


Data Management for Social Scientists

The “data revolution” offers many new opportunities for research in the social sciences. Increasingly, traces of social and political interactions can be recorded digitally, leading to vast amounts of new data that become available for research. This poses new challenges for the way we organize and process research data. This book covers the entire range of data management techniques, from flat files to database management systems. It demonstrates how established techniques and technologies from computer science can be applied in social science projects, drawing on a wide range of different applied examples. The book covers simple tools such as spreadsheets as well as file-based data storage and processing, and then moves on to more powerful data management software such as relational databases. In the final part of the book, it deals with advanced topics such as spatial data, text as data, and network data.

Nils B. Weidmann is Professor of Political Science at the University of Konstanz. Trained both as a computer scientist and a political scientist, he completed his PhD at ETH Zurich in 2009. His work has been recognized with several awards, including the 2020 Karl Deutsch Award of the International Studies Association.



Methodological Tools in the Social Sciences

SERIES EDITORS

Paul M. Kellstedt, *Associate Professor of Political Science, Texas A&M University*

Guy D. Whitten, *Professor of Political Science and Director of the European Union Center at Texas A&M University*

The Methodological Tools in the Social Sciences series is comprised of accessible, stand-alone treatments of methodological topics encountered by social science researchers. The focus is on practical instruction for applying methods, for getting the methods right. The authors are leading researchers able to provide extensive examples of applications of the methods covered in each book. The books in the series strike a balance between the theory underlying and the implementation of the methods. They are accessible and discursive, and make technical code and data available to aid in replication and extension of the results, as well as enabling scholars to apply these methods to their own substantive problems. They also provide accessible advice on how to present results obtained from using the relevant methods.

Other books in the series

Eric Neumayer and Thomas Plümper,
Robustness Tests for Quantitative Research

Data Management for Social Scientists

From Files to Databases

NILS B. WEIDMANN

University of Konstanz, Germany



CAMBRIDGE
UNIVERSITY PRESS



Shaftesbury Road, Cambridge CB2 8EA, United Kingdom
One Liberty Plaza, 20th Floor, New York, NY 10006, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
314-321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre,
New Delhi – 110025, India
103 Penang Road, #05-06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment,
a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of
education, learning and research at the highest international levels of excellence.

www.cambridge.org

Information on this title: www.cambridge.org/9781108845670

DOI: [10.1017/9781108990424](https://doi.org/10.1017/9781108990424)

© Nils B. Weidmann 2023

This work is in copyright. It is subject to statutory exceptions and to the provisions of
relevant licensing agreements; with the exception of the Creative Commons version the
link for which is provided below, no reproduction of any part of this work may take place
without the written permission of Cambridge University Press & Assessment.

An online version of this work is published at doi.org/10.1017/9781108990424 under
a Creative Commons Open Access license CC-BY-NC 4.0.

All versions of this work may contain content reproduced under license from third parties.

Permission to reproduce this third-party content must be obtained from these third
parties directly. When citing this work, please include a reference to the
DOI [10.1017/9781108990424](https://doi.org/10.1017/9781108990424).

First published 2023

A catalogue record for this publication is available from the British Library.

ISBN 978-1-108-84567-0 Hardback

ISBN 978-1-108-96478-4 Paperback

Cambridge University Press & Assessment has no responsibility for the persistence
or accuracy of URLs for external or third-party internet websites referred to in this
publication and does not guarantee that any content on such websites is, or will
remain, accurate or appropriate.

To my family.
To Michael D. Ward (1948–2021).

Contents

Preface *page xi*

PART I INTRODUCTION

1	Motivation	3
1.1	Data Processing and the Research Cycle	4
1.2	What We Do (and Don't Do) in this Book	5
1.3	Why Focus on Data Processing?	7
1.4	Data in Files vs. Data in Databases	9
1.5	Target Audience, Requirements and Software	11
1.6	Plan of the Book	12
2	Gearing Up	14
2.1	R and RStudio	14
2.2	Setting Up the Project Environment for Your Work	16
2.3	The PostgreSQL Database System	20
2.4	Summary and Outlook	22
3	Data = Content + Structure	23
3.1	What Is Data?	23
3.2	Data Content and Structure	24
3.3	Tables, Tables, Tables	26
3.4	The Structure of Tables Matters	30
3.5	Summary and Outlook	35

PART II DATA IN FILES

4	Storing Data in Files	39
4.1	Text and Binary Files	40
4.2	File Formats for Tabular Data	43

4.3	Transparent and Efficient Use of Files	54
4.4	Summary and Outlook	57
5	Managing Data in Spreadsheets	59
5.1	Application: Spatial Inequality	60
5.2	Spreadsheet Tables and (the Lack of) Structure	63
5.3	Retrieving Data from a Table	64
5.4	Changing Table Structure and Content	66
5.5	Aggregating Data from a Table	67
5.6	Exporting Spreadsheet Data	70
5.7	Results: Spatial Inequality	70
5.8	Summary and Outlook	71
6	Basic Data Management in R	74
6.1	Application: Inequality and Economic Performance in the US	75
6.2	Loading the Data	76
6.3	Merging Tables	79
6.4	Aggregating Data from a Table	82
6.5	Results: Inequality and Economic Performance in the US	84
6.6	Summary and Outlook	85
7	R and the <code>tidyverse</code>	87
7.1	Application: Global Patterns of Inequality across Regime Types	88
7.2	A New Operator: The Pipe	89
7.3	Loading the Data	90
7.4	Merging the WID and Polity IV Datasets	92
7.5	Grouping and Aggregation	93
7.6	Results: Global Patterns of Inequality across Regime Types	96
7.7	Other Useful Functions in the <code>tidyverse</code>	97
7.8	Summary and Outlook	99

PART III DATA IN DATABASES

8	Introduction to Relational Databases	103
8.1	Database Servers and Clients	105
8.2	SQL Basics	108
8.3	Application: Electoral Disproportionality by Country	109
8.4	Creating a Table with National Elections	110
8.5	Computing Electoral Disproportionality	115
8.6	Results: Electoral Disproportionality by Country	117
8.7	Summary and Outlook	118
9	Relational Databases and Multiple Tables	121
9.1	Application: The Rise of Populism in Europe	122
9.2	Adding the Tables	123
9.3	Joining the Tables	125

9.4	Merging Data from the PopuList	127
9.5	Maintaining Referential Integrity	129
9.6	Results: The Rise of Populism in Europe	131
9.7	Summary and Outlook	132
10	Database Fine-Tuning	135
10.1	Speeding Up Data Access with Indexes	136
10.2	Collaborative Data Management with Multiple Users	140
10.3	Summary and Outlook	143

PART IV SPECIAL TYPES OF DATA

11	Spatial Data	147
11.1	What Is Spatial Data?	147
11.2	Application: Patterns of Violence in the Bosnian Civil War	150
11.3	Reading and Visualizing Spatial Data in R	151
11.4	Spatial Data in a Relational Database	158
11.5	Results: Patterns of Violence in the Bosnian Civil War	163
11.6	Summary and Outlook	164
12	Text Data	166
12.1	What Is Textual Data?	167
12.2	Application: References to (In)equality in UN Speeches	169
12.3	Working with Strings in (Base) R	170
12.4	Natural Language Processing with <code>quanteda</code>	175
12.5	Using PostgreSQL to Manage Documents	179
12.6	Results: References to (In)equality in UN Speeches	183
12.7	Summary and Outlook	184
13	Network Data	187
13.1	What Is Network Data?	187
13.2	Application: Trade and Democracy	190
13.3	Exploring Network Data in R with <code>igraph</code>	191
13.4	Network Data in a Relational Database	197
13.5	Results: Trade and Democracy	204
13.6	Summary and Outlook	205

PART V CONCLUSION

14	Best Practices in Data Management	209
14.1	Two General Recommendations	209
14.2	Collaborative Data Management	212
14.3	Disseminating Research Data and Code	214
14.4	Summary and Outlook	216

	<i>Bibliography</i>	219
--	---------------------	-----

	<i>Index</i>	223
--	--------------	-----

Preface

More than a decade ago, while I was still a PhD student, Mike Ward encouraged me to develop a book project about relational databases for social science applications. The book proposal was not successful, but I never completely abandoned the idea. Later in my career, when working with many excellent students, I realized that there is still a huge need to establish data management as part of our quantitative social science curricula. Most of the training we offer in political science focuses on (oftentimes advanced) methods for statistical analysis and causal inference, but does not really help students get to the datasets required for this. As a result, “many social scientists will find themselves ‘hacking together’ datasets in a fundamentally ad hoc way,” as one reviewer for this book commented on the status quo in our field. I hope that this book contributes to improving this.

In comparison to the original idea, the focus of this book has been expanded considerably, beyond relational databases. The first half of the book describes different tools to manage data in a file-based workflow, without interfacing with a dedicated database system. Yet, more technically advanced readers will wonder why I focus so much on databases in the second half of the book, given that this is – at least by computer science standards – a fairly old technology. Still, relational databases continue to be around, and they allow me to cover a number of key learnings that easily generalize beyond this technology. First, with the need to explicitly define data structures (tables) before we can use them, databases force us to think about data structure much more than we commonly do in social science data analysis. What information should the individual tables contain, how many do we need, and how are they linked? There are different

ways to do this, and some are better than others. Even if readers later move on to less-structured data – which is becoming more and more common also in the social sciences –, they will do so being fully aware of the strengths and weaknesses of the different approaches. Relational databases also allow me to cover some basic techniques for managing large amounts of data, which are essential as our datasets become bigger. Indexing a table is a standard operation in a database, and we can nicely illustrate what we gain from it. Last, databases are a great way to demonstrate how a client-server setup works. As our data management becomes more complex, for example due to the amount of data we need to process, there is an increasing need to perform certain tasks on specialized servers rather than one's own personal computer. This makes it necessary to interact with these servers, which is something we do in this book using a relational database management system.

This book benefited from the help and support by several people and institutions. The initial development of the material was funded by the German Federal Ministry of Education and Research under the “*b*³ – beraten, begleiten, beteiligen” project, and Lukas Kawerau, with his extensive skills as a computational social scientist, was essential in getting a first draft off the ground. I am grateful to Lars-Erik Cederman and the International Conflict Research group at ETH Zurich for hosting me during the Winter term 2019–2020, which gave me the opportunity to work intensively on this project. During this time, Luc Girardin with his joint computer science and social science background provided many useful comments and suggestions. At Konstanz, the members of my group (Frederik Gremler, Anna-Lena Hönig, Eda Keremoğlu, Sebastian Nagel, Stefan Scholz and Patrick Zwerschke) and the students in my courses on data management (summer term 2020 and 2021) were critical and constructive readers, and contributed greatly to the improvement of this book. Also, an early presentation of this project at our department's Center for Data and Methods (CDM) proved to be extremely helpful in setting the general scope, clarifying the main goals of the project. I am very grateful to Guy Whitten and Paul Kellstedt for including this book in the Methodological Tools in the Social Sciences series at Cambridge University Press, and for supporting me with their expertise and advice. The open access publication of this book was made possible through financial support from the University of Konstanz's Open Access Fund and the the Cluster of Excellence “The Politics of Inequality”

(EXC-2035/1-390681379). Lastly, I want to thank all the developers of the datasets and the free software used in this book. We often fail to realize that behind every database we use and every package we install, there is a person or a team investing so much time and effort for the benefit of the entire research community.

PART I

INTRODUCTION

Motivation

The way in which we conduct empirical social science has changed tremendously in the last decades. Lewis Fry Richardson, for example, was one of the first researchers to study wars with scientific methods in the first half of the twentieth century. Among many other projects, he put together a dataset on violent conflicts between 1815 and 1945, which he used in his *Statistics of Deadly Quarrels* (Richardson, 1960). Richardson collected this information on paper, calculating all of the statistics used for his book manually. Today, fortunately, empirical social science leverages the power of modern digital technology for research, and data collection and analysis are typically done using computers.

Most of us are perfectly familiar with the benefits of digital technology for empirical social science research. Many social science curricula – for example, in political science, economics, or sociology – include courses on quantitative methods. Most of the readers of this book are trained to use software packages such as SPSS, Stata, or R for statistical analysis, which relieve us of most of the cumbersome mathematical operations required for this. However, according to my experience, there is little emphasis on how to prepare data for analysis. Many analyses require that data from different sources and in potentially different formats be imported, checked, and combined. In the age of “Big Data,” this has become even more difficult due to the larger, and more complex, datasets we typically work with in the social sciences.

I wrote this book to close this gap in social science training, and to prepare my readers better for new challenges arising in empirical work in the social sciences. It is a course in data processing and data management, going through a series of tools and software packages that can

assist researchers getting their empirical data ready for analysis. Before we discuss what this book does and who should read it, let us start with a short description of the research cycle and where this book fits in.

I.1 DATA PROCESSING AND THE RESEARCH CYCLE

Most scientific fields aim to better understand the phenomena they study through the documentation, analysis, and explanation of empirical patterns. This is no different for the *social* sciences, which are the focus of this book. I fully acknowledge that there is considerable variation in the extent to which social scientists rely on empirical evidence – I certainly do not argue that they necessarily should. However, this book is written for those that routinely use empirical data in their work, and are looking for ways to improve the processing of these data.

How does the typical research workflow operate, and where does the processing of data fit in? We can distinguish three stages of an empirical research project in the social sciences:

1. Data collection
2. Data processing
3. Data analysis

The first stage, data collection, is the collection or acquisition of the data necessary to conduct an empirical analysis. In its simplest form, researchers can rely on data collected and published by someone else. For example, if you conduct a cross-national analysis of economic outcomes, you can obtain data from the comprehensive *World Development Indicators* database maintained by the World Bank (2021). Here, acquisition for the end users of this data is easy and just takes a few mouse clicks. Similarly, excellent survey data can be obtained from large survey projects such as the *Demographic and Health Surveys* (US Agency for International Development, 2021) or the *Afrobarometer* (2021). In other cases, data gathering for a research project is more difficult. Researchers oftentimes collect data themselves, for example by coding information from news reports or other sources, or by conducting surveys. In these cases, data collection is a fundamental part of the contribution a research project aims to make, and requires considerable resources.

The output of the first stage is typically a (set of) *raw* dataset(s). Before the raw data can be used in the analysis, it needs to be processed in different ways. This data processing can include different operations. For example, we may have to adjust text-based codings in our data, since our

statistical package can only deal with numbers. In many cases, we need to aggregate information in our dataset; for example, if our original raw data contains survey results at the level of households, but we conduct our analysis at the level of villages, we have to compute the sum or the average over all households in a village. In other cases, we have to combine our original dataset with others. For instance, if we study the relationship between the level of economic development and the level of democracy, we may have to combine information from the *World Development Indicators* database with data on regime type, for example from the *Varieties of Democracy* project (Coppedge et al., 2019).

The third stage in our simple research workflow is data analysis. “Analysis” refers to any kind of pattern we derive from the data prepared in the previous stage, such as a correlation table, a graphical visualization of the distribution of a particular variable, or the coefficients from a regression model estimated on the data. Data analysis – whether it is descriptive, graphical, or statistical – requires that our data be provided in a particular format, a format that is not necessarily the most convenient one for data collection or data storage. For example, if we analyze the relationship between development and regime type as mentioned earlier, it is necessary to combine data from different sources into a single dataset that is ultimately used for the analysis. Hence, separating data processing from data analysis – as we do in this book – is not simply a convenient choice in the research workflow, but rather a necessity.

1.2 WHAT WE DO (AND DON'T DO) IN THIS BOOK

The focus on the three stages of an empirical analysis suggests the following workflow, depicted in Figure 1.1. The first stage, data collection, produces one or more raw datasets that are input to the second stage, data management. At this stage, the data are processed in various ways: they are cleaned, recoded, and combined in order to yield one or more datasets for analysis, which are used during the data analysis stage. The gray box shows what is covered in this book: how to get from the raw dataset(s) to those used for analysis.

This depiction of the research workflow does not mean that these three stages are always carried out in strict sequence. Rather, in reality researchers will likely go back and forth between them. This is necessary when adding an additional variable to the analysis, for example, as a new control variable: In this case, we have to adjust the data processing step, such that the variable is part of the analysis dataset. In some instances,

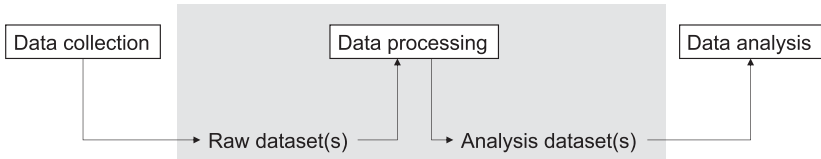


FIGURE 1.1. Research workflow in an empirical social science project. The gray area is what we cover in this book.

this may even require us to go back to the first stage (data collection), since we may not yet have the necessary information in our raw data. Similarly, we may have to go back to the data processing stage for much simpler operations, for example, when the format of a variable in our analysis cannot be processed by our statistical software or our visualization toolkit.

While useful for the purpose of illustration, the three stages are clearly a simplification of the typical research workflow and omit a number of additional steps. To name just one example, the “Data Lifecycle” by the US Geological Survey (Faundeen et al., 2014) is a more complete illustration of the different phases involved in an empirical research project. It includes six phases: (1) Plan, (2) Acquire, (3) Process, (4) Analyze, (5) Preserve, and (6) Publish/Share. The planning of a project in Stage 1 is obviously of key importance, and a social science research proposal normally includes all the necessary details for how the empirical analysis should be carried out. At this stage, one would also pre-register a study, for example, within the *Center for Open Science’s Open Science Framework*. Stage 2 in their model corresponds to what we call “data collection,” and Stages 3 and 4 correspond to our second and third stages. Stage 5 covers the documentation and physical storage of the data, such that it can later be accessed and used again. This stage is closely related to Stage 6, since documentation, anonymization, and technical description are required for both.

This book focuses on the practical aspects of data processing, and thus covers primarily the second step in our three-stage model. Research design and, in particular, data analysis are part of most social science programs that focus on empirical work, while basic questions of data management and processing are typically left out. This means that most researchers will be perfectly able to design their study and carry out an empirical analysis, while struggling with the processing of data. In this book, we will see that some of the standard practices that have evolved in the community can lead to inefficient workflows that make life difficult

for researchers, or even introduce errors in the data. For example, in contrast to conventional wisdom, spreadsheets are in most cases not a good choice for managing your data, even if they appear to be simple and intuitive. Also, managing and analyzing data in the same software can be challenging or even impossible, since data processing oftentimes requires specialized functionality that data analysis software simply lacks. Hence, I recommend to treat data processing as a step that is different from – but of course closely connected to – data collection (which comes before) and data analysis (which is what we do after the processing of our data).

