applications with R, for example with reference to the work cited in the previous section, deepening your understanding of **R itself** is a logical next step. R's fundamental classes such as `data.frame` and `matrix` are the foundation of **sf** and **terra** classes, so studying them will improve your understanding of geographic data. This can be done with reference to documents that are part of R, and which can be found with the command `help.start()` and additional resources on the subject such as those by Wickham (2019) and Chambers (2016).

Another software-related direction for future learning is **discovering geocomputation with other languages**. There are good reasons for learning R as a language for geocomputation, as described in Chapter 1, but it is not the only option.[2] It would be possible to study *Geocomputation with: Python*, *C++*, *JavaScript*, *Scala* or *Rust* in equal depth. Each has evolving geospatial capabilities. **rasterio**, for example, is a Python package with similar functionality as the **terra** package used in this book. See *Geocomputation with Python*, for an introduction to geocomputation with Python.

Dozens of geospatial libraries have been developed in C++, including well-known libraries such as GDAL and GEOS, and less well-known libraries such as the **Orfeo Toolbox** for processing remote sensing (raster) data. **Turf.js** is an example of the potential for doing geocomputation with JavaScript. GeoTrellis provides functions for working with raster and vector data in the Java-based language Scala. And WhiteBoxTools provides an example of a rapidly evolving command line GIS implemented in Rust. Each of these packages/libraries/languages has advantages for geocomputation and there are many more to discover, as documented in the curated list of open source geospatial resources Awesome-Geospatial.

There is more to geocomputation than software, however. We can recommend **exploring and learning new research topics and methods** from academic and theoretical perspectives. Many methods that have been written about have yet to be implemented. Learning about geographic methods and potential applications can therefore be rewarding, before writing any code. An example of geographic methods that are increasingly implemented in R is sampling strategies for scientific applications. A next step in this case is to read-up on relevant articles in the area such as Brus (2018), which is accompanied by reproducible code and tutorial content hosted at github.com/DickBrus/TutorialSampling4DSM.

[2] R's strengths are particularly relevant to our definition of geocomputation due to its emphasis on scientific reproducibility, widespread use in academic research and unparalleled support for statistical modeling of geographic data. Furthermore, we advocate learning one language for geocomputation in depth before delving into other languages/frameworks because of the costs associated with context switching, and R is an excellent starting point on your geocomputational journey.

16.6 The open source approach

This is a technical book, so it makes sense for the next steps, outlined in the previous section, to also be technical. However, there are wider issues worth considering in this final section, which returns to our definition of geocomputation. One of the elements of the term introduced in Chapter 1 was that geographic methods should have a positive impact. Of course, how to define and measure 'positive' is a subjective, philosophical question that is beyond the scope of this book. Regardless of your worldview, consideration of the impacts of geocomputational work is a useful exercise: the potential for positive impacts can provide a powerful motivation for future learning and, conversely, new methods can open-up many possible fields of application. These considerations lead to the conclusion that geocomputation is part of a wider 'open source approach'.

Section 1.1 presented other terms that mean roughly the same thing as geocomputation, including geographic data science (GDS) and 'GIScience'. Both capture the essence of working with geographic data, but geocomputation has advantages: it concisely captures the 'computational' way of working with geographic data advocated in this book — implemented in code and therefore encouraging reproducibility — and builds on desirable ingredients of its early definition (Openshaw and Abrahart, 2000):

- The *creative* use of geographic data
- Application to *real-world problems*
- Building 'scientific' tools
- Reproducibility

We added the final ingredient: reproducibility was barely mentioned in early work on geocomputation, yet a strong case can be made for it being a vital component of the first two ingredients.

Reproducibility:

- Encourages *creativity* by shifting the focus away from the basics (which are readily available through shared code) and toward applications
- Discourages people from 'reinventing the wheel': there is no need to redo what others have done if their methods can be used by others
- Makes research more conducive to real-world applications, by enabling anyone in any sector to apply one's methods in new areas

If reproducibility is the defining asset of geocomputation (or command line GIS), it is worth considering what makes it reproducible. This brings us to the 'open source approach', which has three main components:

- A command line interface (CLI), encouraging scripts recording geographic work to be shared and reproduced
- Open source software, which can be inspected and potentially improved by anyone in the world
- An active user and developer community, which collaborates and self-organizes to build complementary and modular tools

Like the term geocomputation, the open source approach is more than a technical entity. It is a community composed of people interacting daily with shared aims: to produce high-performance tools, free from commercial or legal restrictions, that are accessible for anyone to use. The open source approach to working with geographic data has advantages that transcend the technicalities of how the software works, encouraging learning, collaboration and an efficient division of labor.

There are many ways to engage in this community, especially with the emergence of code hosting sites, such as GitHub, which encourage communication and collaboration. A good place to start is simply browsing through some of the source code, 'issues' and 'commits' in a geographic package of interest. A quick glance at the `r-spatial/sf` GitHub repository, which hosts the code underlying the **sf** package, shows that 100+ people have contributed to the codebase and documentation. Dozens more people have contributed by asking questions and by contributing to 'upstream' packages that **sf** uses. More than 1,500 issues have been closed on its issue tracker, representing a huge amount of work to make **sf** faster, more stable and user-friendly. This example, from just one package out of dozens, shows the scale of the intellectual operation underway to make R a highly effective and continuously evolving language for geocomputation.

It is instructive to watch the incessant development activity happen in public fora such as GitHub, but it is even more rewarding to become an active participant. This is one of the greatest features of the open source approach: it encourages people to get involved. This book is a result of the open source approach: it was motivated by the amazing developments in R's geographic capabilities over the last two decades, but made practically possible by dialogue and code-sharing on platforms for collaboration. We hope that in addition to disseminating useful methods for working with geographic data, this book inspires you to take a more open source approach.

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

Bibliography

Abelson, H., Sussman, G. J., and Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, second edition.

Adams, R. and Bischof, L. (1994). Seeded region growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):641–647.