The focus on hands-on data processing also distinguishes this book from what is typically referred to as “data management” without the focus on practical questions. This alternative definition of “data management” includes strategic questions of organizations about how to acquire, store, document, and disseminate research data (Henderson, 2017). These are typically issues that are addressed by dedicated organizational units, for example, university libraries. Of course, data management also needs to solve practical questions about processing and storage such as the ones we discuss here, and therefore overlaps partly with the content of this book. In other words, data management for large organizations requires much of the knowledge and skills I try to convey in this book, but also entails a number of other challenges we do not cover. Rather, the book caters to the needs of individual researchers or small research groups, who are oftentimes responsible for designing most of their data processing procedures themselves – something that, according to my own experience, is probably what most researchers in the social sciences do.

1.3 WHY FOCUS ON DATA PROCESSING?

Readers may wonder why we need an entire book on data processing. There are several reasons why researchers should devote more attention to working with data. In particular, I believe that there are several major advantages to treating data processing as a separate step in the research workflow, which requires particular skills and (potentially) specialized software.

DOCUMENTATION. One of the most important goals of this book is to show you how to properly document the processing of your data. That is, every operation you apply to the raw data you start from until you end up with a dataset for analysis must be written down, such that you – or someone else – can later return to it. For many of us, this type of documentation is standard practice for data analysis – that is, we produce

code in R, Stata, or another statistical package that executes the different steps required to produce a plot or to run a statistical model. Manually merging or aggregating data in a spreadsheet, for example, is very different; here, you essentially point-and-click to achieve the desired result, and these operations are difficult, if not impossible, to understand and repeat later. In contrast, (almost) all the different methods I present in this book are automated; they allow you to prepare and process your data using a set of instructions to a data management software. As a result, your research workflow improves in several ways.

CONVENIENCE. One of the advantages of fully documented data processing is simply the added convenience for you as the researcher. Automated data processing is more powerful and much faster, since you can let the computer process many entries in your dataset at once, rather than manually fixing them. Also, you can later modify your processing instructions in case you change your mind, or if you discover mistakes. By adjusting the data processing code, it is possible to change the coding of individual variables, introduce different types of aggregation, or derive datasets for different types of analysis from your raw data. All of this is extremely cumbersome if you resort to manual data management, for example, by using spreadsheet software.

REPLICABILITY AND TRANSPARENCY. Another major advantage of a fully documented data workflow from the raw data to the dataset used for analysis is the transparency resulting from this process. This documentation is not just an advantage to you as a researcher, but it allows you to share your data processing code in the research community, thus making your work perfectly replicable by others. The replication of empirical research has been at the forefront of current attempts towards increased research transparency (see, e.g. the DA-RT Initiative, 2015). Almost all major journals in the social sciences now require that the data and code used for the analysis be published along with an article. While this is a move in the right direction, increased transparency should also apply to the data processing stage. Thus, with the techniques presented in this book, it is possible to create fully documented data workflow, which can make data preparation transparent and replicable.

SCALABILITY. One of the benefits of the digital transformation is the increasing amount of data that becomes available to social scientists. Rather than analyzing a few dozen observations, as Richardson (1960) did in his empirical analysis of wars, researchers now possess datasets that are several orders of magnitude larger. For example, recent work

with data collected from social media can easily include millions of observations. This requires data processing techniques that can deal with large datasets, in other words, that scale with increasing amounts of data. This is where conventional tools quickly come to their limits. Spreadsheets are not suitable for these amounts of data, not just because manual editing of data is no longer possible, but also because they have an upper limit on the number of entries in a dataset they can process (for MS Excel, for example, this limit is about 1 million). Also, many statistical tools are not suitable, since they too have difficulties processing large datasets (although there are extensions that make this possible). In contrast, some of the more advanced tools I present in this book are perfectly scalable; they are designed to store and process large datasets while hiding most of the complexity of these operations from the user. Again, it is sometimes useful to use specialized, but different, software tools for data processing and data analysis, since each of them have different strengths and weaknesses.

VERSATILITY. Along with the increase in the amount of data that social scientists use for their work, we also witness an increase in the complexity of the data formats used. Rather than relying exclusively on single tables of data where observations are nicely arranged in rows and columns, there is now a variety of different types of data, each stored in a specific data format. For example, many different social science projects now use observations with geographic coordinates, where each observation in a dataset is tagged with a reference to a particular location on the globe: The *Demographic and Health Surveys*, for example, distribute geographic coordinates for their more recent survey waves, which makes it possible to locate each group of households that participated in the survey on a map. One potential use of these geo-coordinates is to combine the survey results with other information based on their location, for example, with night light emissions. Spatial data is just one example for new types of information requiring adjustments to the standard tabular data model; in this book, we present others, such that researchers can make an informed choice about their specific requirements and the software tools they should use for their work.

I.4 DATA IN FILES VS. DATA IN DATABASES

Most of the data we use in the social science comes in electronic files. That is, in a quantitative research project we rely on files to store the data we

have collected, and we use files to archive the data and to pass them on to other users. There are many different types of files for data: You are probably familiar with Microsoft's Excel files for storing spreadsheets, or Stata and SPSS files for tabular datasets. However, while files will continue to be the primary way by which we distribute data, they can be tricky to work with. For example, if the data is spread out across different files, you need to merge them before you can run your analysis. Also, you need to manually check for errors in your data, for example, whether a numeric variable mistakenly contains text. File-based data storage also means that multiple users can cause issues when accessing the same file, for example, if one user overwrites changes made by another user. Finally, file-based data storage can quickly get to its limits when we deal with large datasets. In most cases, the entire data contained in these files needs to be imported into your statistics package, where it stays as long as you work with it. This is not a problem if your dataset is not particularly large, but can be a real issue once you need to process a large amount of data. Even simple operations such as ordering or filtering your data can become extremely slow.

This is why for many applications, it is beneficial to store your data not in files, but in specialized data management software. We refer to these systems as “database management systems” (DBMS). DBMS have existed for a long time, and there are many different flavors. What they have in common is the ability to manage a set of databases for you. A database is a repository for all data required for a specific project. For example, if you intend to use a DBMS for a research paper, you would create a database for your project, which then contains a set of tables with all the data for this project. DBMS optimize the processing of the data contained in their databases. For example, some types of databases are designed for tabular data. They make sure that a table does not contain basic errors (e.g., non-numeric text in a numeric column), and they support the quick merging of tables that depend on each other. DBMS also facilitate efficient filtering and ordering of your data, and they can be accessed by different users concurrently. All this happens behind the scenes – users do not have to worry about where the data is physically stored, or how to enable fast and efficient access to them. Connecting to a database is possible from almost any type of statistical software or programming language – in this book, we use R to do this. Using R's database interface, you can send requests to the database server, for example, for changing or updating the data on the server, and for fetching data directly into R for further analysis.

1.5 TARGET AUDIENCE, REQUIREMENTS AND SOFTWARE

This book will be useful for any social scientist working with empirical data. Here, I define “social sciences” broadly, and include fields such as economics, sociology, psychology, anthropology, linguistics, communication studies and of course political science, my home discipline. I fully realize that there are significant differences across these fields when it comes to the predominant statistical methods they use; however, I also believe that these differences typically manifest themselves at the data analysis level. For example, while regression analysis is one of the main types of statistical approach used in my field (political science), psychologists may be more used to factor analysis. Importantly, however, there are few differences in the way the data needs to be prepared for these different types of analysis. In other words, the analysis dataset will likely be the same, regardless of whether it is later used in a regression model or a factor analysis. For that reason, the concepts and tools we cover in this book can likely be used across many different fields in the social sciences.

This is an applied book, and I try to illustrate our discussion with real examples wherever possible. Owing to my own background, these examples are largely drawn from my own discipline. At the same time, however, the book requires no substantive background in political science, and each example will be briefly introduced so that all readers, regardless of their background, can understand what research question we deal with, and what data we use in the example. Similarly, the book is designed to address social scientists of different generations. I believe that a deeper engagement with practical questions of data management will be useful for social science students as part of their training in empirical methods, or while working on their first research project. Still, the book also speaks to more advanced researchers at universities, governmental and non-governmental organizations, and private companies that have an interest in improving their quantitative research workflow.

I fully realize that the book’s readership will differ strongly in their technical experience, partly due to the variation in quantitative training that people have gone through. As regards the latter, it would be very difficult to custom-tailor this book to different statistical packages that readers have experience with. For this reason, I mainly use the R statistical toolkit in this book. R is free and open source, which means that there are no expensive licences to be purchased to work with the book. Also, R is one of the most flexible and powerful programming packages for statistical analysis out there, and can be used not just for estimating statistical

models, but also for advanced data management. Still, despite our focus on R in the code examples, many of the basic concepts and procedures for data management we cover are not tied to R, and therefore apply irrespective of which statistical software you use. While I provide many step-by-step examples in R, the book does not include a basic introduction to this software. It is therefore required that readers have some experience in R, either as part of their training in quantitative methods, or from one of the numerous R introductory books and courses that are available.

1.6 PLAN OF THE BOOK

The book consists of five parts. This and the following two chapters together constitute the introduction. So far, we discussed the role of data preparation and management within the research cycle, and defined the scope of the book. In Chapter 2, we go through the setup of the software used in this book. We rely mostly on R, but the advanced chapters on database systems require us to install a DBMS locally. Chapter 3 provides a conceptual introduction to data as a combination of informational content with a particular structure. We discuss the most important data structure in the social sciences (tables), but also talk about their design. In this first part of the book, we do not use any real examples for readers to practice – this starts only in the second part.

The second part of the book covers the processing of data stored in files. This is by far the most frequently used type of workflow in the social sciences, which is why we start with an overview of file-based data storage and different file formats in Chapter 4. In Chapter 5, we focus on data management with spreadsheet software such as MS Excel. This software is easy to use and most readers will be familiar with it. Still, it encourages certain bad practices for data management that we discuss (and caution against) in this chapter. We then turn to data management using R. The first chapter on this topic (Chapter 6) introduces R's basic features for reading data tables, as well updating and merging them. Chapter 7 presents a powerful extension of R's basic functionality: the *tidyverse* environment.

The third part of the book deals with data stored and processed in specialized systems, so-called databases. Here, data no longer reside in files, but are contained in database systems that users interact with via the network. This has a lot of advantages when it comes to avoiding errors and inconsistencies in the data, but also for handling large and complex

datasets accessed by several users. Relational databases for tabular data constitute probably the most frequent type of database, and we cover them in two chapters. We start with a single table in Chapter 8, and later extend this to several tables in Chapter 9. In Chapter 10, we address important technical features of relational databases, such as the ability to efficiently work with large datasets, or to allow collaborative access by different users.

The fourth part of the book addresses more specialized types of data that do not neatly fit into the standard tabular model. For each of these data types, we discuss (i) file-based data processing using R and the corresponding extension libraries and (ii) data processing using a database system. We start with a discussion of spatial data, that is, observations that have geographic coordinates attached to them (Chapter 11). The subsequent chapters cover text as data (Chapter 12) and network data (Chapter 13).

In the fifth part of the book, we conclude our introduction to data management with some recommendations for collaborative data projects, as well as for the publication and dissemination of research data.

The aim of this book is *not* to give readers detailed, in-depth introductions to the different tools and techniques we discuss. Rather, my goal is to convey a good intuition of the key features, but also the strengths and weaknesses, of the different tools and approaches for data management. This way, readers get a good overview of the available techniques, and can later choose a software and workflow that best matches their research needs.

Gearing Up

For this book, we rely on a number of different software tools, and with the exception of Microsoft Excel, all of them are available free of charge and for all major operating systems (Windows, macOS, or Linux). The most important one is R, a free, open-source statistical toolkit that comes with its own programming language. As stated in the previous chapter, it is required that readers have some experience in R, as the book does not include a basic introduction. The best way to work with R is to use RStudio, a powerful interface to the R engine. You will be able to complete Parts I and II of the book with R and RStudio only; if you also cover the more advanced chapters in Parts III and IV, you will also need the PostgreSQL database management system.

In this chapter, we go through the software required for the book. Detailed installation steps, as well as the sample datasets discussed in the book, are provided as part of the book's companion website at

<https://dmbook.org>

where you will always find up-to-date instructions and data. You do not have to install all the software tools below at once. It is perfectly possible to start with R and RStudio, and later return to this setup as you begin exploring the more advanced chapters on database systems, starting with Chapter 8.

2.1 R AND RSTUDIO

Please follow the installation instructions on the book's website to install the R statistical toolkit on your system. The R software includes the main

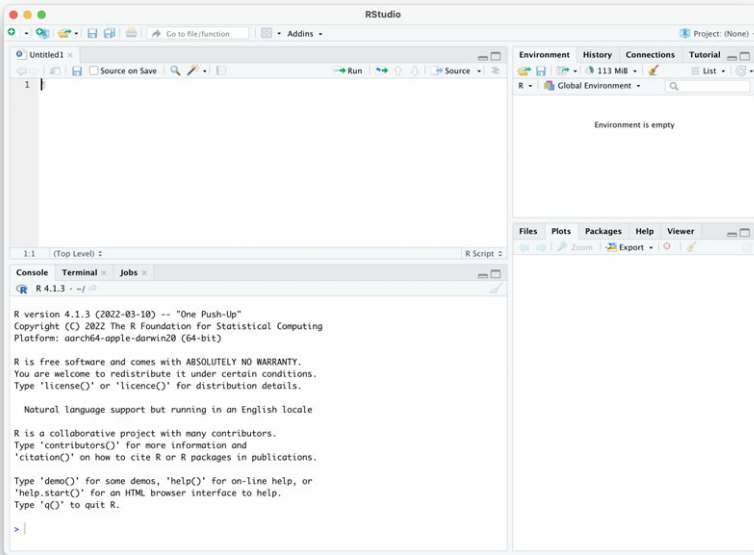


FIGURE 2.1. The RStudio interface.

engine that does most of the work: It executes the commands you enter, reads datasets, runs statistical models, and generates plots. The commands for doing this must be specified in the R programming language. While it is possible to work with R out of the box, I recommend that you also install a much more powerful development interface for R: RStudio. Instructions for this are also provided on the book's companion website. After the installation, start RStudio, and click on `File` `>>` `New File` `>>` `R Script`. Your RStudio window should now look like Figure 2.1.

Let us go through some of the main elements of RStudio. At the bottom left, you see the R console. This is where you see the output produced by R (unless this output is graphical). You can also use the console to send short commands to R. For example, if you type

```
Sys.Date()
```

on the console and hit `Return`, R will show you the current date. While you could do all the work for this book via the console by entering commands one by one, this is generally not a good idea since all this work would be lost when you close RStudio. This is why we typically work with *files* of R code, such as the one you created (which, at the moment, is still empty).

The files are displayed in the editor window of RStudio, at the top left just above the console. You can enter the aforementioned command in this window and save the file with **File** » **Save** under a specific name. This will preserve your R code such that you can later open and modify it. The green arrow at the top of the code editor allows you to run the currently selected part of code. It is absolutely essential for reasons of transparency and replicability that you properly store your code in files, so always use the code editor unless you are testing short commands!

On the right of the RStudio interface you can see two sets of panels, one with different panels called “Environment,” “History,” etc, the other with “Files,” “Plots,” etc. These panels become active once you continue to work with your R project. For example, if you load a dataset into R (which we will do in later chapters), you will see a new entry in the “Environment” panel that allows you to view the new dataset. Also, you can view graphics created by R in the “Plots” panel at the bottom. RStudio is an extremely powerful and versatile development environment for R, and we cannot go into more details here. There are a number of introductions available online, which you should consult if you want to learn more about RStudio’s features.

Nevertheless, I want to make one recommendation: Resist the temptation to use the different menu-based features in RStudio. For example, it is possible to read data using RStudio’s import feature under **File** » **Import Dataset**. This will internally execute one of R’s import functions for you. However, unless you save the corresponding R code displayed on the console explicitly as part of your R file, it will be lost when you close RStudio, making replication and error correction impossible. This is why I recommend that, wherever possible, you rely entirely on R code written by yourself, which you can properly save in your R file. This way, you later have a complete record of the individuals steps you carried out, which makes it possible to correct/extend your analysis when necessary, or share it with others so that they can replicate exactly what you did.

2.2 SETTING UP THE PROJECT ENVIRONMENT FOR YOUR WORK

The examples in the book cover many files and datasets, and they require a number of R packages. This setup has been prepared as a pre-configured RStudio project, to make it as easy as possible for you to get started. Go to the companion website for this book, which includes a link to this material. The download comes as a single zip file. Unpack the archive by

double-clicking, and move *the entire content of the archive* to a newly created folder that you would like to use as your main directory for the exercises, for example, `dmbook`.

R uses a given folder as a working directory, which is where it looks first when you open a file, or where it saves a file unless you specify a different path. For example, if the `dmbook` folder is placed in your Documents folder, then `Documents/dmbook` should be your main working directory (or “project” directory) for the book. Note that the directory paths provided here use the notation on macOS and Linux systems (with a forward slash / separating the different folder levels). On Windows systems, directory paths are specified using backslashes, for example `C:\Documents\dmbook`, so the paths will look slightly different.

Inside your main project directory, you will find a number of files and directories, which were originally contained in the archive you downloaded. This is roughly what your project directory looks like:

```
/Users/nils/Documents/dmbook/
├── ch04/
├── ch05/
├── ...
├── ch13/
├── dmbook.Rproj
├── ex04/
├── ...
├── ex13/
├── renv/
└── renv.lock
```

Let us quickly go through the most important folders and files. The data used in the code examples of the book is contained in the sub-folders (`ch04`, `ch05`, etc) for each chapter. If you follow the code examples in the book, you will need the files in these folders. Similarly, additional data for the exercises is contained in the sub-folders `ex04`, `ex05`, etc, again ordered by chapter. The file `dmbook.Rproj` is a project configuration file for RStudio. It is good practice to use these project files when working with RStudio. When you double-click this file (don’t do this yet!), RStudio will open a new session and switch to the directory containing the file as the working directory. The `renv` folder and lockfile contain the project setup for the book, which we introduce below.

R will treat all file names as *relative to the working directory, that is, the location of this project file*. There is one issue, however, that arises due

to the differences in how operating systems denote file paths. I mentioned earlier that macOS and Linux use forward slashes, and Windows uses backslashes. In our code examples, we often access files, for example, when importing data into R for further processing. To avoid including separate code examples for macOS/Linux and Windows, I rely on the built-in `file.path()` function that adjusts file paths depending on the underlying operating system. For example, if we want to access the file `csv-example.csv` in the `ch04` subfolder, we can simply refer to this file with

```
file.path("ch04", "csv-example.csv")
```

in our code, and R will automatically convert this to `ch04/csv-example.csv` for macOS or Linux, and to `ch04\csv-example.csv` if you use Windows. This file path is relative to the working directory, so we can omit the path to this directory (e.g., `/Users/nils/Documents/dmbook` or `C:\Documents\dmbook`).

2.2.1 R's Extension Libraries

One of the core strengths of the R system is its extensibility. There are thousands of packages for R that extend R's functionality in different ways. In this book, we rely on a number of these packages. Before you can load a package in an R session to use it, you must install it on your system. The standard way of doing this is via R's command line with

```
install.packages("tidyverse", dependencies = T)
```

This will make sure that apart from the new package itself, R will also install other packages that the new package depends on. Alternatively, you can use RStudio for installing packages, using [Tools >> Install Packages](#) in the menu bar.

Along with the code of the installed packages, you get the documentation of the functions it contains. It is absolutely essential that you learn to use this documentation, as it contains all the necessary information for you to use the package correctly and efficiently. In many cases, these documentations are not written as accessible introductions and may be difficult to read. This is why I give many pointers to useful functions and parameters, which you can then look up yourself if you need more details about how they work. The simplest way to display the reference for a function is the `?` operator, followed by the name of a package or a function. For example,


```
?install.packages
```

shows you the documentation for the `install.packages()` function in RStudio's help window on the bottom right.

Many users do not worry too much about package management and simply install packages in their main local library when they need them. This is R's default behavior, and it works just fine for most applications. However, I prefer a more sophisticated approach to package management: the use of different R *environments*. An environment is simply the set of all packages used for a particular project, such that each project keeps a separate list of packages (and their versions) it requires, without interfering with others. This also has the advantage of us being able to distribute a project along with a list of required packages, such that R can *automatically install all of them*.

We use the `renv` package to enable package management within an environment specifically for this book. The R project environment you downloaded above has `renv` enabled by default. If you double-click the `dmbook.Rproj` file that was distributed with the online material for the book, RStudio opens a new session and initializes the environment. It first downloads `renv` and does a check if all required packages (as specified in the `renv.lock` file) are installed. If packages are missing or are not available in the specified version, a warning appears. You can now type

```
renv::restore()
```

on the R console, and `renv` shows you a list of all required packages. After you confirm with `y`, it downloads and installs them. Note that the packages are installed in the respective version that was tested for the book, which is probably not the latest one. However, this is not a problem; in line with `renv`'s approach to compartmentalize installed packages into different environments, these package will be available only in the project environment we use for the book. This means that they are *not* available for your other projects unless you install them there as well. Under Windows, some package installations can fail, in particular for those where `renv` cannot find the pre-compiled version and instead relies on a source package. If you encounter this problem, I recommend that you do a

```
renv::equip()
```

and then try `renv::restore()` again.

2.3 THE POSTGRESQL DATABASE SYSTEM

We discussed in the introductory chapter that for certain applications, it is useful to keep your data in a specific system optimized for data storage and processing, a database management system (DBMS). There are different DBMS for different kinds of data, and in this book we will examine one of them in particular: The PostgreSQL database management system, which we use in Chapter 8 and the following ones. PostgreSQL is a *relational* DBMS designed for databases that contain tables, but it can also deal with more complex types of data. The installation process differs slightly between operating systems, which is why you should once again refer to the online repository to obtain more information required for the precise steps required (see the link at the beginning of this chapter). *Before proceeding, it is necessary that you complete the individual steps for your operating system described on the book's companion website.*

PostgreSQL is a multi-user system, and each user must identify with a username and a password. PostgreSQL installations under different operating systems use different approaches here. The default usernames differ, and some allow you to set your own password while others do not require a password (just a username). This is why after installing PostgreSQL, *make sure to memorize the username and the password* to access PostgreSQL on your system. The online installation instructions for this book contain more information about this.

With PostgreSQL set up on your computer, it is a good idea to test whether the connection works. In R, make sure that you have the RPostgres package installed along with all the other packages it depends on (if you use the pre-configured R environment described earlier, this is done automatically). The following code should then output the PostgreSQL version you are running. Make sure to adjust the username and password to match your setup (see the online instructions). `postgres` and `pgpasswd` are just placeholders, which we use here and later in the book – they may not work on your system.

```
library(RPostgres)
db <- dbConnect(Postgres(),
  user = "postgres",
  password = "pgpasswd")
dbGetQuery(db, "SELECT version()")
dbDisconnect(db)
```

2.3.1 Setting Up a New PostgreSQL Database

A database server can work with multiple databases, each of which is a collection of data that belong to one project. In this book, I follow the convention to use *a new database for each chapter of the book*, such that the examples and exercises for each of the chapters do not interfere with each other. The code below shows how to create a new database `dbintro`, which we use in Chapter 8 of the book. Again, make sure to adjust your username and password! You can use this code to create more databases for the subsequent chapters – just replace `dbintro` with the name of the database you would like to create. The code is presented here without much further explanation; in Chapter 8, we go through the process of connecting to the server step by step.

```
library(RPostgres)
db <- dbConnect(Postgres(),
  user = "postgres",
  password = "pgpasswd")
dbExecute(db, "CREATE DATABASE dbintro")
dbDisconnect(db)
```

2.3.2 Code Examples and Style

R allows you to be quite flexible in how you write your code, within the limits of the R syntax. To be consistent in the code I present in this book, I follow Hadley Wickham's *tidyverse Style Guide* (Wickham, 2021). Although it is designed for R code within the *tidyverse* framework (see Chapter 7), much of the recommendations also apply to code outside this framework. Here are some conventions used throughout this book:

- All file and directory names are lowercase. Different parts of the file name will be separated with a hyphen. Example: `csv-example.csv`
- R objects have lowercase names, and different parts of the object name are separated with an underscore. Example: `dataset_new`
- We use the `.R` ending for R code files.

For readers with an electronic copy of this book, it may be tempting to simply copy and paste the code examples into RStudio. Try not to do this. Rather, I strongly recommend that you *type the code yourself* and make modifications to it. This allows you to become more independent and experienced as an R user, but also to find out what does *not* work and why.

2.4 SUMMARY AND OUTLOOK

Data management requires a number of different tools, and in this chapter we covered those required for this book. Most importantly, we rely on the R statistical toolkit and the RStudio environment for most of the exercises. When you work in R, you mostly rely on data stored in files. This works for many applications, but sometimes our datasets become bigger and more complex. In these cases, it is useful to store data in specialized DBMS. These systems allow you to quickly search and filter large datasets, to check your data for consistency, or to manage access to the data by multiple users. We use the DBMS later to perform various operations, such as creating a database or loading data into it. With the technical preparations out of the way, we can now proceed to lay some theoretical groundwork. Chapter 3 discusses some general concepts about data, and introduces the most important data structure for the social sciences: tables.

Data = Content + Structure

Before we delve into the practical challenges of data processing, let us take a closer look at some core concepts we need throughout the book. The concept of scientific “data” is obviously of key importance. We need to clarify what we mean by it, and how information can be represented digitally as data. We will also learn to separate the data *content* – which refers to the actual information – from the logical *structure* in which this information is contained. Tables are by far the most frequently used data structure in the social sciences, which is why we spend a great deal of this chapter discussing the tabular data representation and its limits. We also review some basic functions of R: What are data frames, and how do we use them to store information? As discussed at the beginning, the book does not give a comprehensive introduction to R, but the examples below will help you refresh your memory.

3.1 WHAT IS DATA?

In our research, we use scientific data, which is systematically coded information about the real world.¹ Thus, we represent particular aspects of the real world by using *codes* so that this information can be stored as part of our dataset and later be processed by the researchers themselves, or by computers. In most cases, we will use numbers as codes, which represent, for example, the population of countries, or the vote counts of parties in

¹ Note that *data* is the plural of the Latin word *datum*. However, it is increasingly being used also as a singular word. Throughout this book, we follow the same convention. See also the blog post by Izzo (2012) about this.

an election. In other cases, we can use words as codes. For example, a list of political parties will likely include the party name, encoded as text in a particular language. Also, much simpler codes are possible, for example if we represent the presence or absence (0/1) of a particular feature (for instance, if a country is considered to be democratic or not).

Most scientific datasets are created to help us conduct comparisons between different entities – countries, precincts, experimental subjects, etc. This is why a dataset typically contains data about many different, yet comparable, entities. A social science dataset can be generated in different ways. In a survey, for example, we simply record the answers that subjects give to the specific survey questions. Here, the coding is predetermined by the way we design our survey and the questions we include. Other datasets are created by human coders, for example most of the cross-national datasets on political regimes or violent conflict. Yet another type of dataset requires little to no additional coding; for example, if we are interested in communication on social media, we can obtain a dataset of tweets directly from the Twitter platform. Here, again, information about each tweet will be encoded in a particular way, for example the date and time it was sent, or the name of the Twitter handle it was sent from.

The process of assigning codes to represent particular characteristics of real-world entities is sometimes simple (e.g., Twitter data comes with a precise time stamp readily assigned to each tweet), while it is much more difficult in other cases: For example, coding whether a country has a democratic system is difficult and subject to a major debate in political science. The challenges to coding and measurement in the social sciences are typically characterized by the requirements of *validity* and *reliability* that most readers will be familiar with; the former means that the coding or the measurement in a dataset should correspond to the theoretical concept we aim to capture, while the latter demands that the assignment of codes in our data be transparent, replicable, and uniformly applied across all the different entities we cover. These challenges arise at the data *collection* stage, which is why they are not discussed in this book. What matters for us is *how* data of particular types is represented and processed, but not *where* this information comes from.

3.2 DATA CONTENT AND STRUCTURE

In the previous section, we defined data as systematically coded information about the real world. For this data to be useful for scientific analysis,

we need to make sure that it is kept in a format that can be stored, shared, and analyzed. In other words, we need to find a good *representation* for it. Consider the following example:²

```
sdb <-  
"Switzerland is a country with 8.3 million inhabitants,  
and its capital is Bern. Another country is Austria;  
its capital is Vienna and the population is 8.7 million."
```

The simple object `sdb` is essentially a database; it contains information about two countries, their capitals, and their population. This information is what we call the *content* of the data. However, the information contained in this text may be obvious and easy to extract for humans, but it is much more difficult to process computationally. In other words, this data comes without a clear *structure*; unless we understand human language (which computers usually do not), we do not know what entities are referred to in the text, nor is it straightforward to locate the information about these entities. Now compare this example with the following method to set up a database, where we use the same content but with a given structure:

```
tdb <- data.frame(  
  country = c("Switzerland", "Austria"),  
  population = c(8.3, 8.7),  
  capital = c("Bern", "Vienna"))
```

In this example, we use R's default data structure for tables, a data frame, to create our database in a structured way. For each country contained in our tabular database `tdb`, we have different types of information, clearly labeled as such. In a table, each line typically refers to an observation, while the columns contain the different variables we have for the observations. This structure makes the second dataset much easier to understand and process as compared to the simple database `sdb` above. In short, while the two examples are the same in terms of content, they differ significantly when it comes to their structure. Almost all data we use in our work comes in tabular formats, and all statistical toolkits are designed to process data in tables. Despite the omnipresence of tables, it is, however, important to understand that a table is just *one type* of data structure; it is one that is very convenient for social science applications, but also has its limits, as we will see later.

² Population estimates for the following examples were obtained from the United Nations Department of Economic and Social Affairs (2019) and rounded.

3.3 TABLES, TABLES, TABLES

Tables (or so-called rectangular datasets) are the main type of data structure in the social sciences. They have *rows* and *columns*. In social science terminology, each row represents a *case* or an *observation*, and each column a *variable* in our dataset. Let us take a look at how R deals with tables, using again the data frame we created above. There are several standard operations we can perform on a table.

3.3.1 Accessing Data

R gives us several easy ways to access the information in our table. For example, we can access a single value by using the row and the column index. For example, Switzerland's (row 1) population (column 2) can be retrieved with

```
tdb[1,2]
[1] 8.3
```

Alternatively, we can filter out the entire record for Switzerland by omitting the column identifier, as in

```
tdb[1, ]
      country population capital
1 Switzerland      8.3     Bern
```

In general, the square brackets notation is used in R for subsetting. Here, we apply it to data frames, but it can also be used for simple vectors, matrices of numbers, etc. Note the comma in the expression, which indicates that the given number is a row and not a column index. Selecting particular columns can also be done by providing their indices (here, the range from 1 to 2) as follows:

```
tdb[1:2]
      country population
1 Switzerland      8.3
2  Austria      8.7
```

or simply by providing the name of the column:

```
tdb$population
[1] 8.3 8.7
```


The `$` operator extracts a column from the table as a vector. It is important to mention that R automatically keeps track of the kind of information that is contained in the columns of a data frame. In other words, it maintains *types* for the columns. In our above example, some information in our dataset is coded as text, for example the capitals of the two countries. These short pieces of text are also referred to as *strings* in computer science. Other variables contain *numbers*, such as the country populations. Let us check the types that R has assigned to our dataset:

```
typeof(tdb$capital)
[1] "character"
typeof(tdb$population)
[1] "double"
```

As you can see, the names of the capitals are stored in a column of type “character,” while the population estimates are of the type “double,” which is the default type for numeric information. There are several other data types for vectors in R (such as “logical” values that can be either TRUE or FALSE, or the “integer” type used for storing integer numbers).

Oftentimes, we want to extract only a subset of the table that satisfies a particular filtering criterion. For example, we can extract the records for Switzerland (which, in our case, is only one) using:

```
tdb[tdb$country == "Switzerland", ]
  country population capital
1 Switzerland      8.3   Bern
```

Here, the `tdb$country == "Switzerland"` expression internally calculates a set of indices for those rows where the country column contains Switzerland. As above, we need to use the comma operator to tell R that the filtering condition we apply (the specification of a particular country name) applies to the rows of the table. If you think this expression is too complicated, there is also a simpler way to subset tables using the `subset()` function:

```
subset(tdb, country == "Switzerland")
  country population capital
1 Switzerland      8.3   Bern
```

3.3.2 Updating Data

Updating the information in a table is also straightforward. We can use the indexing notation again to update particular values in the table, for example, Switzerland's population:

```
tdb[1,2] <- 8.4
tdb
```

	country	population	capital
1	Switzerland	8.4	Bern
2	Austria	8.7	Vienna

This, however, is not convenient, since we have to refer to a column using the index and not the name. Instead, we can do the following:

```
tdb[1, "population"] <- 8.3
tdb
```

	country	population	capital
1	Switzerland	8.3	Bern
2	Austria	8.7	Vienna

This is still not optimal, since we need to know Switzerland's row index. To identify the rows for Switzerland, we can again use the statement we introduced above:

```
tdb[tdb$country == "Switzerland", "population"] <- 8.2
tdb
```

	country	population	capital
1	Switzerland	8.2	Bern
2	Austria	8.7	Vienna

3.3.3 Adding Data

Adding new data to a table can be done by either (i) inserting new rows or (ii) adding new columns. The latter can be done by simply assigning values to the new column:

```
tdb$area <- c(41, 83)
tdb
```

	country	population	capital	area
1	Switzerland	8.2	Bern	41
2	Austria	8.7	Vienna	83

Inserting rows to our table is done using the `rbind()` function, which “binds” rows together. You can use it to combine two tables into one

(provided they have the same structure), but here we use it to add a single line:

```
tdb <- rbind(tdb, c("Liechtenstein", 0.038 , "Vaduz", 0.16))
tdb
```

	country	population	capital	area
1	Switzerland	8.2	Bern	41
2	Austria	8.7	Vienna	83
3	Liechtenstein	0.038	Vaduz	0.16

Note that `rbind()` creates a new data frame from the inputs it gets. Therefore, we need to store the newly created table again in the original variable, which essentially deletes the old `tdb`.

3.3.4 Deleting Data

Finally, we also need to demonstrate how to remove data from our table. Again, there are two possible operation for deletions, namely, those affecting the columns and those affecting the rows of the table. Deleting columns is simple:

```
tdb$area <- NULL
tdb
```

	country	population	capital
1	Switzerland	8.2	Bern
2	Austria	8.7	Vienna
3	Liechtenstein	0.038	Vaduz

The deletion of rows from an R data frame may not be completely intuitive, as you need to create a subset of the rows you would like to keep, and overwrite the old data frame. This can be done, for example, using the `subset()` function we have described above:

```
tdb <- subset(tdb, country != "Liechtenstein")
tdb
```

	country	population	capital
1	Switzerland	8.2	Bern
2	Austria	8.7	Vienna

In this statement, we subset our data frame to those rows where the country column does *not* equal Liechtenstein, and store the result in the `tdb` variable.

3.4 THE STRUCTURE OF TABLES MATTERS

Before we start digging into actual data using different tools, let us spend some more time thinking about tables and their structure. While for many applications, it is entirely obvious what columns you need in your table, in some cases finding a good structure for your table is not as straightforward as it seems. This is why we will take a closer look at a few more toy examples, so as to better understand why and how the structure of tables matters. The recommendations here constitute the traditional way to organize data, which applies to most applications and projects we deal with in the social sciences.