Alessandretti, L., Natera Orozco, L. G., Battiston, F., Saberi, M., and Szell, M. (2022). Multimodal urban mobility and multilayer transport networks. *Environment and Planning B: Urban Analytics and City Science*, page 23998083221108190.

Appel, M. and Pebesma, E. (2019). On-demand processing of data cubes from satellite image collections with the gdalcubes library. *Data*, 4(3).

Baddeley, A., Rubak, E., and Turner, R. (2015). *Spatial Point Patterns: Methodology and Applications with R*. CRC Press.

Baddeley, A. and Turner, R. (2005). Spatstat: An R package for analyzing spatial point patterns. *Journal of statistical software*, 12(6):1–42.

Bellos, A. (2011). *Alex's Adventures in Numberland*. Bloomsbury Paperbacks, London.

Bischl, B., Lang, M., Kotthoff, L., Schiffner, J., Richter, J., Studerus, E., Casalicchio, G., and Jones, Z. M. (2016). Mlr: Machine Learning in R. *Journal of Machine Learning Research*, 17(170):1–5.

Bischl, B., Sonabend, R., Kotthoff, L., and Lang, M. (2024). *Applied Machine Learning Using Mlr3 in R*. CRC Press.

Bivand, R. (2000). Using the R statistical data analysis language on GRASS 5.0 GIS database files. *Computers & Geosciences*, 26(9):1043–1052.

Bivand, R. (2001). More on Spatial Data Analysis. *R News*, 1(3):13–17.

Bivand, R. (2003). Approaches to Classes for Spatial Data in R. In Hornik, K., Leisch, F., and Zeileis, A., editors, *Proceedings of DSC*.

Bivand, R. (2016a). *Rgrass7: Interface Between GRASS 7 Geographical Information System and R*. R package.

Bivand, R. (2016b). *Spgrass6: Interface between GRASS 6 and R*. R package.

Bivand, R. (2017). *Spdep: Spatial Dependence: Weighting Schemes, Statistics and Models*. R package.

Bivand, R. (2021). Progress in the R ecosystem for representing and handling spatial data. *Journal of Geographical Systems*, 23(4):515–546.

Bivand, R. (2023). *rgrass: Interface Between GRASS Geographical Information System and R*. R package version 0.3-9.

Bivand, R. and Gebhardt, A. (2000). Implementing functions for spatial statistical analysis using the language. *Journal of Geographical Systems*, 2(3):307–317.

Bivand, R., Keitt, T., and Rowlingson, B. (2023a). *rgdal: Bindings for the Geospatial Data Abstraction Library*. R package version 1.6-7.

Bivand, R. and Lewin-Koh, N. (2017). *Maptools: Tools for Reading and Handling Spatial Objects*. R package.

Bivand, R. and Neteler, M. (2000). Open source geocomputation: Using the R data analysis language integrated with GRASS GIS and PostgreSQL data base systems. In Neteler, M. and Bivand, R. S., editors, *Proceedings of the 5th International Conference on GeoComputation*.

Bivand, R., Nowosad, J., and Lovelace, R. (2023b). *spData: Datasets for Spatial Analysis*. R package version 2.3.0.

Bivand, R., Pebesma, E., and Gómez-Rubio, V. (2013). *Applied Spatial Data Analysis with R*. Springer.

Bivand, R. and Rundel, C. (2023). *rgeos: Interface to Geometry Engine - Open Source (GEOS)*. R package version 0.6-4.

Blangiardo, M. and Cameletti, M. (2015). *Spatial and Spatio-temporal Bayesian Models with R-INLA*. John Wiley & Sons, Ltd, Chichester, UK.

Böhner, J. and Selige, T. (2006). Spatial prediction of soil attributes using terrain analysis and climate regionalisation. In Böhner, J., McCloy, K.R., and Strobl, J., editors, *SAGA - Analysis and Modelling Applications*, page 19. Goettinger Geographische Abhandlungen, Goettingen.

Böhner, J., Selige, T., and Ringeler, A. (2006). Image segmentation using representativeness analysis and region growing. In Böhner, Jürgen, McCloy, K.R., and Strobl, J., editors, *SAGA - Analysis and Modelling Applications*, page 10. Goettinger Geographische Abhandlungen, Goettingen.

Bondaruk, B., Roberts, S. A., and Robertson, C. (2020). Assessing the state of the art in Discrete Global Grid Systems: OGC criteria and present functionality. *Geomatica*, 74(1):9–30.

Borcard, D., Gillet, F., and Legendre, P. (2011). *Numerical Ecology with R*. Use R! Springer, New York.

Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1):5–32.

Brenning, A. (2012a). *RPyGeo: ArcGIS Geoprocessing in R via Python*. R package.

Brenning, A. (2012b). Spatial cross-validation and bootstrap for the assessment of prediction rules in remote sensing: The R package sperrorest. pages 5372–5375. IEEE.

Brenning, A., Bangs, D., and Becker, M. (2022). *RSAGA: SAGA Geoprocessing and Terrain Analysis*. R package version 1.4.0.

Brewer, C. A. (2015). *Designing Better Maps: A Guide for GIS Users*. Esri Press, Redlands, California, second edition.

Bristol City Council (2015). Deprivation in Bristol 2015. Technical report, Bristol City Council.

Brus, D. J. (2018). Sampling for digital soil mapping: A tutorial supported by R scripts. *Geoderma*.

Bucklin, D. and Basille, M. (2018). Rpostgis: Linking R with a PostGIS Spatial Database. *The R Journal*.

Burrough, P. A., McDonnell, R., and Lloyd, C. D. (2015). *Principles of Geographical Information Systems*. Oxford University Press, Oxford, New York, third edition.

Cawley, G. C. and Talbot, N. L. (2010). On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11(Jul):2079–2107.

Chambers, J. M. (2016). *Extending R*. CRC Press.