3.4.1 Tables Should Grow Down, Not Sideways

A general rule of thumb you should observe when defining a tabular structure is that the columns – that is, the variables in the table – should be independent from the observations it eventually contains. That is, you need to select columns that capture all the important aspects of your data, regardless of how many cases/rows you later add to the table. A common mistake we oftentimes see is the use of case-specific information in the column *names* rather than in the individual cells of the table. This happens frequently in cross-sectional time series data, which is data about different entities (e.g., countries) that are observed at multiple time points (e.g., years). Consider our example from above, now revised to record the country population in different years:

```
bad_table <- data.frame(
  country = c("Switzerland", "Austria"),
  pop1950 = c(4.7, 6.9),
  pop1960 = c(5.3, 7.1),
  pop1970 = c(6.2, 7.5))
bad_table
```

	country	pop1950	pop1960	pop1970
1	Switzerland	4.7	5.3	6.2
2	Austria	6.9	7.1	7.5

This format is called a “wide” table. The setup of the table may be convenient for human readers, but it causes many issues when processing the data computationally. It obviously violates our requirement that the variables we record in the dataset (which constitute the columns in the table) should be independent from the set of entities we record these characteristics for. In the above example, when adding population estimates

for more recent years, we would have to add more columns, rather than rows, to the table. This is not an issue in itself, but this structure is difficult to work with if we want to perform simple calculations on our table. For example, suppose we want to compute the average population across different observations in our table. This is easy to do by year:

```
mean(bad_table$pop1950)
[1] 5.8
mean(bad_table$pop1960)
[1] 6.2
mean(bad_table$pop1970)
[1] 6.85
```

However, what if we are interested in the average across all countries and years? With the table above, this is more difficult:

```
mean(c(
  mean(bad_table$pop1950),
  mean(bad_table$pop1960),
  mean(bad_table$pop1970)))
[1] 6.283333
```

This still looks acceptable, but now imagine that we are adding observations for more years to our dataset. This will make the table grow sideways, not down. If we compute the average population from the table, the statement becomes longer and longer. And, even more problematic, we need to update the calculation of the average population *every time we add a new year* to our table, which is not very convenient. How, then, is it possible to fix this? Can we design a better table structure for time series data? Consider this example:

```
good_table <- data.frame(
  country = c(rep("Switzerland", 3), rep("Austria", 3)),
  year = c(rep(c(1950, 1960, 1970), 2)),
  population = c(4.7, 5.3, 6.2, 6.9, 7.1, 7.5))
good_table
```

	country	year	population
1	Switzerland	1950	4.7
2	Switzerland	1960	5.3
3	Switzerland	1970	6.2
4	Austria	1950	6.9
5	Austria	1960	7.1
6	Austria	1970	7.5

This format is called a *long* table. The main difference between the `bad_table` and the `good_table` is obvious: Rather than using table columns for different years, we now introduce a new column `year` to link population values not just to the respective country, but also to the year they refer to. This makes working with our table much easier: Adding observations for more years is simple in this table structure; we can just append more rows to the table. Also, computing the average population over all observations is now a simple operation:

```
mean(good_table$population)
```

```
[1] 6.283333
```

You will never have to change this statement, regardless of how many observations and years you are adding to the data frame. Readers may now wonder how we get the annual average out of this table, which was easy in the `bad_table` above. For the `good_table`, we do this by letting R compute averages over *groups* of data, rather than the entire set of observations. This is called *aggregation*. One way to perform an aggregation in R is by using the `summaryBy()` function in the `doBy` package:

```
library(doBy)
```

```
summaryBy(population ~ year, data = good_table, FUN = mean)
```

	year	population.mean
1	1950	5.80
2	1960	6.20
3	1970	6.85

In the statement above, we need to specify which variable we would like to aggregate over (`population`), and which variable(s) we would like to use for grouping (`year`). Also, we need to tell the function what the data frame is for the aggregation (`good_table`), as well as the summary function we would like to use (`mean`). The function then combines all observations with the same values in the grouping variable, and applies the summary function to each of these groups. This is exactly what we need, and it returns the annual averages from our dataset. So overall, the structure in our `good_table` seems to be much easier to handle, at least when we process our data computationally. You still see many examples similar to the `bad_table`, which may be due to the fact that they can be easier to understand for human readers. As we will see later, spreadsheets such as Excel are useful when humans interact manually with data, but not when we try to push the automation of data processing for maximum efficiency and transparency, which is our aim in this book.

3.4.2 One or Multiple Tables?

The above example showed us that there are good and bad ways to structure individual tables. We now turn to the question of *how many* tables we need for a good representation of our data. Again, let us consider the `good_table`. Let us assume that, in addition to the yearly population estimates, we would like to store information about national capitals, like we did in the examples above. The simplest way to do this is to add the names of the capitals in a new column:

```
good_table2 <- good_table # create a copy to keep the original one
good_table2$capital <- c(rep("Bern", 3), rep("Vienna", 3))
good_table2
```

	country	year	population	capital
1	Switzerland	1950	4.7	Bern
2	Switzerland	1960	5.3	Bern
3	Switzerland	1970	6.2	Bern
4	Austria	1950	6.9	Vienna
5	Austria	1960	7.1	Vienna
6	Austria	1970	7.5	Vienna

Since national capitals rarely change, the information in the capitals column is essentially constant over the years in our dataset, and we need to repeat it for every single year in the dataset. From a data representation point of view, this is clearly not optimal, as we have *redundant* information in our dataset. This makes data maintenance more difficult and error-prone. First, inserting the information in the first place is cumbersome, since we have to copy and paste the name of the capital of a given country for each year this country is listed in the dataset. This may be easy in our toy example, but quickly becomes infeasible when we deal with a much longer time series. Also, updating the data is equally difficult, for example, if we decide to refer to the capitals not in English, but in the respective national language (which would require us to replace Vienna with Wien). Redundant information also means that we can have inconsistencies in our data; for instance, if we forget to update all instances of Vienna, we may end up with a dataset that sometimes refers to the capital of Austria as Wien, while in other cases it uses the English name.

The problem of data redundancy always comes up if we store information about different entities that refer to each other in a single table. In our example, we have two types of entities: the countries (each of which has a capital), and the country-years (each of which has a population estimate). This data structure should better be stored in two tables that link to each other. For example, rather than adding a new column to `good_table`, we

could *add a new table* that contains only the information on the national capitals:

```
populations <- data.frame(
  country=c(rep("Switzerland", 3), rep("Austria", 3)),
  year=c(rep(c(1950, 1960, 1970), 2)),
  population=c(4.7, 5.3, 6.2, 6.9, 7.1, 7.5))
populations
```

	country	year	population
1	Switzerland	1950	4.7
2	Switzerland	1960	5.3
3	Switzerland	1970	6.2
4	Austria	1950	6.9
5	Austria	1960	7.1
6	Austria	1970	7.5

```
capitals <- data.frame(
  country=c("Switzerland", "Austria"),
  capital=c("Bern", "Vienna"))
capitals
```

	country	capital
1	Switzerland	Bern
2	Austria	Vienna

As a result, our database now consists of two tables: a capitals table with country-level information (in our case, only the capitals) and a populations table with information at the country-year level (in our case, population estimates). In this setup, each piece of information is contained *only once* in the dataset; in other words, we have eliminated redundant data. This makes data maintenance extremely easy. For example, if we want to adjust the name of the Swiss capital, we do this in exactly one place:

```
capitals[capitals$country == "Switzerland", "capital"] <- "Berne"
capitals
```

	country	capital
1	Switzerland	Berne
2	Austria	Vienna

The split of data into several tables is clearly something that may be desirable from a data management point of view, as it reduces (and, ideally, eliminates) redundancies in our data. At the same time, it is likely not a good way to interface with software for statistical analysis, most of which requires the data to be nicely arranged in a single rectangular table. What can we do about it? The solution to this is what we alluded to in

Chapter 1: the need to separate (i) data processing and management and (ii) data analysis into different stages of our workflow, potentially using different software tools supporting these stages. Recall that in Chapter 1, I recommended that you create “analysis datasets,” which are tailored to the respective analysis and the software you use at the analysis stage. For our example, if we need information from the populations and the capitals tables in a single, rectangular format, we can simply merge the two tables:

```
merge(populations, capitals, by = "country")
```

	country	year	population	capital
1	Austria	1950	6.9	Vienna
2	Austria	1960	7.1	Vienna
3	Austria	1970	7.5	Vienna
4	Switzerland	1950	4.7	Berne
5	Switzerland	1960	5.3	Berne
6	Switzerland	1970	6.2	Berne

Of course, we would only do this once we have finished the processing of our data, since we introduce redundancy in the merged dataset. Later in this book, we will deal with relational databases, which are designed to work with many tables at the same time, thus providing a suitable way to manage even complex datasets.

3.5 SUMMARY AND OUTLOOK

In this chapter, our main focus was the distinction between the content and the structure of data. Data without structure (such as human speech, for example) can be difficult to process computationally, since it is difficult for computers to locate the important bits of information. In the social sciences, research data is usually collected and stored in tabular data structures. Tables are omnipresent, and they constitute the main way in which most statistical packages import and process data. In its simplest form, a tabular data structure is very easy to handle. It only requires us to specify

- A set of columns and their names (which correspond to the variables in our dataset)
- The types of each of these columns (a number, or a string of text)

We can then insert rows into the table, which represent the different observations in our dataset. Of course, these rows need to conform with the table definition, such that the columns contain the correct type of information. Note that while most software toolkits (such as R) keep

track of the type of information stored in the columns of a table, they cannot check for other sorts of errors. For example, if you record the age of respondents in a numeric column and mistakenly enter the value 200, R will not complain. Therefore, it is up to you to identify semantic errors in your data and correct them.

Because of the importance of data structure, working with research data usually requires us to think about data content *and* structure. Before we can populate a dataset with information about survey responses, country-level indicators, or conflict events, we need to define what our dataset should look like, or, in other words, what its structure should be. This is usually referred to as *data definition*. Once we have a structure for our data, we can fill it with new information, update existing information, or delete parts of it. Together, these operations are referred to as *data manipulation*. Last, we use our dataset for scientific analyses, which is why eventually we need to output it in some way that is suitable for processing with other tools. This is called *data extraction*.

In this chapter, we also took a closer look at the structure of tables. In particular, I showed that it is beneficial to choose a table structure that lets your table grow down, not sideways, as you add more data. Also, I demonstrated that you may be better off splitting your data into separate tables, in particular if you deal with different types of entities. You may wonder why we spend so much time thinking about table structure, as this question is entirely straightforward to solve for many applications. This is true, but table structure matters a lot as soon as we deal with more complex scenarios. In particular, as soon as our observations vary along more than one dimension (e.g., countries and years), choosing a sub-optimal table structure can make your life difficult. By introducing some important considerations about tables and their design, we pave the way for later topics we cover in this book, in particular relational databases. In short, it pays off to think about the table structure before you start collecting your data. If you rely on existing data, you may benefit from transforming a given table to a more suitable design, such that you can optimize your research workflow down the road.

Now that we have completed a basic introduction using some toy examples, it is time to do some real work. In the next chapter, we will start with several tools that rely on *file-based* data storage. This means that your data is contained in files; you temporarily open these to process your data, and later the save the result again to a file. In later parts of the book, we discuss an alternative approach, where your data is stored in a database.

PART II

DATA IN FILES

Storing Data in Files

In this and the next chapters of this book, we will focus on tools for data contained in files: Your data resides in physical files on your hard disk, from where it is opened with a software of your choice, processed in various ways, and then stored again in a file. This is by far the most common workflow used in social science projects. Why do we need files at all? The answer is very simple: We use files for permanent data storage. When you work with a dataset in R (or in some other software, such as Excel or Stata), the table(s) – for example, the data frames in R – are temporarily stored in your computer’s main memory. This is the part of your computer where data and programs are kept for fast access during the actual operation of your system. The problem is that this volatile memory does not function anymore when you turn off your system, and the entire content (and, thus, your data) disappears. Therefore, every computer has another type of data storage that remains *persistent* even when the system is shut down. This is usually your hard disk drive, but it can also be a network drive or a cloud storage folder.

When we save tabular data contained in the computer’s main memory to files, we need to make sure that the tabular structure is preserved. Recall our discussion of the importance of data structure in the initial chapters of this book – a persistent storage of data in files would be useless if the actual structure of the data were lost. Therefore, there are different ways in which tabular data can be stored as files, such that the tabular structure is preserved. For a given file, the *file type* typically indicates if it contains a data table and how this table is stored in the file. You are probably familiar with file types for text documents (e.g., the Word format indicated by the .docx extension) or for graphics (e.g., the JPEG format using the .jpg

or .jpeg extension). Similarly, there are different file types to store data tables. These file types are designed to keep the logical structure of our data table as a set of columns of particular types, and a set of rows. These file types constitute the main focus of this chapter.

4.1 TEXT AND BINARY FILES

Before we go through the list of the most commonly used file formats for data in the social sciences, we need to make a basic distinction between text and binary files. As the name suggests, *text* files contain information stored as plain text, such as program code for R and other programming languages. This is why you can open them with any text editor (such as the one built into RStudio) and view the contents. In contrast, *binary* files can be used and processed only by particular software tools – they essentially contain only 0s and 1s that make little sense to humans (but can be understood by the software tools designed for them). The term “binary” applies to all files that are not text and is used for many different file types, not just those that contain tabular data. For example, images, video, and sound are typically stored in binary files. To illustrate the difference, Figure 4.1 shows the contents of a binary file, viewed with the Unix `hexdump` command.

As you can see, the information in a binary file is not human-readable – the contents are completely cryptic and can only be processed by software designed for this file type. In contrast, the content of a text file can be understood by humans. You can create and open text files even with RStudio: Just choose `File >> New File >> Text File`, and you get a new editor pane, where you can start adding content to your text file and save it (see Figure 4.2).

The screenshot shows that text files contain text and numbers, but also various other invisible characters that are usually hidden in the text editor. Under `Tools >> Global Options`, you can turn on/off the display of these characters in the “Code” section in RStudio’s preferences menu, in the “Display” pane. Just tick the box for “Show whitespace characters,” and your text file will look similar to the one in Figure 4.2. There are

```
00000000 d0 cf 11 e0 a1 b1 1a e1 00 00 00 00 00 00 00 00 |.....|
00000010 00 00 00 00 00 00 00 00 3e 00 03 00 fe ff 09 00 |.....>.....|
00000020 06 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 |.....|
00000030 27 00 00 00 00 00 00 00 00 10 00 00 29 00 00 00 |!'.....)|...|
00000040 01 00 00 00 fe ff ff ff 00 00 00 26 00 00 00 |.....&.....|
00000050 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |.....|
```

FIGURE 4.1. Contents of a binary file.

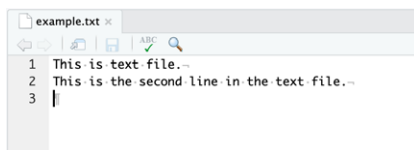


FIGURE 4.2. Example of a text file viewed in RStudio.

different invisible characters in the text file above; for example, white spaces (denoted as gray dots in the editor) are used to separate words, and there is a linebreak character at the end of each line. The end of the entire file is again marked with a specific character.

The set of characters you can use in a text file is defined by the file encoding. A file encoding is a mapping of numbers (which the computer stores internally, so nothing you need to worry about) to actual characters. There are lots of different file encodings for computers, partly because there was a need to encode different human languages and their special characters. Luckily, however, most conversion issues can now be avoided due to the Unicode standard, which accommodates most languages and special characters worldwide. Still, you may encounter other encoding standards, so watch out if you use (or if your data contains) special characters. To demonstrate what can go wrong if you choose the wrong file encoding, open the `un-secretaries.txt` file in the RStudio editor. You should see a list of UN Secretaries General, ordered by the year they served. Note that there are two special characters in this list: Dag Hammarskjöld's name contains an "ö" (an o-umlaut), and António Guterres's first name is spelled with an "ó" (an o-acute). This file is encoded in Unicode, which is the default for macOS and other current systems.

Before we start the conversion to a different encoding scheme, save the file under a different name with `File >> Save As...`, so that we do not overwrite the original version. Now, let's save the file in a different encoding. Go to `File >> Save with Encoding`, which brings up the dialogue box in Figure 4.3.

The current coding of the file is *UTF-8*, which refers to the *Unicode* standard. This is also the default for my current operating system (macOS) and therefore labeled as such in the list. Now, select *ASCII* and click OK. This will transform the file to the *American Standard Code for Information Interchange* (ASCII) standard, which is an old encoding standard developed in the USA to encode text in English (this is the file

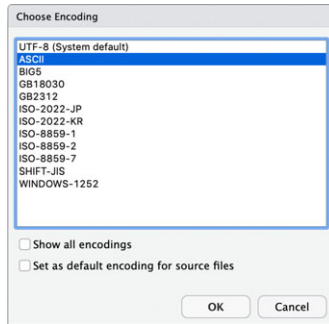


FIGURE 4.3. Choosing the file encoding in RStudio.

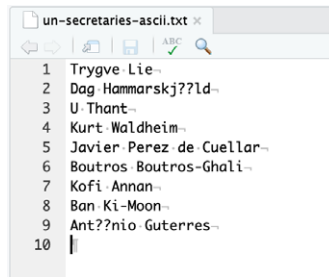


FIGURE 4.4. Viewing a file encoding in ASCII.

`un-secretaries-ascii.txt` included in the supplementary material for this chapter). If you now close the file and open it again, Figure 4.4 shows what you get.

This looks almost the same as the old file, but there are differences in two places: The special characters are missing. It is not difficult to understand why: Since the ASCII encoding does not include characters such as “ö” and “ó”, they are simply replaced in the converted file (in our case, with two question marks). This example illustrates that if you leave the Unicode world and deal with text files in other encodings, you need to be careful, since special characters and symbols can be transformed in unexpected ways or simply disappear.

What if you encounter a file and are not sure about its encoding? Unfortunately, it is not straightforward to recognize the encoding. The `readr` package offers a function that guesses the encoding of a given file together with a confidence estimate. The following example demonstrates this:


```
library(readr)
guess_encoding(file.path("ch04", "un-secretaries.txt"))$encoding
[1] "UTF-8"
guess_encoding(file.path("ch04", "un-secretaries-ascii.txt"))$encoding
[1] "ASCII"
```

The function detects the encoding of the original UN secretaries file correctly, and also for the ASCII version we created. This may be helpful if you encounter conversion errors during the import that could be due to encoding (such as garbled characters) – in this case, you can try to set the encoding manually (e.g., by using the `fileEncoding` parameter in R's `read.csv()` function) to fix these problems.

4.2 FILE FORMATS FOR TABULAR DATA

There is a variety of file types designed for storing tabular data. The discussion below takes you through the most important ones, many of which you will already be familiar with. In the remainder of the book, we will encounter several other file formats that can be used for tables, but also for other types of data.

The format of a file is typically indicated by the file extension, which is the dot and the letters at the end of a file name. For example, MS Excel files use the extension `.xlsx` (or `.xls` for the legacy Excel format), while simple text files are usually marked with `.txt`. It is important to note, however, that the file extension is no guarantee that the file actually conforms to a particular format. For example, you can easily rename an Excel document such that it ends in `.docx` (the file extension for Word documents). If you then double-click your file, your operating system calls Word to open it, since it believes that this is a Word file because of the file extension. Word, however, cannot open the file, since internally it uses the Excel format.

On some operating systems such as macOS and Windows, file extensions are hidden by default and you might be wondering what we are talking about. To show the file extensions on all files on your computer, follow these steps. On Windows:

- Open Windows Explorer
- Expand the Ribbon menu (Shortcut: `Ctrl` + `F1`)
- Click on the “View” tab
- Check the box that says “File name extensions”

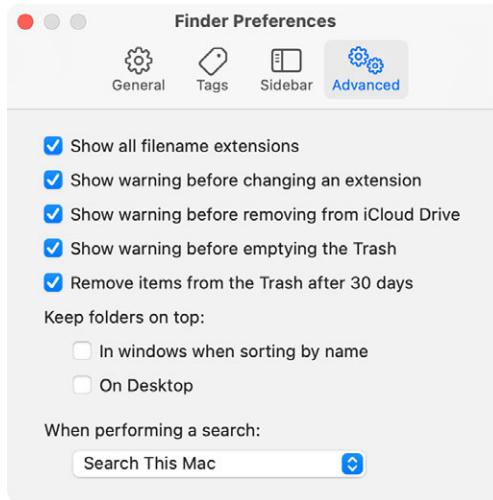


FIGURE 4.5. Check the first box to display all filename extensions on macOS.

On a Mac:

- Open the Finder app
- Click on `Finder >> Preferences`
- Click on `Advanced`
- Check “Show all filename extensions” as shown in Figure 4.5

Linux usually does not use file extensions to determine how to open a file and just considers them part of the filename. You will not need to change any settings if your computer runs Linux.

In the following sections of this chapter, I will briefly introduce the most common file formats you are likely to encounter when working with social science data. For each of these formats, we cover some general features, as well as how to open and save it in R. The discussion starts with several *text* file formats typically used for storing tables. As we have seen above, the advantage of using text files is that you can manually check the content of a file. Also, almost any software tool for data analysis can read and write text-based files with tabular data. At the same time, however, there is no real standardization: This means that the file import can go wrong, and you need to check that the imported table actually corresponds to what is in the file and what you expect.

4.2.1 CSV Files

The format most often used for storing tabular data in files is the *Comma-Separated Values (CSV)* format. In a CSV file, each line in the text represents a row from the table, and the cells in that line are separated by a special character such as a comma (hence the name). As in any proper tabular structure, each line must have the same number of cells for the table to be perfectly rectangular. When I use the term “CSV” in this book, I mean any kind of text file that stores tables in the same way (possibly using characters other than the comma as field separator). These files sometimes use the .dat file extension, but also others. Let us take a look at an example of a CSV dataset. Open the file `csv-example.csv` that is part of the data repository for this chapter in RStudio’s text editor. The file contains distances between all national capitals worldwide, compiled by Gleditsch (2020). For simplicity, we use only a subset of it – the distances between Washington, DC and other countries’ capitals. This is what you should see in the first three lines:

```
numa,ida,numb,idb,kmdist,midist
2,USA,20,CAN,738.31,460.56
2,USA,31,BHM,1639.23,1022.12
2,USA,40,CUB,1831.13,1141.3
```

The first line is the header of the table and contains the names of the six columns. Each distance is measured between the capital of one country, which has an identifier and a name (`numa` and `ida`), and a second country, also referenced with an identifier and a name (`numb` and `idb`). Finally, the distances are provided in kilometers (`kmdist`) and miles (`midist`). The data start in the second row. The cells are separated with a comma (the *field separator* character), and each line ends with an (invisible) newline character. Let us now import this file as an R data frame and take a look at the first three lines:

```
csv <- read.csv(file.path("ch04", "csv-example.csv"))
csv[1:3, ]
  numa ida numb idb kmdist midist
1    2 USA  20 CAN  738.31  460.56
2    2 USA  31 BHM 1639.23 1022.12
3    2 USA  40 CUB 1831.13 1141.30
```

There is no single standard for CSV files, which is why they come in many different forms. One issue you may encounter is that a .csv file

uses a separator other than a comma as the separator. Open `csv-example-semicolon.csv` in RStudio's text editor, which contains a slightly modified version of the original data. This is what you should see:

```
numa;ida;numb;idb;kmDIST;midist
2;USA; 20;CAN; 738,31; 460,56
2;USA; 31;BHM; 1639,23; 1022,12
2;USA; 40;CUB; 1831,13; 1141,30
```

In this file, the cells are separated by a semicolon instead of a comma, and the comma is used as a decimal indicator (which is the standard in many countries in Europe and elsewhere). This clearly illustrates the problems that can arise when using CSV and related file types in the absence of a fixed definition of a file format: is the comma or the semicolon used as field separator? We can clearly see this when eyeballing the file, but it is not straightforward for the software we use. So when you try to import `csv-example-semicolon.csv` in the way we showed you above, this does not work:

```
csv_semicolon <- read.csv(file.path("ch04", "csv-example-semicolon.csv"))
Error in read.table(file = file, header = header, sep = sep, quote = quote, : more
columns than column names
```

The problem is that R's `read.csv()` function by default assumes a comma as the separator. This results in a mismatch between the header of the file – which is treated as only one column name, since it does not contain a comma – and the actual data, which, when split at the comma characters, has *three* fields per row. To correctly import this dataset, you have to specify the separator explicitly:

```
csv_semicolon <- read.csv(file.path("ch04", "csv-example-semicolon.csv"),
  sep = ";")
csv_semicolon[1:3,]
  numa ida numb idb kmDIST midist
1    2 USA  20 CAN  738,31  460,56
2    2 USA  31 BHM 1639,23 1022,12
3    2 USA  40 CUB 1831,13 1141,30
```

A similar issue arises when dealing with strings that contain the field separator. For example, we may want to add the name of the first country's capital, Washington, DC, to our capital distances data. This can lead to confusion when importing the dataset, since R (or other tools for that matter) will interpret the comma in the capital name as

a field separator. For that reason, CSV files often enclose the affected strings with double-quotation marks ("), which basically means: "Treat the entire content between the quotes as a single string, regardless of what it contains." To see how this works, open the file `csv-example-quotes.csv` in RStudio's text editor, and take a look at the first line below the header:

```
2,USA,20,CAN,738.31,460.56,"Washington, DC"
```

The dataset now has a seventh column with the name of the capital of the first state. Since this name contains a comma, the entire string is enclosed in double quotes. But as you may have guessed, this again does not fully solve the issue. What if your string variable contains " characters that are *not* used for quotation? The standard way of dealing with this is to replace them with "" (two double-quotation marks next to each other). When you export files as CSV, the software usually takes care of string quotation. However, when you import CSV files, it might still be the case that quotes are not handled properly and errors occur, so you need to be careful and double-check that the import works correctly. If it does not, in many cases your only option is to open the file in a text editor, identify the source of the error, and fix it manually.

Saving CSV files from R is simple. R provides the `write.csv()` function for doing this, which is part of R's basic set of functions. By default, `write.csv()` uses comma as the field separator, and string quotation is enabled by default. This is all fine, but by default, the function produces a file that looks like this:

```
write.csv(csv, file.path("ch04", "output.csv"))
```

```
","numa","ida","numb","idb","kmdist","midist"
```

```
"1",2,"USA",20,"CAN",738.31,460.56
```

```
"2",2,"USA",31,"BHM",1639.23,1022.12
```

```
"3",2,"USA",40,"CUB",1831.13,1141.3
```

First, note that the function by default quotes all strings, regardless of whether this is necessary. In our case, none of the field names or the data contain a comma, so we could actually omit the quotes in the header and in the string variables in the data. Second, R adds a new (unnamed) column to the data. This column contains the row numbers, which is what R uses to preserve the order of the data in the file. In practice, however, the ordering of rows in a data frame oftentimes does not (and should not) matter, which is why we recommend that you disable this feature:

```
write.csv(csv, file.path("ch04", "output.csv"), row.names = F)
```

There are a number of other useful parameters for the `write.csv()` function, most importantly the `sep` parameter that lets you define a field separator to be used for the file. The `col.names` parameter allows you to disable the inclusion of a header should you wish to do so (although this is generally not recommended). In addition, there is one important feature of R read/write functions that is very useful when dealing with large files: You can use it to *compress* files. File compression (“zipping”) is a technique where text files are saved such that they reduce the space they need on disk. For example, our (uncompressed) dataset of capital distances needs about 6 kilobytes of disk space. However, if we use R file compression, we can reduce the size considerably. This is done by using R’s functionality to create “gzipped” files, a frequently used algorithm for compressing files:

```
write.csv(csv, gzfile("csv-example.csv.gz"), row.names = F)
```

The compressed file now only uses around 3 kilobytes, which is about 50% of the size of the original file. For larger files, the size reduction is usually much higher. Compression works particularly well if your data contain long sequences of repeated characters, which is typically the case for tables with a lot of text. Of course, R can also read the zipped CSV files again – there is no need for using additional functions, and you can simply provide the name when importing it, as in `read.csv("csv-example.csv.gz")`.

In this section, we only covered the basic features of CSV files, and the standard way to process them in R. There are various other packages and functions for this, some of which we will introduce in the next chapters. As regards the CSV format in general, it is important to emphasize again that it is only a *convention* for using text files to store tabular data rather than a fixed standard. While there actually exists a standard for CSV files (Shafranovich, 2005), it is not widely known and most tools (including Excel) do not conform to it, so you can safely ignore it. This means that you need to be aware of the potential pitfalls when using CSV files. We discussed the most common ones, which include the file encoding, the definition of the field separator (a comma, a semicolon or the invisible tab character `\t` are the typical choices), and the quotation of strings. Also, unlike in our examples above, CSV files sometimes do not contain the headers of the table in the first line, in which case you would have to set them manually in your code.

Still, the CSV format has a number of advantages that explain why it is so widely used: It is an open format that is completely transparent, since you can open a CSV file on just about any computer system and inspect its contents. For that reason, the CSV format is compatible with most data processing and data analysis tools, and belongs to the most commonly used file types for data storage. One important downside is the lack of meta-information (such as column types or documentation information), so this must be provided in associated data such as codebooks or README files.

4.2.2 Excel

In the previous section, I described how to use text files to store tabular data. Now, we are turning to a number of binary data formats for the same purpose. However, as we discussed above, binary files are oftentimes designed to be used with a particular software and cannot be inspected manually with a simple text editor. A well-known example of this kind is the MS Excel file format, which, due to the popularity of the MS Excel spreadsheet software, is still a widely used format also for social science data. Excel files come in one of two formats: the legacy `.xls` format (which is a truly binary format), and the current `.xlsx` format that is actually a zipped collection of different text files, which together contain the data.

Different packages allow you to read and write Excel files in R. I recommend the `readxl` library for this, since it installs and runs without any additional configuration and is nicely integrated into the `tidyverse` environment that we cover in Chapter 7. As an example, let us use the data in the file `unsc-membership.xls`, which contains information on UN Security Council membership from Dreher et al. (2009). If you try to open this file with a regular text editor, you will see that it is a binary file, the contents of which are not human-readable. In R, we can open the file as follows:

```
library(readxl)
xls <- read_excel(file.path("ch04", "unsc-membership.xls"),
  sheet = 2,
  na = ".")
```

The R function that does all the work is `read_excel()`, and you need to specify the name of the input file (with a complete path if necessary), as well as the number or the name of the sheet you are importing. In our case, this is the second sheet, since the first one only contains metadata

about the dataset. We also set the `na` option to `."`, so that during the import process, fields that contain this character are interpreted as missing values. Writing data to an Excel file is equally simple with the `openxlsx` package:

```
library(openxlsx)
write.xlsx(xls, file.path("ch04", "output.xlsx"))
```

According to my experience, however, importing data directly from a spreadsheet can be tricky. As we discuss in more detail in the next chapter, the problem is that spreadsheets do not impose a strict tabular structure, while almost all statistical software tools do. This means that, for example, numeric columns can contain text, or data can even be placed in the spreadsheet outside the area where the regular dataset is kept. This is why you often encounter problems and errors during the import process. As the next chapter will make clear, I generally recommend against using MS Excel (or any other spreadsheet software) for data management, if you can avoid it. However, since many datasets are still distributed in spreadsheet formats such as Excel or LibreOffice/OpenOffice, it is hard to avoid them completely. This is why we spend an entire chapter on MS Excel (see Chapter 5), where we cover the various issues that can arise when managing research data with spreadsheet software.

4.2.3 Stata

Stata is one of the major tools for statistical analysis in economics and political science. It uses its own binary `.dta` format for data storage. Unlike Excel files, Stata's data files only contain a single table. This mirrors Stata's workflow well. It allows users to keep only a single table at a time in their working environment, which serves as input to all the analyses and visualizations the user carries out – this is very different from R, where you can have several data frames in your workspace. `.dta` files contain variable names and data, but optionally also short labels for the variables in the dataset.

Due to the fact that each Stata data file only contains a single, rectangular table, its import and use in R is typically much less problematic compared to spreadsheet files. As an example, we use the data on the targets of terrorism compiled by Polo (2020). The import function is provided by the `haven` package, and is straightforward to use:

```
library(haven)
dta <- read_dta(file.path("ch04", "terrorism-targets.dta"))
```


When importing Stata files, the short variable labels are preserved and shown when you click on the data frame in the *Environment* tab of RStudio. Alternatively, you can display, set, or change the variable labels with the `labelled` package, which can be very helpful when inspecting a data frame for the first time:

```
library(labelled)
var_label(dta$attacksum)
[1] "Number of attacks"
```

Similar to reading Stata files, `haven` can also write R data frames in the `.dta` file format:

```
write_dta(dta, file.path("ch04", "terrorism.dta"))
```

Stata has introduced several versions of its file format over time. `haven` can read and write all versions that Stata has used so far, although you may have to set the `version` parameter manually (the version refers to the Stata version used to create the file). One important fact to keep in mind, however, is that before version 14, Stata did not use a fixed string encoding, which means that you can run into the encoding problems we discussed above. Since Stata 14, text is saved in UTF-8 format and can therefore contain characters in any language and a wide variety of other symbols. Stata as well as SPSS (below) also differ from R in how they handle labeled data and missing values. It is beyond the scope of this book to discuss these differences in detail, especially because the `haven` documentation is very detailed in explaining these issues.

4.2.4 SPSS

SPSS (now called *IBM SPSS Statistics*) is another commercial software package that is frequently used in the social sciences. Similar to Stata, SPSS also comes with its own file format for data files, identified by the `.sav` file extension. These files are also binary, which means that they cannot be opened with a text editor and inspected manually. `.sav` files also contain only one table or list, with variable names, the data and (optionally) labels and documentation for the data. As an example for a dataset in SPSS format, we use the 2012–2016 version of the Worlds of Journalism Study (WJS, 2019), which assesses the state of journalism around the world. We again import the data with the `haven` package and summarize the first three columns:

```
library(haven)
sav <- read_sav(file.path("ch04", "journalism.sav"))
```

The original WJS data are based on interviews with journalists in different countries and cover topics such as editorial independence or how journalists see their role. The data we have here are national-level aggregates over all respondents. The WJS SPSS file also allows variables to be labeled, which we can use to find out what the variable names really mean:

```
library(labelled)
var_label(sav$C9)
[1] "Editorial autonomy: selecting stories (means)"
var_label(sav$C10)
[1] "Editorial autonomy: aspects emphasized (means)"
```

Similar to Stata files, the haven package can also write R data frames to SPSS files:

```
write_sav(sav, file.path("ch04", "wjs.sav"))
```

There are various other conventions and potential pitfalls when working with Stata and SPSS files. If you need more information on this, the documentation of the haven package is a good place to start.

4.2.5 R Data

R cannot only read and write files in other formats, but has its own file formats for preserving data. There is an important difference between the file formats I described above and R data files. While the above formats were all designed to store tabular data, R's data files can be used to store *any* kind of R object. So, for example, if you have a single vector, a list, or a data frame, they could all be permanently stored on disk using R's own file formats.

There are two types of R data formats: R data files, which have the extension `.RData` or `.rda`, and “serialized” R data files with the extension `.rds`. Both file types store the data in binary format (the default behavior). The difference is that `.rds` files save only a single object, and *without the name* the object was previously given. This means that when you load the object again from the file, you need to assign a new name. This is different for `.rda` files: A single file can contain *several* R objects, and will save each of them *with its name*. So when you load your data again, each

object (data frame, list, vector, etc.) will be available in your workspace under the name it was given previously.

To demonstrate this, let us use the replication data for Barberá (2015), which analyzes Twitter behavior of world leaders. In the replication data you will find a file called `leaders-twitter.RData`. This file contains data derived from Twitter accounts of political actors in six countries (US, Spain, Netherlands, UK, Italy, and Germany). The data set also indicates the party to which each actor belongs (if applicable). In our example, we first clear all objects in our workspace with `rm()`, import the data with `load()` and then show the objects in the workspace with `ls()`:

```
rm(list = ls())
load(file.path("ch04", "leaders-twitter.Rdata"))
ls()

[1] "elites.data"
```

As you can see, you now have an object called `elites.data` in your environment, even though we did not specify a name. Rather, `elites.data` was created by the author of the dataset, and then saved to the file. What type of object is this? A data frame? Let us check:

```
class(elites.data)

[1] "list"

summary(elites.data)
```

	Length	Class	Mode
US	23	data.frame	list
UK	4	data.frame	list
spain	4	data.frame	list
NL	4	data.frame	list
germany	4	data.frame	list
italy	4	data.frame	list

`elites.data` is a list with six entries, each of which is a data frame. This means that we have one table for each country, which can be accessed using the country name (e.g., `elites.data$germany`). For a simple demonstration of how to save `.RData` files, let's extract the data for Germany and Italy as two separate objects, and `save()` them:

```
elites_germany <- elites.data$germany
elites_italy <- elites.data$italy
save(elites_germany, elites_italy, file = "elites-germany-italy.rda")
```

Loading this file (after wiping our environment with `rm()`) makes the two objects available again:

```
rm(list = ls())
load("elites-germany-italy.rda")
ls()
[1] "elites_germany" "elites_italy"
```

Finally, let us do a quick comparison with the `.rds` format. Remember that we can only save one object at a time; in our case, we use `elites_italy`. We save this object using `saveRDS()`, and load it again with `readRDS()` under a different name:

```
saveRDS(elites_italy, "elites-italy.rds")
italy <- readRDS("elites-italy.rds")
```

This example shows that the original (`elites_italy`) and the newly loaded (`italy`) datasets can exist in the same workspace, but with different names (since we can adjust this during the loading). A simple check reveals that they contain identical data:

```
identical(elites_italy, italy)
[1] TRUE
```

Due to their ability to store any kind of R objects, R data files are extremely flexible, as long as you do not want to exchange data with other software tools. Both data formats can only be processed with R, and users of other statistical packages will *not* be able to use your data. Using our above example, you are now able to import `.RData` and `.rds` files from other sources; however, you should think about whether distributing your own data in one of these formats is a good idea. In particular when dealing with tabular data, I rather recommend a text-based CSV format, which most statistical packages and programming languages are able to read.