Cheshire, J. and Lovelace, R. (2015). Spatial data visualisation with R. In Brunsdon, C. and Singleton, A., editors, *Geocomputation*, pages 1–14. SAGE Publications.

Clementini, E. and Di Felice, P. (1995). A comparison of methods for representing topological relationships. *Information Sciences - Applications*, 3(3):149–178.

Conrad, O., Bechtel, B., Bock, M., Dietrich, H., Fischer, E., Gerlitz, L., Wehberg, J., Wichmann, V., and Böhner, J. (2015). System for Automated Geoscientific Analyses (SAGA) v. 2.1.4. *Geosci. Model Dev.*, 8(7):1991–2007.

Cooley, D. (2020). *Sfheaders: Converts between R Objects and Simple Feature Objects*. R package.

Coombes, M. G., Green, A. E., and Openshaw, S. (1986). An Efficient Algorithm to Generate Official Statistical Reporting Areas: The Case of the 1984

Travel-to-Work Areas Revision in Britain. *The Journal of the Operational Research Society*, 37(10):943.

Coppock, J. T. and Rhind, D. W. (1991). The history of GIS. *Geographical Information Systems: Principles and Applications, vol. 1.*, 1(1):21–43.

de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer Science & Business Media.

Diggle, P. and Ribeiro, P. J. (2007). *Model-Based Geostatistics*. Springer.

Dillon, M. O., Nakazawa, M., and Leiva, S. G. (2003). The Lomas formations of coastal Peru: Composition and biogeographic history. In Haas, J. and Dillon, M. O., editors, *El Niño in Peru: Biology and Culture over 10,000 Years*, pages 1–9. Field Museum of Natural History, Chicago.

Douglas, D. H. and Peucker, T. K. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122.

Dunnington, D. (2021). *ggspatial: Spatial Data Framework for ggplot2*. R package.

Dunnington, D., Vanderhaeghe, F., Caha, J., and Muenchow, J. (2024). *R package qgisprocess: use QGIS processing algorithms*. R package version 0.3.0.

Eddelbuettel, D. and Balamuta, J. J. (2018). Extending R with C++: A Brief Introduction to Rcpp. *The American Statistician*, 72(1):28–36.

Egenhofer, M. and Herring, J. (1990). A mathematical framework for the definition of topological relations. In *Proc. the Fourth International Symposium on Spatial Data Handing*, pages 803–813.

Galletti, C. S., Turner, B. L., and Myint, S. W. (2016). Land changes and their drivers in the cloud forest and coastal zone of Dhofar, Oman, between 1988 and 2013. *Regional Environmental Change*, 16(7):2141–2153.

Gelfand, A. E., Diggle, P., Guttorp, P., and Fuentes, M. (2010). *Handbook of Spatial Statistics*. CRC Press.

Gillespie, C. and Lovelace, R. (2016). *Efficient R Programming: A Practical Guide to Smarter Programming*. O'Reilly Media.

Giraud, T. (2021). *Mapsf: Thematic Cartography*. R package.

Gómez-Rubio, V. (2020). *Bayesian Inference with INLA*. CRC Press.

Gonçalves, J., Pôças, I., Marcos, B., Mücher, C., and Honrado, J. P. (2019). SegOptim—A new R package for optimizing object-based image analyses

of high-spatial resolution remotely-sensed data. *International Journal of Applied Earth Observation and Geoinformation*, 76:218–230.

Goovaerts, P. (1997). *Geostatistics for Natural Resources Evaluation*. Applied Geostatistics Series. Oxford University Press, New York.

Graser, A. and Olaya, V. (2015). Processing: A Python Framework for the Seamless Integration of Geoprocessing Tools in QGIS. 4(4).

Grolemund, G. and Wickham, H. (2016). *R for Data Science*. O'Reilly Media.

Hengl, T. (2007). *A Practical Guide to Geostatistical Mapping of Environmental Variables*. Publications Office, Luxembourg.

Hengl, T., Nussbaum, M., Wright, M. N., Heuvelink, G. B., and Gräler, B. (2018). Random forest as a generic framework for predictive modeling of spatial and spatio-temporal variables. *PeerJ*, 6:e5518.

Hijmans, R. J. (2016). *Geosphere: Spherical Trigonometry*. R package.

Hijmans, R. J. (2023a). *geodata: Download Geographic Data*. R package version 0.5-9.

Hijmans, R. J. (2023b). *raster: Geographic Data Analysis and Modeling*. R package version 3.6-26.

Hijmans, R. J. (2023c). *terra: Spatial Data Analysis*. R package version 1.7-55.

Hollander, Y. (2016). *Transport Modelling for a Complete Beginner*. CTthink!

Horni, A., Nagel, K., and Axhausen, K. W. (2016). *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press.

Huff, D. L. (1963). A Probabilistic Analysis of Shopping Center Trade Areas. *Land Economics*, 39(1):81–90.

Ince, E. S., Barthelmes, F., Reißland, S., Elger, K., Förste, C., Flechtner, F., and Schuh, H. (2019). ICGEM – 15 years of successful collection and distribution of global gravitational models, associated services, and future plans. *Earth System Science Data*, 11(2):647–674.

Jafari, E., Gemar, M. D., Juri, N. R., and Duthie, J. (2015). Investigation of Centroid Connector Placement for Advanced Traffic Assignment Models with Added Network Detail. *Transportation Research Record: Journal of the Transportation Research Board*, 2498:19–26.

James, G., Witten, D., Hastie, T., and Tibshirani, R., editors (2013). *An Introduction to Statistical Learning: With Applications in R*. Number 103 in Springer Texts in Statistics. Springer, New York.

Jasiewicz, J. and Stepinski, T. F. (2013). Geomorphons — a pattern recognition approach to classification and mapping of landforms. *Geomorphology*, 182:147–156.

Jenny, B., Šavrič, B., Arnold, N. D., Marston, B. E., and Preppernau, C. A. (2017). A guide to selecting map projections for world and hemisphere maps. In Lapaine, M. and Usery, L., editors, *Choosing a Map Projection*, pages 213–228. Springer.