4.3 TRANSPARENT AND EFFICIENT USE OF FILES

Over the course of your research projects you are likely to accumulate a large number of data files: data from different sources and data you create yourself, using different naming schemes and file types. While you are working on a project, it is often possible to keep track of what these files contain, where they come from, and what you need them for. But experience shows that once you take a break from a project, it can be difficult to make sense of the different files in your project. I therefore provide you with some simple guidance on how to effectively organize

your research projects to minimize headaches and make your life easier. Many of these suggestions come from Jennifer Bryan's (2015) excellent talk on the matter, combined with my own experience.

4.3.1 Directory Structure

Good file organization starts at the directory structure of your project, that is, the folders in which your files are stored. In particular if your project involves many data files, I recommend that you create three sub-folders in your project folder:

1. `/raw` contains all the raw data you collected yourself or that comes from other sources. You should consider this folder *read only!* This is uncleaned data that your R scripts should never change, only read.
2. `/analysis` This folder contains the output of all your data cleaning and processing, ready for analysis and structured however is best for your project. If you remember our recommended workflow from Chapter 1, this folder contains the analysis datasets. Importantly, you should consider the contents of this folder as *transitory*, and there should never be any data in this folder that cannot be recreated by running your scripts again. You should be able to delete *everything in this folder* and still arrive at the same data (and analysis results) after re-running your scripts that process the raw data.
3. `/replication` This folder should be populated at the end of your project with all the data necessary to replicate your results. It should contain only properly anonymized, cleaned data that is ready to be shared with others.

Usually, your R scripts will be located in the main working directory. To easily see what each R script does, consider using informative and consistent file names.

4.3.2 File Names

Once your directory structure is set up, you should also consider sticking to some conventions regarding the names of the files you use. While files in the `raw` folder should not be changed after you download them, it is up to you to give useful names to all the ones you create. Jennifer Bryan (2015) gives three principles for naming files that you should stick to, a

recommendation I fully support: file names should be *machine readable*, *human readable* and *play well with the default ordering* of files on computers. What does this mean in practice?

1. *Machine-readable file names*: The great benefit of machine readable file names is that they make your life much easier when you process files automatically. Also, you can computationally extract information from the names that would be cumbersome to store and retrieve otherwise. To make files machine readable, avoid spaces, punctuation, and non-ASCII characters in your filenames and make sure you avoid case sensitivity (you should not have two different files called `myData.csv` and `mydata.csv`). Sticking to these rules makes using the default search function of your computer much more powerful, but also to retrieve and process the files using your script.
2. *Human readable file names* means that you should be able to tell from the name of a file what is in it. Giving your files names such as `01_clean-data.R` is vastly superior to just calling the file `01.R` or `data1.R`. By using delimiters such as the underscore and the hyphen consistently, you can also encode metadata about your file in the filename. For example, you can encode the order in which to run the files by starting with a numeral, and what the files do. Use underscores to separate these elements of metadata in your files and hyphens to separate words within the meta data.
3. *File names that sort well*: Starting your file names with a numeral allows for proper ordering when shown on your computer. You should always left-pad your numbers with a leading zero (otherwise on many systems, `10_analysis.R` will be sorted before `1_analysis.R`). When you use dates in your filenames, they should follow the ISO 8601 format YYYY-MM-DD and preferably be put at the beginning. This results in proper chronological ordering and prevents confusion from the different ordering of days and months in Europe and North America. While it is tempting to insert dates into filenames to denote different versions of a file over time (such as `20190312_data.R` and `20190313_data.R`), this oftentimes results in large, confusing numbers of files. If you want to preserve earlier versions of your code, consider using a version control system, which is particularly useful when collaborating with others. In Chapter 14, we cover these systems briefly.

4.4 SUMMARY AND OUTLOOK

Most social science research data is contained and distributed in files, and there are many file formats that can be used for tabular data. In this chapter, I provided a general introduction to the most common file formats you are likely to encounter in your work. The most convenient and flexible way is to use simple text files for tabular data, as for example the CSV format. Reading and writing is possible with almost any software package, and we can check contents manually with a simple text editor. However, there is no established standardization for these files; important features such as the choice of the field separator or the inclusion of a header can vary, all of which requires some caution when working with CSV files.

There also exist a number of binary file formats for tabular data, most of which can be processed with R. Among the most frequently used ones are spreadsheet files, most importantly Excel. Stata and SPSS also have their proprietary data formats, designed to store individual research datasets along with some documentation (e.g., labels of variables and values). The haven package in R offers a lot of functionality to work with these files. R objects (which includes data frames) can also be stored as files in R's own formats. However, exchanging data with other software tools using these formats is impossible, which is why you should use these file formats only when you really need them (e.g., if your data does not follow a tabular format and therefore cannot be easily stored in a CSV).

There are lots of other file formats, many of which are also used for social science data. As a general overview of file formats, the comprehensive file format guide by the US Library of Congress (2019) may be helpful. In case you cannot open a file with the R libraries we used in this chapter, I recommend you take a look at the *rio* package, which is able to read a large number of file formats in the fastest and most efficient way possible. In sum, here is a list of recommendations based on the discussion in this chapter:

- *Understand the basics of file-based storage:* When working with data stored in files, it is important to understand how files work, irrespective of whether they contain research data or not. We discussed the important difference between text and binary files. For the former, you need to be aware of the fact that there are different encoding schemes for text, and choosing the wrong one can lead to strange characters and errors in your data. Luckily, Unicode has emerged as the standard on many operating systems, which means that conversion issues can largely be avoided, at least when working with more recent files.

- *Familiarize yourself with different file formats:* There are few established conventions when it comes to storing tabular data in files. This means that for quantitative social scientists, there is a need to be familiar with different file formats as well as their strengths and weaknesses. Datasets for social science projects are distributed in many different formats, and it is likely that you will encounter a rarely used, legacy format in your work. Using the concepts and tools introduced in this chapter, you should be able to work even with the more difficult ones.
- *Organize your directories and files consistently:* To make the organization of your data and code as transparent as possible, try to stick to a consistent naming of files and folders. This is not only useful for others as they replicate your work, but it also helps you when you return to your project after some time. File and directory names should clearly indicate their content, and they should be constructed in a consistent way, such that they can be processed both by humans and computers.
- *For your data, choose a simple, well-known file format:* When you think about how to store your own data, it is advisable to prefer generic, software-independent file formats. For example, (correctly formatted) CSV files can be imported by almost any type of statistical software. Since they are text files, they also permit inspection by humans. It is generally recommended to avoid proprietary file formats such as Excel or SPSS. This also applies to R's custom file formats (.RData and .rds), which other software cannot process.

Managing Data in Spreadsheets

The previous chapter started our discussion of file-based data storage with a presentation of several file formats used for tabular data. In this and the following chapters of the book, I present different software tools that are commonly used for file-based data management and processing. While our focus in this book is on the R statistical toolkit, we start with one of the most widely used type of data management software: spreadsheets. A spreadsheet is a big table where users enter data into cells and perform calculations with them. The most common spreadsheet software that probably all readers are familiar with is Microsoft's *Excel* (part of the Microsoft Office suite). Apple's *Numbers* is a similar package, available only for the macOS platform. There are also different free spreadsheet systems such as the *Calc* software that is available as part of OpenOffice or LibreOffice.

Spreadsheets are not especially designed for managing research data, but they can be used for this purpose. This, however, comes with significant limitations. Most of these limitations are due to the fact that spreadsheets were developed to facilitate data management and processing *by humans*, and not by computers. In other words, they support a workflow where humans enter information primarily for other humans to look at. This is why there are almost no constraints to ensure the consistency of a table, but also why spreadsheets have so many features to format data for visual consumption.

Therefore, as we will see below, spreadsheets do not help much to ensure that a table is correctly formatted, and that the content of the cells in a table conforms with the type of a variable. Also, working with spreadsheets means that you manage data through pointing and clicking

with your mouse and through manual editing, which makes it difficult, if not impossible, to replicate your revisions later. It is therefore not a surprise that things can go wrong when you rely on spreadsheet software for data management; the *Economist* even covered some of the most popular cases in a report on Excel errors in science (The Economist, 2016). I nevertheless include spreadsheets in this book, because they – perhaps unfortunately – still constitute one of the main ways in which scholars manage their data. It is my hope that this chapter can prevent you from committing some of the fundamental mistakes that can occur when using Excel or similar tools; we discuss some of them at the end of the chapter.

For the illustration below, we rely on Microsoft Excel Version 2019, the most frequently used spreadsheet software. This is the most recent version of the software for Mac users. While the basic functions we cover here are also available in earlier versions of Excel and in Excel for Windows, they are sometimes accessed under different names and with different menu entries. For the most important functions, I mention these differences in the text. Still, the screenshots presented below are based on Excel for Mac.

5.1 APPLICATION: SPATIAL INEQUALITY

Inequality is traditionally defined as an unequal distribution of income, wealth, or some other quantity across the individuals in a society. However, there are other types of inequality, for example, between different regions in a country. This “spatial” inequality is what we cover in the example for this chapter, by computing a national-level estimate of economic inequality across a country’s different locations. More precisely, we are interested in the extent to which regions in a country differ with respect to their economic performance.

Most economic indicators are provided at the level of individual countries – think of the gross domestic product (GDP) or national growth estimates. While important for comparisons between countries, these national indicators cannot capture variation *within* countries. This is why economists have started to systematically collect data on economic variables at the subnational level for large samples of countries. These data allows us to examine regional variation in economic outcomes, but at the same time compare these patterns across countries.

One of the first global datasets of this type is the G-Econ dataset (Nordhaus, 2006; Chen and Nordhaus, 2011). G-Econ divides up the

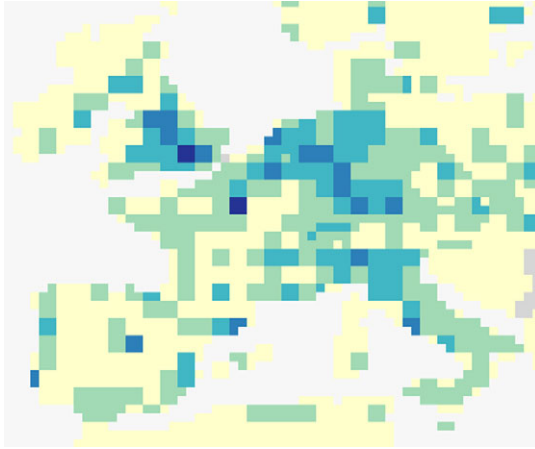


FIGURE 5.1. G-Econ data for Europe in 2005. Visualization created with the PRIO-Grid mapping tool at <https://grid.prio.org/>, see Tollefsen et al. (2012).

different countries into equal-sized, quadratic cells and reports economic indicators for each of them. The most important cell-based indicator is the Gross Cell Product (GCP), which is the equivalent of the GDP, but measured at the level of cells. It contains data for four different years (1990, 1995, 2000, and 2005). The map in Figure 5.1 shows a visualization of the data for Western Europe. We can clearly see the different cells, each colored according to its GCP value with economically wealthy areas displayed in darker shading. The plot shows the strong economic differences within Europe, with London and Paris being among the major economic hubs. Also, we see considerable variation within countries, for example, in Italy (the north vs. the south).

In this chapter, we use G-Econ to show how to perform simple data operations with spreadsheet software, in order to produce an estimate of spatial inequality for the countries contained in G-Econ. We use Version 4.0 of the data, released in May 2011. The online repository for this book contains a copy of the data, see the file `g-econ.xls`. The data comes in a single Excel file with two sheets (tables), shown as different tabs in Excel (see Figure 5.2).

One tab is called “definitions”, and it contains a list with the variables in the dataset and their descriptions. The other is called “GEcon40” and contains the actual data, in a large table with 27,445 rows. The first row in the table contains the column names, each corresponding to one entry in the documentation. Take some time to familiarize yourself with the data.

27442	5101.825	Zimbabwe	390.3988	390.3988	390.3
27443	71.39693	Zimbabwe	707.9332	707.9332	707.9
27444	3474.651	Zimbabwe	665.1368	665.1368	665.1
27445	1844.421	Zimbabwe	619.0456	619.0456	619.0
27446	11.89949	Zimbabwe	536.7775	536.7775	536.7
27447					
27448					

FIGURE 5.2. The different sheets in G-Econ’s Excel file.

In particular, see if you can make sense of the variables and the values contained in the dataset. There are a few things to note:

- Some cells do not seem to belong to actual countries. One of the entities in the data is “Antarctica,” which consists of many cells without economic activity (hence the missing values, coded as #N/A). Other small entities are also listed in the dataset (e.g., Svalbard or other small islands). This is not a problem in itself, but we need to keep this in mind when we assign cells to countries.
- For some variables in the data, the measurement units do not seem to be correct. For example, the column DIS_OCEAN is supposed to contain distances to the nearest (ice-free) ocean in kilometers and has many values in excess of 100,000. This is not really plausible, so this is likely a scaling issue and the values are probably given in meters rather than kilometers.
- Variables D1 and D2 are supposed to contain the same information (distance to the coast) according to the codebook, but in the actual data there are some rows where the values for these variables actually differ. Thus, there must be different ways in which the distance to the coast was computed.
- Did you spot how the most important variables in the data, the gross cell products, are encoded in the data? They are stored in the MER* and PPP* columns. Recall that this table design is not optimal, as we discussed in Chapter 3. Data structure and content are not independent, since we need to add additional columns to the table if we want to add more years to our database.

How do we work with spreadsheets in practice? In the following, we go through some basic steps to familiarize ourselves with spreadsheets, and to compute national-level estimates of spatial inequality. Since most readers will be familiar with spreadsheet software, much of this should be easy and straightforward.

5.2 SPREADSHEET TABLES AND (THE LACK OF) STRUCTURE

Spreadsheets do not care about data structure: Columns do not have clearly defined types (such as text or numeric columns). Rather, you can insert any value into any cell of a spreadsheet. This is why there is no real data definition you have to do before you can work with a table; if you need a new table, all you have to do is open a new Excel workbook (a collection of tables), or add a new empty sheet to an existing workbook (using the + sign next to the different tabs, see Figure 5.2).

In a spreadsheet, each table always has the same rectangular structure, where columns are labeled with capital letters and rows with numbers. This is problematic, since for a social science data table we usually want to name variables (i.e., columns) ourselves. Therefore, in a spreadsheet, we usually define columns by inserting their names in the first row, as is done in the G-Econ dataset. However, this is a *convention* rather than a requirement of the software – for example, nothing prevents you from adding new data *above* the row with the column names, which would break the structure. You may have noticed that in the G-Econ table, the first row appears to be fixed – it does not move when you scroll up and down in the table. While this does not mean that Excel treats the column names in any other way, it is a simple display adjustment to always keep the header names visible. This is called “freezing” the display of a part of the dataset. You can enable this by selecting the entire line below the row you want to freeze, and then clicking on **Window >> (Un)freeze Panes**.

Also, Excel does not really care about the type of data that goes into particular columns. All you can do is change the formatting of the cells. Simply select the entire column by clicking on the column header (e.g., column G for the DIS_LAKE variable in G-Econ) and click on **Format >> Cells**. This brings up the dialogue box in Figure 5.3.

In the list, you can choose different formats for the cells – since you selected an entire column, any settings you change here apply to the entire column. Now select “Number” and tick the box to use the 1000 separator (a comma). This will change the display of the different values in the column, but it does not define a fixed type for the column such that it stores, for example, only text or only numbers. Rather, we can still mix numbers and text in that column, as we do, for example, for the variable name in row 1 (text) and the data values in the remaining columns (numbers). Defining a format for a set of cells does not change their internal values, it only affects how they are displayed on the screen. In the above example, Excel still keeps simple numbers in the cells, but

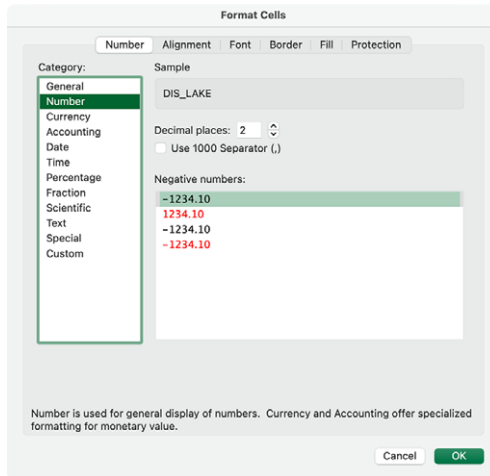


FIGURE 5.3. Excel's cell formatting dialogue box.

adjusts their display such that, for instance, the value 156602 is shown as 156,602.00. This may sometimes be confusing, since *what we see is not exactly what Excel stores internally*. When you select a cell, you can always view its true, unformatted content in the formula bar at the top, below the toolbars.

In Excel, each workbook consists of one or more sheets (tables) and it is stored in a single file. These files use the extension .xlsx, or .xls for the older legacy Excel file format (which is what the G-Econ database does). Any modifications you make to the data or the appearance of the table must be saved to the file. This is the same for all the common spreadsheet tools, where the data content and the presentation of the table are kept in the same file. None of these tools allow us to define strict types for the columns in a table, which would avoid coding mistakes and erroneous input.

5.3 RETRIEVING DATA FROM A TABLE

In many cases, you need to retrieve subsets of your table: For example, you may be interested only in selected variables from the G-Econ dataset, which are required for an analysis you want to conduct. As many data operations in spreadsheets, this is a manual operation, where you select the columns you need by clicking on their header (the capital letters), and then copy-paste them to a new sheet. You can select multiple columns at

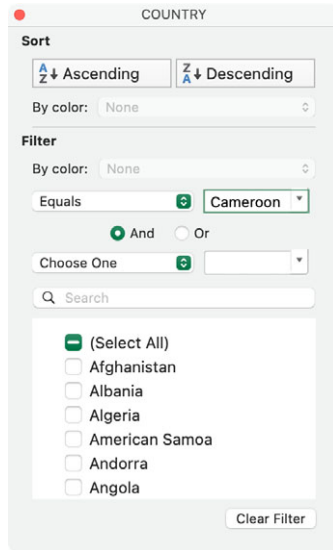



FIGURE 5.4. Excel's sort/filter dialogue box.

the same time – if they are not adjacent, you can hold down **command** (Windows: **Ctrl**) while clicking to select them. The retrieval of particular rows works in a similar way, by clicking (while pressing the Command/Ctrl keys if necessary) on the row numbers on the left. If you want to extract a set of rows with a particular value, this can be done by using one of Excel's filtering features. For example, let us extract all the cell values for Cameroon from G-Econ: Select the COUNTRY column, go to the **Data** tab in the menu, and select **Filter**. This displays a small arrow symbol next to the column name. Click on it, and you will see the dialogue box shown in Figure 5.4.

This allows you to sort and filter your data. For example, to extract the entries for Cameroon in G-Econ, choose the operator **Equals** in the Filter section, and make sure that only **Cameroon** is selected in the input field. This will display all rows for Cameroon from the data – the original row numbers remain the same, but are shown in blue to indicate that some rows are not displayed. As you can see, you can specify other filtering conditions, such as values that **Begin with** a particular sequence of characters. If you apply this filtering mechanism to a column with numbers, you will be given different selection operators, such as **Greater than** or **Less than**. You can also link different conditions to each other using logical operators such as **AND** or **OR**. Once you have

filtered the rows that you are interested in, you can select and copy-paste them to a different sheet for further processing.

Excel also allows you to sort the data in your spreadsheet, using again the dialogue box in Figure 5.4. If you select the COUNTRY column, bring up the Sort/Filter dialogue () and click on Ascending (Windows: A to Z), Excel will first ask you about the extent of the data that should be sorted. Here, you need to select *Expand the selection*, or Excel will *only sort values in the COUNTRY column*, which breaks the entire logic of a table because the values in the sorted column are assigned to different rows. While filtering is temporary in the sense that it displays a subset of the table but leaves the underlying data unchanged, sorting permanently changes the order of the rows in the table. Also, it is useful to note that Excel implicitly assumes that the first line of the table (the one containing the variable names) is static and therefore does not include it in the sorting. Importantly, again, this is an *assumption* the software makes, since there is no mechanism in Excel that lets you assign fixed column names that the software then works with.

5.4 CHANGING TABLE STRUCTURE AND CONTENT

Due to the absence of a fixed data structure that is maintained by the software, changes to the logical design of the table and its content are easy: You can access new columns simply by using some of the empty ones in your spreadsheet, or insert them by right-clicking on the header of a column. While there is an upper limit to the number of columns in any sheet (16,384), this is unlikely to matter in practice since tables of this size are impossible to navigate by a user. For each new column you insert, you can follow the convention to specify the variable name at the top, although, again, this is something that the software does not require. As for all the others, there is no preset type for the new column, but you can change the display of the values by formatting cells as we have shown above.

Changing the actual data in a spreadsheet is also done in a similar way, simply by manually editing the content of the cells in your table. Alternatively, similar to a word processor, you can use the standard search and replace feature to do this for multiple values. For example, take a look at the PRECAVNEW80_08 column in the G-Econ table. There are many erroneous entries in this column with the value #DIV/0!. If you want to work with the dataset for your analysis, these values can cause problems

down the road: Obviously, we do not really know what to do with these values, and the occurrence of text in a column that is supposed to be numeric can interfere with mathematical operations you perform with it. Select the `PRECAVNEW80_08` column by clicking on its header, and go `Edit >> Find >> Replace` (Windows: `Home >> Find & Select >> Replace`). In the dialogue box that shows up, enter the value we are looking for (`#DIV/0!`) and the value we want to replace it with. In this case, it is probably safest to simply remove the strange values, which we can do by leaving the “Replace with” field blank. If you then click “Replace all,” the erroneous values in the column will disappear.

The above operations for changing your data structure and updating the data values in your table show the data workflow that is common to all spreadsheet tools. Most of the data work consists of manual operations: navigating to a particular cell of a table and typing in a certain value. This workflow may sometimes be convenient, for example, when we create a new dataset that requires human coding and manual input. But even for these tasks, the lenient approach of spreadsheets when it comes to data types and structure can cause problems, since the software does not automatically recognize mistakes (e.g., when entering text for a variable with only numeric values). Thus, the software lets you do almost anything with your data – this freedom is likely something you pay for with inaccuracies in your data and inefficiencies in your workflow. However, before we turn to the question of how you get your data out of a spreadsheet, let us first complete our description of the two remaining data operations and how they can be done in Excel.

5.5 AGGREGATING DATA FROM A TABLE

Excel has different features that allow you to aggregate your data. For our purpose of computing an indicator of spatial inequality, we present only one here: The Pivot table. A Pivot table is a tool to summarize data in a flexible way, for example, by allowing users to introduce different grouping dimensions over which the values in the data can be aggregated. This is what we need for our application. Let us start with a simple example that counts the number of cells per country in G-Econ. Before we proceed, you need to select the entire G-Econ table to make sure that all the data is included in the Pivot table. You can do this via the menu (`Edit >> Select All`) or by clicking the square box at the top left of the table. Now, you can access the `Summarize with PivotTable` feature via the `Data` menu (Windows:

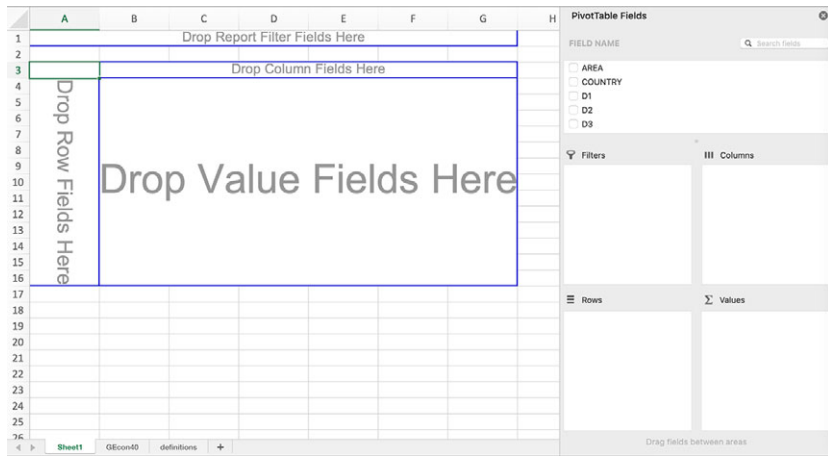


FIGURE 5.5. Setting up a Pivot table.

Insert > **Tables** > **PivotTable**). It starts with a simple dialogue box where you need to specify the subset of the data you want use as the basis for your Pivot table. In our case, this defaults to the part of the sheet that is filled with data, so there is nothing to change here. Also, we choose to have the new Pivot table placed in a new sheet as part of your Excel workbook.

This creates a new sheet with an empty roster for a Pivot table. The structure of such a table is simple: it consists of column and row fields, which are the main levels for grouping. The value field at the center of the table is the one that displays the summarized data. Figure 5.5 shows the basic configuration of a Pivot table.

The part on the right is where you set up the field(s) you want to use as grouping level(s) for your table. You can simply drag and drop field names from the box at the top into the empty boxes below. First, drag and drop the **COUNTRY** field into the “Rows” box. This adds all the different values for this field (the individual countries) as rows in your Pivot table. So far, however, we do not have any summary values we are computing for each of these countries. To set this up, drag the **PPP2005_40** field (the gross cell product computed using Purchasing Power Parity for the year 2005) from the list at the top and drop it into the “Values” box at the bottom right. This way, Excel computes aggregate (summary) statistics over the gross cell product for each of the grouping levels (currently, the countries on the left). The default summary statistics Excel computes is to *count* the number of observations for each country, which is why the entry in the “Values” box reads “Count of PPP2005_40.” For Afghanistan, for

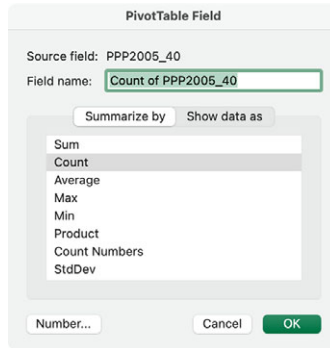


FIGURE 5.6. Changing the aggregation function for a Pivot table.

example, there are 92 cells in the dataset, while Albania is much smaller with only 9 cells.

Now, let us compute the *standard deviation* of all gross cell products for each country, as a measure of the spread of the gross cell product values and hence the spatial inequality. For this, we need to change the aggregation function; rather than counting the number of observations, we want to compute the standard deviation of all gross cell product values for 2005 for each country. You can change the aggregation function by clicking on the small “i” symbol next to the entry in the “Values” box (in Excel for Windows, this is a little arrow that brings up a drop-down list, where you can select “Field Settings”). This brings up a dialogue box, shown in Figure 5.6.

The count function (highlighted) is what we currently use for the Pivot table. If you change this to StdDev, the values in the Pivot table will reflect the standard deviation of the gross cell product values for each country. This turns out to give us many invalid values, such as the familiar #DIV/0! values (which indicate a division by zero), but also #N/A values, which are present in the original data sheet. Here, we see some of the problems resulting from Excel’s (and other spreadsheets’) sloppy use of data types. In the main data, we have many values in the (numeric) gross cell product variable that are not numbers but strings (text). Since we cannot compute a mathematical sum over text values, Excel simply outputs these values directly. We will later work with tools that require us to specify a fixed type for each column of a table and that allow for a consistent coding of missing values to properly exclude them from computations.

5.6 EXPORTING SPREADSHEET DATA

We now have the values of spatial inequality and could continue to analyze them in Excel. However, in order to show you a complete workflow and a nice R plot at the end of this chapter, we export the data to a text file (CSV) for further analysis in R. To create a CSV file from our Pivot table, we simply save the worksheet with the table in this format: **File** >> **Save As**. There are different CSV file formats available; I recommend that you choose the one with UTF-8 support. During the export process, Excel warns you that only the active sheet will be saved to CSV, since this file format cannot deal with multiple tables in one file – each CSV file is supposed to contain exactly one table. You can acknowledge this warning and proceed with “OK.” The second warning refers to the fact that many Excel features such as colored cells or formatted text are lost when saving the file as a CSV. This is why you should not keep the entire workbook in this format (the CSV file will be saved anyway).

5.7 RESULTS: SPATIAL INEQUALITY

If you take a look at the values we have computed in our Pivot table above, there are some that reflect our basic intuition of spatial inequality. You see that some of the inequality estimates are very large, for example, for South Korea (86.04) or the United Kingdom (67.63). These are countries that are very strong economically, but where economic activity is disproportionately concentrated in single economic centers such as Seoul or London. Although comparable in terms of overall economic performance, other countries such as Germany (41.93) have lower values of spatial inequality, because there are several economic hubs in the country.

A plot of the overall distribution of spatial inequality using the exported CSV file (see Figure 5.7) shows that a large number of countries have very low values of spatial inequality, or in other words, a relatively even geographic distribution of economic activity. This could be due to a number of reasons. For example, the size of the country could affect its spatial inequality; if a small country only consists of three cells, the spread of economic activity within that country will likely be limited. Another reason could be the level of economic development. Many countries have very low levels of economic activity throughout, which will also affect the scores we have computed. All this suggests that the way in which we compute spatial inequality may not be entirely satisfactory, and that other measures may be preferable.

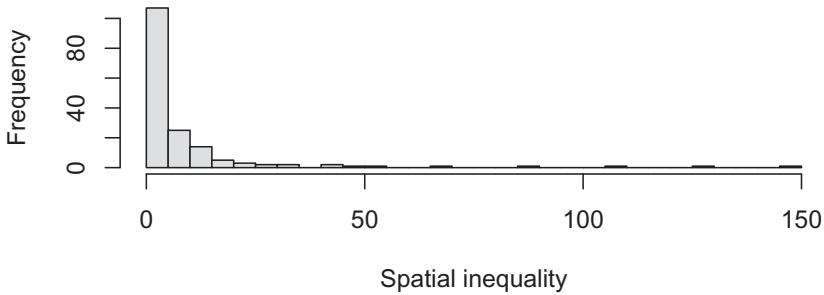


FIGURE 5.7. Histogram of spatial inequality scores.

5.8 SUMMARY AND OUTLOOK

Although we have only scratched the surface in terms of Excel's functionality, you should now have a good idea of how data processing with spreadsheets works. Spreadsheets leave you considerable freedom when it comes to the structure of your table(s), and the data you store in them. They also let you adjust the visual display of your data. For example, you can increase the font size for a particular cell, or change its background color. This is because spreadsheets are designed for a mix of different tasks: for data *storage* in a (loosely defined) tabular structure, but also for the visual *presentation* of your data in different fonts or colors. The interaction with spreadsheets works mostly by means of manual editing; for example, you navigate the spreadsheet using your mouse, and edit content, delete rows or columns, or execute other tasks such as sorting your data.

The features that spreadsheets offer may be suitable for certain tasks in the data collection process. For example, if you manually put together a human-coded dataset, spreadsheets can serve as useful and intuitive tools for coders to enter the coded information. However, for many (if not most) other tasks in social science data management and processing, spreadsheets are not a good choice: The lack of a pre-defined structure of your table makes it difficult to spot and fix errors in your data, such as wrong labels, inconsistently coded missing values, or text in numeric columns. Also, the fact that you work with spreadsheets through manual interactions makes it difficult to replicate the data processing steps you completed. There is no record of what you did; if something went wrong, you likely have to repeat most steps yourself. If others want to replicate your work, it would be hardly possible.

While in general I explicitly recommend against the use of spreadsheet software in data management, there are certain applications for which it can be useful, for example, for the manual creation of a new dataset. Also, several datasets in the social sciences are still distributed in Excel format, which is why it is difficult to avoid this software completely. If you need to manage and process your data with Excel, here is some advice for avoiding major problems down the road. The suggestions below apply in particular to a data processing workflow where the data is eventually exported from Excel to be analyzed in a statistical toolkit, such as R or Stata. If you use Excel solely to prepare your data for humans to look at, most of them do not apply. Much of what I mention here overlaps with recommendations from other scholars, see, for example, Data Carpentry (2017).

- *Use one table per sheet:* In our example above, we worked with different sheets that are all part of the same Excel file (recall that you can switch between them using the tabs at the bottom). I strongly recommend that you store at most one table per sheet, not multiple ones. This makes it easier to maintain a consistent type for a given column. Also, exporting the data becomes much easier, since you can select the entire sheet – and not just a subset of it – and save it to a separate file.
- *Stick to the rectangular table format:* Within any given sheet, strictly keep the rectangular table structure intact. That is, do not let sub-headers or any other layout elements interrupt the table structure. Also, do not use the “merge cells” feature, which allows you to combine adjacent cells into bigger ones, as this also breaks the structure of the table and creates major problems when exporting your data. The same applies to comments you place somewhere outside the main table. If you want to record and preserve comments in a spreadsheet, create a separate column and place your annotations there. It also makes your life easier if you do *not* leave a margin of empty cells around your main table, as is often done for visual purposes. The top-left cell (A1) is where your main data should start.
- *Use proper variable names:* A correctly formatted table requires that columns be named. In Excel, you do this by placing the variable names in the first row. You can use any text as variable names, but most other software packages are more restrictive here. This is why you should not use white spaces, special characters, or mathematical symbols in variable names. Also, numeric characters at the beginning of variable

names are usually not a good idea (and not permitted in some tools, for example in R). If you require variable names with multiple words, you can separate them with an underscore (as in `new_variable`). Since you may later be using these variable names in statistical code, it is useful to keep them relatively short.

- *Make sure that data values are valid:* Since spreadsheet software such as Excel lets you enter almost anything into the cells of a table, it is up to you to make sure that the data in a particular column complies with the type of that column. For example, a column recording the time of an observation (the year) should only have values such as 1816 or 1946, but not 1990s or 2001 (and 2002). The former are purely numeric values, the latter are not. Non-numeric values can be recognized in Excel due to the fact that they are left-aligned in the cell, while true numbers are right-aligned. Alternatively, you can use one of Excel's functions such as `ISNUMBER()` to test whether a content of a cell is a number, and similar functions also exist for other types. It is also important to use consistent coding for missing values; I recommend to leave these cells empty.
- *Do not use formatting elements to store information:* A frequent mistake when using Excel is to use formatting elements such as cell coloring, font styles, or others for storing information. For example, if you create a country-level dataset with information about whether a country is democratic or autocratic, it is *not* a good idea to color democracies in green and autocracies in red. This information can be used by humans only, and it is lost when you export the data in any format other than Excel. Therefore, it is best not to care about font styles, cell backgrounds, etc., but stick to the defaults set by the spreadsheet software.

Basic Data Management in R

As we saw in the previous chapter, using spreadsheets to prepare data for analysis may be convenient at first, but entails a number of major drawbacks. In this chapter, we introduce the basics of data management using the R statistical toolkit. R is one of the most popular software tools for data analysis – it has numerous features and extension packages for statistics, visualization, machine learning, etc. Therefore, it is convenient to also use it to prepare your data *before* you actually analyze it. This way, you can stick to a single software package and one language to implement your entire research workflow from beginning to end.

As you know, you interact with R not by pointing and clicking with your mouse, but by entering commands in the R programming language. This way, you can have R run statistical analyses for you, visualize your data, but also perform data management operations. While cumbersome at first, this mode of interaction is extremely powerful and has a number of advantages. Most importantly, the set of commands you send to R (which is typically called a “script”) can be saved, such that you can later return to it, fix potential problems, or simply replicate the steps you carried out to arrive at a particular result. This resolves one of the main drawbacks in the spreadsheet-based data management approach we discussed in the previous chapter, where it is difficult – if not impossible – to keep track of the different modifications you made to your data.

In this chapter, we focus on “base R,” which is the set of commands and functions that are part of R’s core functionality. We do this with a particular emphasis on R’s features for data storage and processing, and how we can get data into R and back out. In the next chapter, I describe

the *tidyverse*, an R extension that follows a different approach for data processing. While many consider the *tidyverse* as superior, I believe that it is still necessary to be familiar with R's basic functions and syntax for data management.