Jeworutzki, S. (2023). *cartogram: Create Cartograms with R*. R package version 0.3.0.

Kahle, D. and Wickham, H. (2013). Ggmap: Spatial Visualization with ggplot2. *The R Journal*, 5:144–161.

Kaiser, M. and Morin, T. (1993). Algorithms for computing centroids. *Computers & Operations Research*, 20(2):151–165.

Karatzoglou, A., Smola, A., Hornik, K., and Zeileis, A. (2004). Kernlab - An S4 Package for Kernel Methods in R. *Journal of Statistical Software*, 11(9).

Knuth, D. E. (1974). Computer Programming As an Art. *Commun. ACM*, 17(12):667–673.

Krug, R. M., Roura-Pascual, N., and Richardson, D. M. (2010). Clearing of invasive alien plants under different budget scenarios: Using a simulation model to test efficiency. *Biological invasions*, 12(12):4099–4112.

Lahn, F. (2021). *Openeo: Client Interface for 'openEO' Servers*. R package.

Lamigueiro, O. P. (2018). *Displaying Time Series, Spatial, and Space-Time Data with R*. Chapman & Hall/CRC, Boca Raton, second edition.

Landa, M. (2008). New GUI for GRASS GIS based on wxPython. *Departament of Geodesy and Cartography*, pages 1–17.

Landau, W. M. (2021). The targets R package: A dynamic Make-like function-oriented pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*, 6(57):2959.

Lang, M., Binder, M., Richter, J., Schratz, P., Pfisterer, F., Coors, S., Au, Q., Casalicchio, G., Kotthoff, L., and Bischl, B. (2019). Mlr3: A modern object-oriented machine learning framework in R. *Journal of Open Source Software*, 4(44):1903.

Liu, J.-G. and Mason, P. J. (2009). *Essential Image Processing and GIS for Remote Sensing*. Wiley-Blackwell, Chichester, West Sussex, UK, Hoboken, NJ.

Livingstone, D. N. (1992). *The Geographical Tradition: Episodes in the History of a Contested Enterprise*. John Wiley & Sons Ltd, Oxford, UK ; Cambridge, USA.

Loecher, M. and Ropkins, K. (2015). RgoogleMaps and loa: Unleashing R Graphics Power on Map Tiles. *Journal of Statistical Software*, 63:1–18.

Longley, P. (2015). *Geographic Information Science & Systems*. Wiley, Hoboken, NJ, fourth edition.

Longley, P., Brooks, S. M., McDonnell, R., and MacMillan, B., editors (1998). *Geocomputation: A Primer*. Wiley, Chichester, England; New York.

Lovelace, R. (2021). Open source tools for geographic analysis in transport planning. *Journal of Geographical Systems*, 23(4):547–578.

Lovelace, R. and Dumont, M. (2016). *Spatial Microsimulation with R*. CRC Press.

Lovelace, R., Félix, R., and Carlino, D. (2022). Jittering: A Computationally Efficient Method for Generating Realistic Route Networks from Origin-Destination Data. *Findings*, page 33873.

Lovelace, R., Goodman, A., Aldred, R., Berkoff, N., Abbas, A., and Woodcock, J. (2017). The Propensity to Cycle Tool: An open source online system for sustainable transport planning. *Journal of Transport and Land Use*, 10(1).

Majure, J. J. and Gebhardt, A. (2016). *Sgeostat: An Object-Oriented Framework for Geostatistical Modeling in S+*. R package.

Maling, D. H. (1992). *Coordinate Systems and Map Projections*. Pergamon Press, Oxford ; New York, second edition.

Massicotte, P. and South, A. (2023). *rnaturalearth: World Map Data from Natural Earth*. R package version 0.3.4, https://github.com/ropensci/rnaturalearth.

McCune, B., Grace, J. B., and Urban, D. L. (2002). *Analysis of Ecological Communities*. MjM Software Design, Gleneden Beach, Oregon, second edition.

Meulemans, W., Dykes, J., Slingsby, A., Turkay, C., and Wood, J. (2017). Small Multiples with Gaps. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):381–390.

Meyer, H., Reudenbach, C., Hengl, T., Katurji, M., and Nauss, T. (2018). Improving performance of spatio-temporal machine learning models using forward feature selection and target-oriented validation. *Environmental Modelling & Software*, 101:1–9.

Miller, H. J. (2004). Tobler's first law and spatial analysis. *Annals of the Association of American Geographers*, 94(2).

Moraga, P. (2019). *Geospatial Health Data: Modeling and Visualization with R-INLA and Shiny*. CRC Press.

Moraga, P. (2023). *Spatial Statistics for Data Science: Theory and Practice with R*. Chapman & Hall/CRC, Boca Raton, Florida, 1st edition.

Moreno-Monroy, A. I., Lovelace, R., and Ramos, F. R. (2017). Public transport and school location impacts on educational inequalities: Insights from São Paulo. *Journal of Transport Geography*.

Morgan, M. and Lovelace, R. (2020). Travel flow aggregation: Nationally scalable methods for interactive and online visualisation of transport behaviour at the road network level. *Environment & Planning B: Planning & Design*, 48(6).

Morgan, M., Young, M., Lovelace, R., and Hama, L. (2019). OpenTripPlanner for R. *Journal of Open Source Software*, 4(44):1926.

Morgan-Wall, T. (2021). *Rayshader: Create Maps and Visualize Data in 2D and 3D*. R package.

Muenchow, J., Bräuning, A., Rodríguez, E. F., and von Wehrden, H. (2013a). Predictive mapping of species richness and plant species' distributions of a Peruvian fog oasis along an altitudinal gradient. *Biotropica*, 45(5):557–566.

Muenchow, J., Brenning, A., and Richter, M. (2012). Geomorphic process rates of landslides along a humidity gradient in the tropical Andes. *Geomorphology*, 139–140:271–284.

Muenchow, J., Dieker, P., Kluge, J., Kessler, M., and von Wehrden, H. (2018). A review of ecological gradient research in the Tropics: Identifying research gaps, future directions, and conservation priorities. *Biodiversity and Conservation*, 27(2):273–285.

Muenchow, J., Hauenstein, S., Bräuning, A., Bäumler, R., Rodríguez, E. F., and von Wehrden, H. (2013b). Soil texture and altitude, respectively, largely determine the floristic gradient of the most diverse fog oasis in the Peruvian desert. *Journal of Tropical Ecology*, 29(05):427–438.

Muenchow, J., Schratz, P., and Brenning, A. (2017). RQGIS: Integrating R with QGIS for statistical geocomputing. *The R Journal*, 9(2):409–428.

Murrell, P. (2016). *R Graphics*. CRC Press, second edition.

Neteler, M. and Mitasova, H. (2008). *Open Source GIS: A GRASS GIS Approach*. Springer, New York, third edition.

Nolan, D. and Lang, D. T. (2014). *XML and Web Technologies for Data Sciences with R*. Use R! Springer, New York.

Nowosad, J. and Lovelace, R. (2023). *spDataLarge: Large datasets for spatial analysis*. R package version 2.0.9.

Nowosad, J. and Stepinski, T. F. (2022). Extended SLIC superpixels algorithm for applications to non-imagery geospatial rasters. *International Journal of Applied Earth Observation and Geoinformation*, 112:102935.

Obe, R. O. and Hsu, L. S. (2015). *PostGIS in Action*. Manning, Shelter Island, NY, second edition.

Office for National Statistics (2014). Workplace Zones: A new geography for workplace statistics - Datasets. https://data.gov.uk/dataset/workplace-zones-a-new-geography-for-workplace-statistics3.

Open Geospatial Consortium (2019). Well-known text representation of coordinate reference systems. Implementation Standard 18-010r7, Open Geospatial Consortium.

Openshaw, S. and Abrahart, R. J., editors (2000). *Geocomputation*. CRC Press, London; New York.

O'Rourke, J. (1998). *Computational Geometry in C*. Cambridge University Press, Cambridge, UK; New York, second edition.

Padgham, M., Rudis, B., Lovelace, R., Salmon, M., and Maspons, J. (2023). *osmdata: Import OpenStreetMap Data as Simple Features or Spatial Objects*. R package version 0.2.5.

Pawley, S. (2023). *Rsagacmd: Linking R with the Open-Source 'SAGA-GIS' Software*. R package version 0.4.2.

Pebesma, E. (2018). Simple features for R: Standardized support for spatial vector data. *The R Journal*, 10(1).

Pebesma, E. (2021). *{stars}: Spatiotemporal Arrays, Raster and Vector Data Cubes*. R package.

Pebesma, E. and Bivand, R. (2005). Classes and methods for spatial data in R. *R news*, 5(2):9–13.

Pebesma, E. and Bivand, R. (2023a). *sp: Classes and Methods for Spatial Data*. R package version 2.1-1.

Pebesma, E. and Bivand, R. (2023b). *Spatial Data Science: With Applications in R*. CRC Press.

Pebesma, E. and Graeler, B. (2023). *gstat: Spatial and Spatio-Temporal Geostatistical Modelling, Prediction and Simulation*. R package version 2.1-1.

Pebesma, E., Mailund, T., and Hiebert, J. (2016). Measurement Units in R. *The R Journal*, 8(2):486–494.

Pebesma, E., Nüst, D., and Bivand, R. (2012). The R software environment in reproducible geoscientific research. *Eos, Transactions American Geophysical Union*, 93(16):163–163.

Pedersen, T. L. and Robinson, D. (2020). *Gganimate: A Grammar of Animated Graphics*. R package.

Pereira, R. H. M., Saraiva, M., Herszenhut, D., Braga, C. K. V., and Conway, M. W. (2021). R5r: Rapid Realistic Routing on Multimodal Transport Networks with R^5 in R. *Findings*, page 21262.

Pezanowski, S., MacEachren, A. M., Savelyev, A., and Robinson, A. C. (2018). SensePlace3: A geovisual framework to analyze place–time–attribute information in social media. *Cartography and Geographic Information Science*, 45(5):420–437.

Probst, P., Wright, M., and Boulesteix, A.-L. (2018). Hyperparameters and Tuning Strategies for Random Forest. *arXiv:1804.03515 [cs, stat]*.

Qiu, F., Zhang, C., and Zhou, Y. (2012). The Development of an Areal Interpolation ArcGIS Extension and a Comparative Study. *GIScience & Remote Sensing*, 49(5):644–663.

R Core Team (2021). *An Introduction to R*.

Ripley, B. D. (2001). Spatial Statistics in R. *R News*, 1(2):14–15.

Rodrigue, J.-P., Comtois, C., and Slack, B. (2013). *The Geography of Transport Systems*. Routledge, London, New York, third edition.

Roussel, J.-R., Auty, D., Coops, N. C., Tompalski, P., Goodbody, T. R., Meador, A. S., Bourdon, J.-F., de Boissieu, F., and Achim, A. (2020). lidR: An r package for analysis of airborne laser scanning (ALS) data. *Remote Sensing of Environment*, 251:112061.

Rowlingson, B., Baddeley, A., Turner, R., and Diggle, P. (2003). Rasp: A Package for Spatial Statistics. In Hornik, K., editor, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*.

Rowlingson, B. and Diggle, P. (2017). *Splancs: Spatial and Space-Time Point Pattern Analysis*. R package.

Rowlingson, B. S. and Diggle, P. J. (1993). Splancs: Spatial point pattern analysis code in S-plus. *Computers & Geosciences*, 19(5):627–655.

Šavrič, B., Jenny, B., and Jenny, H. (2016). Projection Wizard – An Online Map Projection Selection Tool. *The Cartographic Journal*, 53(2):177–185.

Schramm, M., Pebesma, E., Milenković, M., Foresta, L., Dries, J., Jacob, A., Wagner, W., Mohr, M., Neteler, M., Kadunc, M., et al. (2021). The openeo api–harmonising the use of earth observation cloud services using virtual data cube functionalities. *Remote Sensing*, 13(6):1125.

Schratz, P., Becker, M., Lang, M., and Brenning, A. (2021). Mlr3spatiotempcv: Spatiotemporal resampling methods for machine learning in R. *arXiv preprint arXiv:2110.12674*.

Schratz, P., Muenchow, J., Iturritxa, E., Richter, J., and Brenning, A. (2019). Hyperparameter tuning and performance assessment of statistical

and machine-learning algorithms using spatial data. *Ecological Modelling*, 406:109–120.

Shen, J., Chen, M., and Liu, X. (2018). Classification of topological relations between spatial objects in two-dimensional space within the dimensionally extended 9-intersection model. *Transactions in GIS*, 22(2):514–541.

Sherman, G. (2008). *Desktop GIS: Mapping the Planet with Open Source Tools*. Pragmatic Bookshelf.

Simoes, R., Camara, G., Queiroz, G., Souza, F., Andrade, P. R., Santos, L., Carvalho, A., and Ferreira, K. (2021a). Satellite image time series analysis for big earth observation data. *Remote Sensing*, 13(13).

Simoes, R., Souza, F., Zaglia, M., Queiroz, G. R., Santos, R., and Ferreira, K. (2021b). Rstac: An R package to access spatiotemporal asset catalog satellite imagery. In *2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS*, pages 7674–7677.

Sørensen, R., Zinko, U., and Seibert, J. (2006). On the calculation of the topographic wetness index: Evaluation of different methods based on field observations. *Hydrology and Earth System Sciences*, page 13.

Spanier, E. H. (1995). *Algebraic Topology*. Springer, 1st edition.

Talbert, R. J. A. (2014). *Ancient Perspectives: Maps and Their Place in Mesopotamia, Egypt, Greece, and Rome*. University of Chicago Press.

Tallon, A. R. (2007). Bristol. *Cities*, 24(1):74–88.

Tennekes, M. (2018). tmap: Thematic maps in R. *Journal of Statistical Software*, 84(6):1–39.

Tennekes, M. and Nowosad, J. (2024). *Elegant and Informative Maps with {tmap}*. (in progress).

Thiele, J. (2014). R Marries NetLogo: Introduction to the RNetLogo Package. *Journal of Statistical Software*, 58(2):1–41.

Tobler, W. R. (1970). A computer movie simulating urban growth in the Detroit region. *Economic geography*, pages 234–240.

Tobler, W. R. (1979). Smooth Pycnophylactic Interpolation for Geographical Regions. *Journal of the American Statistical Association*, 74(367):519–530.

tom Dieck, T. (2008). *Algebraic Topology*. EMS Textbooks in Mathematics. European Mathematical Society, Zürich.

Tomintz, M. N. M., Clarke, G. P., and Rigby, J. E. J. (2008). The geography of smoking in Leeds: Estimating individual smoking rates and the implications for the location of stop smoking services. *Area*, 40(3):341–353.

Tomlin, C. D. (1990). *Geographic Information Systems and Cartographic Modeling*. Prentice Hall, Englewood Cliffs, N.J.

Tomlin, C. D. (1994). Map algebra: One perspective. *Landscape and Urban Planning*, 30(1-2):3–12.

van der Meer, L., Abad, L., Gilardi, A., and Lovelace, R. (2023). *sfnetworks: Tidy Geospatial Networks*. R package version 0.6.3.

Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer, New York, fourth edition.

Visvalingam, M. and Whyatt, J. D. (1993). Line generalisation by repeated elimination of points. *The Cartographic Journal*, 30(1):46–51.

von Wehrden, H., Hanspach, J., Bruelheide, H., and Wesche, K. (2009). Pluralism and diversity: Trends in the use and application of ordination methods 1990-2007. *Journal of Vegetation Science*, 20(4):695–705.

Walker, K. (2022a). *crsuggest: Obtain Suggested Coordinate Reference System Information for Spatial Data*. R package version 0.4.

Walker, K. E. (2022b). *Analyzing US Census Data: Methods, Maps, and Models in R*. Chapman & Hall/CRC.

Wardrop, J. G. (1952). Some theoretical aspects of road traffic research. *Proceedings of the Institution of Civil Engineers*, 1(3):325–362.

Wegmann, M., Leutner, B., and Dech, S., editors (2016). *Remote Sensing and GIS for Ecologists: Using Open Source Software*. Data in the Wild. Pelagic Publishing, Exeter.

Wickham, H. (2014). Tidy Data. *Journal of Statistical Software*, 59(10).

Wickham, H. (2016). *Ggplot2: Elegant Graphics for Data Analysis*. Springer, New York, NY, second edition.

Wickham, H. (2019). *Advanced R, Second Edition*. CRC Press.

Wickham, H. (2021). *Mastering Shiny: Build Interactive Apps, Reports, and Dashboards Powered by R*. O'Reilly Media, Sebastopol, CA.

Wickham, H., François, R., Henry, L., Müller, K., and Vaughan, D. (2023). *dplyr: A Grammar of Data Manipulation*. R package version 1.1.3.

Wieland, T. (2017). Market Area Analysis for Retail and Service Locations with MCI. *The R Journal*, 9(1):298–323.

Wilkinson, L. and Wills, G. (2005). *The Grammar of Graphics*. Springer Science+ Business Media.

Wimberly, M. C. (2023). *Geographic Data Science with R: Visualizing and Analyzing Environmental Change*. Chapman & Hall/CRC.

Wise, S. (2001). *GIS Basics*. CRC Press.

Wright, M. N. and Ziegler, A. (2017). Ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R. *Journal of Statistical Software*, 77(1):1–17.

Wulf, A. (2015). *The Invention of Nature: Alexander von Humboldt's New World*. Alfred A. Knopf, New York.

Xiao, N. (2016). *GIS Algorithms: Theory and Applications for Geographic Information Science & Technology*. SAGE Publications, London.

Xie, F. and Levinson, D. (2011). *Evolving Transportation Networks*. Transportation Research, Economics and Policy. Springer-Verlag, New York.

Zuur, A., Ieno, E. N., Walker, N., Saveliev, A. A., and Smith, G. M. (2009). *Mixed Effects Models and Extensions in Ecology with R*. Statistics for Biology and Health. Springer-Verlag, New York.

Zuur, A. F., Ieno, E. N., Saveliev, A. A., and Zuur, A. F. (2017). *Beginner's Guide to Spatial, Temporal and Spatial-Temporal Ecological Data Analysis with R-INLA*, volume 1. Highland Statistics Ltd, Newburgh, United Kingdom.

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

Index

agent-based modeling, 310, 311
aggregation, 60, 342
  spatial, 89, 117
algorithm, 1, 255, 273, 274, 277, 278, 280, 281, 283, 286
API, 8, 183, 189, 221, 249, 325, 343
ArcGIS, 8, 172
areal units, 310
attribute, 53, 54
  aggregation, 60, 61
  create, 65, 66
  dropping geometries, 55
  join, 62, 64, 314
  subsetting, 56, 57, 61
AUROC, 291, 292, 297, 301, 303, 361
autocorrelation
  spatial, 251, 287, 290, 291, 297, 303, 304, 309
  temporal, 304

base plot, *see* map-making
big data, 286, 372
binary predicate, *see* topological relations
bounding box, 28, 254, 345

C, 6, 8
C++, 6, 8, 224, 239, 259, 277, 377
cartogram (package), 232
catchment area, 267, 311, 347, 349, 352, 354–356
centroid, 270, 274–278, 310, 318, 320, 343
CGAL, 277, 278
classification, 293, 295, 298, 361
cloud computing, 263, 265
clustering, 286, 293
  kmeans, 292
COG, 174, 179, 182, 263–265
color palettes, 206
command line interface, 249, 379
convex hull, 283
coordinate reference system, *see* CRS
CRAN, 9, 11, 18
cross-validation, 286, 287, 291, 292, 297, 303, 304
  spatial CV, 286, 287, 289, 291, 296, 299, 303, 352, 361, 362, 367
CRS, 11, 19, 346
  custom, 160, 167
  EPSG, 148, 150
  geographic, 47, 147, 159, 242, 340
  introduction, 47
  proj-string, 148, 150, 168
  projected, 48, 147, 159, 340
  reprojection, 158, 161
  SRID, 149, 150
  WKT, 148, 150

data cube, 263–266
data models, 18
data packages, 184
data science, 8, 378
desire lines, 310, 317, 321, 322, 327
digital elevation model, 247, 267, 354
distance relations, 81
dplyr (package), 117, 138

edge, 320, 330, 331

file formats, 172
FORTRAN, 6, 239
FOSS4G, 1

function, 7, 278–280, 355

GDAL, 11, 21, 130, 172, 175, 177, 179, 258, 259, 264, 377
gdalcubes (package), 264
generalized additive model, 303, 366
geoalgorithm, 8, 254, 257, 274, 355
geocoding, 186, 343
geocomputation, 1–4, 8, 73, 171, 196, 239, 372, 377–379
  definition, 3
geocompx, 375
GeoDjango, 224
geofacet (package), 232
geographic data analysis, 2, 285
geographic metadata, 187
geographic web services, 188
  CSW, 188
  WCPS, 188
  WCS, 188
  WFS, 188
  WMS, 188
  WMTS, 188
  WPS, 188
geographical data science, *see* satial data science3
geography, 4, 196
geogrid (package), 232
geomarketing, 337, 338, 349, 357
geomorphons, 248
GeoPackage, 172, 174, 176
geoportals, 183
GEOS, 21, 110, 153, 157, 259, 277, 278, 377
GeoTIFF, 174
ggplot2 (package), 12, 13, 229
GIS, 1, 4, 6, 8, 11, 14, 100, 237, 257, 347, 366
  connotations, 4
  definition, 3
GLM, 285, 286, 289, 293, 299, 302, 303
GML, 172
googleway (package), 222
GPS, 5
graph, 320, 330
graphical user interface, 1, 6, 172, 237, 238
GRASS GIS, 8, 237, 239, 240, 245, 252–254, 257, 260, 263

hillshade, 144, 267, 287, 354
hyperparameter, 292, 296, 298–304, 352, 361–364, 366, 367

IDE, 6, 282
igraph (package), 330
intersects, 80

Java, 6, 8, 225
join, 62
  inner, 315
  left, 315
  non-overlapping, 87
  spatial, 85

KML, 172, 178

leaflet (package), 219, 224
lidR (package), 13
linemap (package), 232
list column, 23, 321
location analysis, 337
loop
  for, 341
  lapply, 276, 279, 345
  map, 345
  vapply, 276, 279
  while, 345

machine learning, 285, 286, 298, 304, 352, 366
map algebra, 94, 338
  focal operations, 98
  local operations, 95, 340
  zonal operations, 100
map-making, 196
  aesthetics, 200
  animated maps, 216
  base plotting, 28
  basic, 27

basic raster, 45
color palettes, 206
faceted maps, 210
inset maps, 212
interactive, 6
interactive maps, 6, 219
layers, 198
mapping applications, 224
metaplot, 200
outputs, 191
static maps, 197
visual variables, 200
mapdeck (package), 219, 222
MapServer, 224
mapsf (package), 231
mapview (package), 219, 220
meridians, 155
mlr3 (package), 289, 292, 293, 296, 298, 303, 360, 362, 366

National Grid, 20
NDVI, 352
nearest neighbor, 321
network, 255, 310, 320, 327–330
NMDS, 354, 357–360, 366, 367
node, 310, 320, 330

OGR SQL, 176
open data, 183, 338
open source software, 338
OpenEO, 263, 265, 266
OpenStreetMap, 185, 254, 313, 330, 335, 338, 344, 345
ordination, 352, 357, 358, 366
Orfeo Toolbox, 258
osmdata (package), 254, 330, 338, 344
osmextract (package), 330
overfitting, 286, 287, 292, 360, 361

parallelization, 301, 302, 304, 367
PCA, 357
pipe operator, 59, 61
plotly (package), 230
point of interest, 344, 348
point pattern analysis, 372
PostGIS, 26, 260, 261, 263, 267
PROJ, 11, 13, 259
projection
Azimuthal equidistant, 160
Lambert azimuthal equal-area, 160
Lambert conformal conic, 160
Stereographic, 160
Universal Transverse Mercator, 160
World Geodetic System, 158
pseudocode, 274
Python, 1, 6, 8, 9, 224, 377

QGIS, 1, 8, 14, 26, 172, 196, 237, 239, 240, 244, 257, 260, 267
qgisprocess (package), 239, 240, 257

R, 3, 4, 6, 8, 9, 277, 286, 289, 335, 349, 356, 369, 377
base, 370
history, 10
installation, 17
language, 6
prerequisites, 17
R-spatial, 9
history, 10
random forest, 298, 304, 352, 360–362, 366, 367
raster, 41, 247, 257, 267, 336, 338, 346, 347, 356, 367, 371
aggregation, 125
categorical, 67
class, 45
cropping, 133
CRS, 152
data input, 179
data output, 181
data types, 181
disaggregation, 125, 126
extraction, 135
extraction fractions, 139
extraction lines, 135
extraction points, 135
extraction polygons, 137

header, 41
intersection, 123
manipulation, 67, 122
masking, 134
merge, 101
merging, 123
origin, 124
rasterize, 346
reprojection, 163
resampling, 127, 128, 130, 163
subsetting, 68, 69, 92
summarizing, 70
transformation, 163
values, 70
warping, 163
raster (package), 12, 13, 42
raster attribute table, 67
raster data model, 41
raster-vector interactions, 133
rasterization, 140
lines, 141
points, 140
polygons, 141
rayshader (package), 13, 232
regression, 292, 293, 295, 361, 362
linear, 303, 366
remote sensing, 5, 357
REPL, 8
reproducibility, 1–3, 270, 272, 369, 375, 378
resampling, 292, 303
rgrass (package), 239, 253, 257
RMSE, 361, 363, 367
routes, 310, 322
routing, 325, 326, 349
RSAGA (package), 12
Rsagacmd (package), 239, 249, 257
rstac (package), 264
RStudio, 6, 17, 271, 272, 282
Rust, 377

S, 11
S2, 21, 22, 39, 111, 153, 155, 157
SAGA, 8, 237, 239, 240, 245, 247, 249, 252, 257, 355, 356
Scala, 377
segmentation, 250
seeded region growing algorithm, 250
superpixels, 252
Sentinel-2, 263
sf, 9, 21, 254, 261, 269, 346, 371, 379
class, 32
geometry collection, 31
geometry types, 30
hole, 30
linestring, 30
multi features, 31
point, 30
sfc, 25
simple feature columns (sfc), 35
st_transform, 346
units, 50
why simple features, 26
sf (package), *see* sf
sfheaders, 37
sfnetworks (package), 324, 330
Shapefile, 11, 172, 176
shiny (package), 225
shortest route, 253, 255, 331, 333
simple features, *see* sf
sits (package), 265
sliver polygons, 245
smoothr (package), 108
source code, 278
sp (package), 22, 370
spatial
join, 85
statistics, 8, 9, 12
subsetting, 75, 92, 113, 115
spatial congruence, 90, 242
spatial data science, 3
spatial database, 5, 239, 253, 254, 256, 257, 259, 260, 263
spatial interpolation, 372
spatial operations, 73
spatial regression, 372
spatial vectorization, 142
contours, 143
points, 143

polygons, 144
spDataLarge (package), 287
STAC, 263–265
stars (package), 13, 42, 164
statistical inference, 372
statistical learning, 286, 289, 292, 295, 298, 303
statistics, 6, 8, 11, 70, 89, 100, 101, 286
stplanr (package), 318, 321, 326
SVM, 286, 298, 303, 304, 361, 362

tanaka (package), 232
TauDEM, 258
terra (package), 13, 42–45, 343
tibble, 55
tidyverse, 26
tidyverse (package), 9, 55, 56, 75, 92, 229, 370
tmap (package), 197, 316, 319
  animated maps, 216
  basics, 197
  break styles, 204
  categorical scale, 205
  color breaks, 203
  continuous scale, 205
  faceted maps, 210
  graticules, 209
  inset maps, 212
  interactive maps, 219
  interval scale, 204
  layouts, 209
  legends, 208
  north arrows, 209
  palettes, 205
  saving maps, 191
  scale bars, 209
  scales, 203
topographic wetness index, 247
topological relations, 76–78, 259
  DE-9IM, 82
topology cleaning, 245, 255
traveling salesman, 253, 255

union, 242, 243
units, 50
UTM, 159

vector
  affine transformation, 111
  buffers, 109, 261
  centroids, 109
  clipping, 113, 115
  CRS, 151, 152
  data input, 175
  data output, 180
  distance relations, 81
  geometry, 23
  geometry casting, 118
  intersection, 113, 116, 131, 244
  multilinestrings, 321
  reprojection, 161
  simplification, 106, 107
  subsetting, 75, 256
  union, 105, 117, 244, 261, 370
vector data model, 19
vegan (package), 357
viewshed, 267
von Humboldt, 4
VS Code, 17
vsicurl, 179

well-known binary, 30, 178
well-known text, 30, 178
WhiteboxTools, 377
WKB, *see* well-known binary
WKT, *see* well-known text