6.1 APPLICATION: INEQUALITY AND ECONOMIC PERFORMANCE IN THE US

Inequality remains a global issue of major concern (Piketty, 2014). In the practical example for this chapter, we focus on the historic development of inequality in the US, which former president Barack Obama considered to be a “defining challenge of our time” (The White House, 2013). How does inequality during Obama’s presidency compare to other presidents? To what extent does inequality depend on the size of the US economy overall?

To find out, we use data from different sources. Data on inequality comes from the World Inequality Database (WID, 2020). The dataset contains time series for several measures related to inequality for many countries, and therefore allows for systematic, historical research into the determinants and consequences of inequality. In our example, we use one of the many indicators for income inequality: the share of the pre-tax income received by the top 10% of all individuals with the highest income in a country. Higher values of this measure indicate higher levels of inequality. The WID has a powerful web interface at <https://wid.world/data/>, where users can select the indicators, the countries and the years of observation they are interested in. The data file in the repository, however, was created using the bulk download function for the entire database, selecting the US and only the variable we are interested in. The resulting table was saved as a CSV file, which you can find in the data repository for this chapter in the file `us-inequality.csv`.

Data on US economic performance can be obtained from the FRED data portal of the US Federal Reserve Bank St. Louis (2020). The *real gross domestic product per capita series* was selected and downloaded in CSV format. The dataset is available in the data repository for this book in the file `us-gdp-pc.csv`. In addition, we combine the inequality estimates from the WID and the GDP data with data on US presidents, available online from the US Library of Congress (2020). For your convenience, the latter data is available in a shortened version in the file `us-presidents.csv` in the data repository.

6.2 LOADING THE DATA

We start by importing the three data files into R. Let us take a closer look at the inequality data from the WID first. If you open this file in RStudio's editor, you will see the following first three lines:

```
country,variable,percentile,year,value,age,pop
US,sptincj992,p90p100,1913,0.4231,992,j
US,sptincj992,p90p100,1914,0.4295,992,j
```

The structure of this file is straightforward; the first line contains the variable names, and the data start in the second row. A comma is used to separate the different fields in a row.¹ In our data, `variable` refers to the particular variable we are using from the WID, in our case the share (thus “s”) of the pre-tax income (“ptinc”). `percentile` specifies the percentile range of the distribution we are looking at: `p90p100` is the range between the 90th and the 100th percentile, and thus corresponds to the top 10% of earners. We can use R's standard functions to read the data from the CSV file:

```
wid <- read.csv(file.path("ch06", "us-inequality.csv"))
```

A quick summary of the data shows that the import worked correctly:

```
summary(wid)
  country      variable      percentile      year
Length:100    Length:100    Length:100    Min.   :1913
Class :character Class :character Class :character 1st Qu.:1938
Mode  :character Mode  :character Mode  :character Median :1963
                                     Mean  :1963
                                     3rd Qu.:1989
                                     Max.   :2014

  value      age      pop
Min.   :0.3384 Min.   :992 Length:100
1st Qu.:0.3604 1st Qu.:992 Class :character
Median :0.4026 Median :992 Mode  :character
Mean   :0.4071 Mean   :992
3rd Qu.:0.4536 3rd Qu.:992
Max.   :0.4803 Max.   :992
```

One issue we should fix is the presence of unnecessary data in our data frame. Many of the variables such as `country` or `variable` are constant, and are only included because our data is a subset of the entire WID

¹ If you choose to download a custom-defined data file from the WID yourself, the import is not straightforward, since these files contain a header that does not conform to a regular CSV format.

(which contains many more countries and variables). We therefore retain only the data in columns 4 and 5 (year and value), and give them more meaningful names:

```
wid <- wid[4:5]
colnames(wid) <- c("year", "p90p100")
summary(wid)
```

	year	p90p100
Min.	:1913	Min. :0.3384
1st Qu.	:1938	1st Qu.:0.3604
Median	:1963	Median :0.4026
Mean	:1963	Mean :0.4071
3rd Qu.	:1989	3rd Qu.:0.4536
Max.	:2014	Max. :0.4803

Next, we need to import the GDP per capita estimates. The CSV data file is formatted according to standard conventions, using a comma as field separator and a header with the column names, which is why the import is straightforward. However, we again adjust the column names to something meaningful and change the type of the first column such that it properly reflects the dates:

```
gdp <- read.csv(file.path("ch06", "us-gdp-pc.csv"))
colnames(gdp) <- c("date", "gdppc")
gdp$date <- as.Date(gdp$date)
summary(gdp)
```

	date	gdppc
Min.	:1947-01-01	Min. :13999
1st Qu.	:1965-03-09	1st Qu.:21153
Median	:1983-05-16	Median :30482
Mean	:1983-05-17	Mean :33533
3rd Qu.	:2001-07-24	3rd Qu.:46691
Max.	:2019-10-01	Max. :58392

As you can see, the `gdp` data frame contains quarterly estimates of GDP per capita. We only need one estimate per year, which is why we retain only the observations for July:

```
gdp <- subset(gdp, as.numeric(format(date, "%m")) == 7)
```

Finally, let us import the dataset with the US presidents. This dataset was exported from a spreadsheet, which is why a semicolon is used as a field separator. This requires us to set the `sep` parameter of the `read.csv()` function accordingly. R again obtains the column names from the first line in the file. It replaces the whitespaces in the names with dots, since R does

not accept column names with spaces. We again set the column names to lower case and retain only those columns we need later.

```
presidents <- read.csv(file.path("ch06", "us-presidents.csv"), sep = ";")
colnames(presidents) <- tolower(colnames(presidents))
presidents <- subset(presidents, select = c(inoffice, president))
summary(presidents)
```

inoffice	president
Length:15	Length:15
Class :character	Class :character
Mode :character	Mode :character

All three datasets – `wid`, `gdp`, and `presidents` – are data frames, which, as you know, is the main data structure for tables in R. In RStudio, you can view data frames just like spreadsheet tables using the `View()` command or by clicking on the data frame in the “Environment” tab in the top right panel. Note that, unlike in a spreadsheet program, you cannot manually edit the data – this would have to be done using R commands. As we have discussed above, there is no fixed standard for storing data in text files (CSV and similar formats). This is why you need to be careful when importing these data and make sure that the import was successful. Above, we checked some of our imported datasets simply by printing a summary. Another way to achieve this is the `str()` function:

```
str(wid)
```

```
'data.frame': 100 obs. of 2 variables:
 $ year : int 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 ...
 $ p90p100: num 0.423 0.429 0.422 0.444 0.449 ...
```

In particular, this allows you to check:

- Whether all rows have been imported. If you load the CSV file in a text editor, you can easily count the rows in the original file. This file typically has one more row than the data frame in R (the header in the first line). Some CSV files have empty lines at the end; these can be excluded from the import using the `nrows` parameter in `read.csv()`, which restricts the number of rows to import.
- Whether all columns have been imported. Usually, inconsistencies in the number of cells between different lines will trigger errors and the file will not be read, but even if there are no error messages, it is still useful to check whether all columns were imported successfully and have the right names.

- Whether the columns have the right data type. The `str()` function shows us the type of data contained in the columns (e.g., `int` and `num` for the WID data frame). Since standard CSV files explicitly specify only the names – but not the types – of the table columns, the default behavior in R is to *infer* the variable types from the data it encounters in the respective columns. That is, if a column contains only numeric values and properly coded missing values – as is the case for the `p90p100` column in the `wid` data frame – R will correctly use a numeric type for it. If, however, we were to denote missing values with the string `n.a.` in the original CSV dataset, R would convert the entire column to a character variable, and you could not use it for any type of analysis that requires numeric input.

In general, if you encounter text files that deviate from common standards and cause issues during the import, I recommend that you try to address these problems using R code rather than fixing the data file manually. For example, you can skip a given number of lines at the beginning of a CSV file with `read.csv()`'s `skip` parameter, which is useful for some CSV files that have a header of more than a single line. Fixing import issues using R's functions rather than manually editing the files has a number of advantages. You could easily replace the old version of the data file with a newly downloaded one, for example, if a new version of the data has been released. Also, you avoid making undocumented modifications to a raw data file, which is something I recommended against at the beginning of the book.

6.3 MERGING TABLES

For comparing inequality and economic performance in the US over time, it is convenient to merge the two tables with each other. Both contain annual observations, so this is straightforward. However, before we can do this, we need to make sure that both tables have columns we can use to join them. The `wid` data frame already has a column containing the year of the observation; for the `gdp` data frame, we still need to create such a column by extracting the year from the date column:

```
gdp$year <- as.numeric(format(gdp$date, "%Y"))
```

We use again the `format()` function for this and convert the result to a number. Now, we are ready to merge the WID and GDP tables and store the result in a new data frame:

```
data_annual <- merge(wid, gdp, by = "year")
summary(data_annual)
```

	year	p90p100	date	gdppc
Min.	:1947	Min. :0.3384	Min. :1947-07-01	Min. :14008
1st Qu.:	1964	1st Qu.:0.3549	1st Qu.:1964-12-30	1st Qu.:21088
Median :	1982	Median :0.3703	Median :1981-12-30	Median :30232
Mean :	1981	Mean :0.3872	Mean :1981-06-30	Mean :32314
3rd Qu.:	1998	3rd Qu.:0.4254	3rd Qu.:1998-03-31	3rd Qu.:43412
Max. :	2014	Max. :0.4714	Max. :2014-07-01	Max. :53452

What does the `merge()` function do? It takes two data frames and joins them line by line, for all lines that have the same values in the year column. This is why our resulting data frame will have all the columns from the first and the second data frame combined, as well as the column(s) used for merging. In our case, the merge column has the same name in both datasets, but `merge()` can also deal with merge columns of different names (you would use the `by.x` and `by.y` parameters instead of `by`). The function can also deal with applications where you merge not just on a single column, but on multiple ones (e.g., if you merge annual observations for different countries).

Now, take a closer look at the number of observations in the original and the merged datasets:

```
nrow(gdp)
[1] 73
nrow(data_annual)
[1] 66
```

The merged data frame contains fewer observations. The reason is that our WID data do not start until 1962, while the GDP data are available from 1947 onwards. `merge()` retains only lines with at least one match in the other dataset, so we lose those observations from `gdp` that do not have a match in the WID. If we wanted to keep all observations from `gdp`, we could use the `all.y = T` parameter setting. However, in the merged table, the corresponding fields for the WID values would remain empty (`NA`).

As a final step, we need to merge the information about the US presidents to our `data_annual` dataset. However, the `presidents` data frame contains time periods, each with a start and an end year. This is why we cannot use it directly in the `merge()` function, because it requires a common attribute. Therefore, we need to make the time periods in the

presidents data frame (which is currently a character variable) compatible with the year variable (which is a number) in the WID. A first step for doing this is to extract the start year and the end year of each president and to add them to the data frame. You probably noticed that the periods given in the data are overlapping; for every presidency, the first year is the same as the last year of the previous one. To avoid confusion in our dataset with annual observations, we therefore reduce the end year given in the data by one, such that we have exactly one president per year:

```
presidents$startyear <- as.numeric(substr(presidents$inoffice, 1, 4))
presidents$endyear <- as.numeric(substr(presidents$inoffice, 6, 9)) - 1
```

Since the information in `startyear` and `endyear` is now redundant in the table, we can remove the old variable:

```
presidents$inoffice <- NULL
```

We now need to merge the two data frames based on the corresponding years; so for each entry in the `data_annual` data frame, we need to look up the corresponding president based on the start and the end year of his tenure. This would be simple if we had a dataset with annual observations of US presidents. We do not have this, so we need to use a simple trick. We first create all possible combinations of rows from `data_annual` and from `presidents`. The `merge()` function does this if we set the `all = T` parameter:

```
data_annual <- merge(data_annual, presidents, all = T)
data_annual[1:5, ]
```

	year	p90p100	date	gdppc	president	startyear	endyear
1	1947	0.3708	1947-07-01	14008	Harry S. Truman	1945	1948
2	1948	0.3891	1948-07-01	14515	Harry S. Truman	1945	1948
3	1949	0.3836	1949-07-01	14182	Harry S. Truman	1945	1948
4	1950	0.3899	1950-07-01	15388	Harry S. Truman	1945	1948
5	1951	0.3771	1951-07-01	16223	Harry S. Truman	1945	1948

The result of this operation is called the *Cartesian product* of the two tables. Obviously, it yields many useless combinations. For example, the first line contains the inequality and GDP values for 1962, combined with the information on President Truman, who was in office during 1945–1949 and 1949–1953, which makes little sense. This result is not surprising, since we specify no condition whatsoever about which rows are supposed to match. However, in an additional step, we can now use simple filtering to get rid of the lines with non-matching information.

Specifically, we retain those lines where the current year (indicated by the year variable) is larger than the first year of the respective president's term (as specified in the startyear column), and smaller or equal to the last year (as contained in the endyear column):

```
data_annual <- subset(data_annual, year >= startyear & year <= endyear)
data_annual[15:20, ]
```

	year	p90p100	date	gdppc	president	startyear	endyear
213	1961	0.3583	1961-07-01	18319	John F. Kennedy	1961	1962
214	1962	0.3609	1962-07-01	19126	John F. Kennedy	1961	1962
281	1964	0.3698	1964-07-01	20567	Lyndon B. Johnson	1963	1964
348	1966	0.3629	1966-07-01	22650	Lyndon B. Johnson	1965	1968
349	1967	0.3529	1967-07-01	23020	Lyndon B. Johnson	1965	1968
350	1968	0.3551	1968-07-01	24009	Lyndon B. Johnson	1965	1968

This gives us exactly what we want: a table with GDP and WID information, combined with information about the US president in office during the respective year. The above approach for merging tables by creating the Cartesian product and then retaining only the matching lines is a recipe you should remember for later parts of this book.

6.4 AGGREGATING DATA FROM A TABLE

We now have a complete data frame with all the data we need for our simple analysis. Before we present the final result of our analysis, let us take a look at how we aggregate the data in different ways to show descriptive statistics for the different presidencies. As we have already discussed in Chapter 3, “aggregate” statistics are computed over *groups* of rows. In our example, we may be interested in the average level of inequality and GDP for each president's term(s). This can be done using the `doBy` package, where we specify the variables to be aggregated as well as the grouping variable(s), as follows:

```
library(doBy)
summaryBy(p90p100 + gdppc ~ president, data = data_annual)
```

	president	p90p100.mean	gdppc.mean
1	Barack Obama	0.4608167	51305.17
2	Bill Clinton	0.4172000	42081.50
3	Dwight D. Eisenhower	0.3586750	17377.50
4	George Bush	0.3893000	37397.00
5	George W. Bush	0.4431250	49458.38
6	Gerald R. Ford	0.3424667	26642.67
7	Harry S. Truman	0.3792667	15103.67
8	Jimmy Carter	0.3463500	29470.75

9	John F. Kennedy	0.3596000	18722.50
10	Lyndon B. Johnson	0.3601750	22561.50
11	Richard M. Nixon	0.3439000	25195.60
12	Ronald Reagan	0.3642375	32733.25

This simple aggregation groups the rows in our data frame by presidents. For each group, it applies the `mean()` function to each of the specified variables, `p90p100` and `gdppc`. Note the naming of the aggregated columns: The default behavior of the `summaryBy()` function is that it uses the name of the original variable and appends the name of the function applied to it. For example, the `p90p100.mean` variable contains the averages of the `p90p100` values for each president.

Computing averages is the default, but we can also specify other aggregation functions. For example, we can count the number of years that the respective president was in office. To do this, we simply add the `length()` function as an additional one to be applied to each group of rows:

```
summaryBy(p90p100 + gdppc ~ president,
  data = data_annual,
  FUN = c(length, mean)
)
```

	president	p90p100.length	gdppc.length	p90p100.mean	gdppc.mean
1	Barack Obama	6	6	0.4608167	51305.17
2	Bill Clinton	8	8	0.4172000	42081.50
3	Dwight D. Eisenhower	8	8	0.3586750	17377.50
4	George Bush	4	4	0.3893000	37397.00
5	George W. Bush	8	8	0.4431250	49458.38
6	Gerald R. Ford	3	3	0.3424667	26642.67
7	Harry S. Truman	6	6	0.3792667	15103.67
8	Jimmy Carter	4	4	0.3463500	29470.75
9	John F. Kennedy	2	2	0.3596000	18722.50
10	Lyndon B. Johnson	4	4	0.3601750	22561.50
11	Richard M. Nixon	5	5	0.3439000	25195.60
12	Ronald Reagan	8	8	0.3642375	32733.25

The `summaryBy()` function returns the result of the aggregation as a new data frame. This is useful if we want to continue working with this result; for example, we may want to order the entries in the aggregation table temporally by the time of each president's term. We can do this by adding the year as an aggregation variable, and the minimum as an aggregation function. This way, for each president, we obtain the first year this president shows up in our dataset, and can use this for ordering our aggregated data frame. You will see that by adding more variables and aggregation functions, the result of the aggregation becomes rather

difficult to work with. This is because the `summaryBy()` function applies each of the aggregation functions to each of the aggregation variables, even though this is not what we want. This is why we simply drop the aggregated columns we do not need, to avoid confusion. In later chapters, I will present better ways for doing this.

```
data_term <- summaryBy(gdppc + p90p100 + year ~ president,
  data = data_annual,
  FUN = c(length, mean, min)
)
data_term <- subset(data_term,
  select = c(
    president,
    gdppc.mean,
    p90p100.mean,
    year.length,
    year.min
  )
)
data_term <- data_term[order(data_term$year.min), ]
print(data_term[1:3, ])
```

	president	gdppc.mean	p90p100.mean	year.length	year.min
7	Harry S. Truman	15103.67	0.3792667	6	1947
3	Dwight D. Eisenhower	17377.50	0.3586750	8	1953
9	John F. Kennedy	18722.50	0.3596000	2	1961

6.5 RESULTS: INEQUALITY AND ECONOMIC PERFORMANCE IN THE US

In the plot in Figure 6.1, we see the development of economic performance and inequality by presidency. Overall, economic performance has been steadily increasing over time in the US, and there are no particular differences observable by presidency. At the same time, inequality does not seem to be tracking this trend closely, until we get to Jimmy Carter's presidency in the late 1970s. This time is seen as the beginning of the American deindustrialization, where inequality and poverty rose due to the increased off-shoring of jobs primarily in the manufacturing sector (Strong, 2021). Since then, inequality in the US has been increasing steadily, until it reached a level that is about 50% higher as compared to the first time periods in our sample. During Barack Obama's presidency, according to our statistics, almost half of the pre-tax national income went to the top 10% of earners.

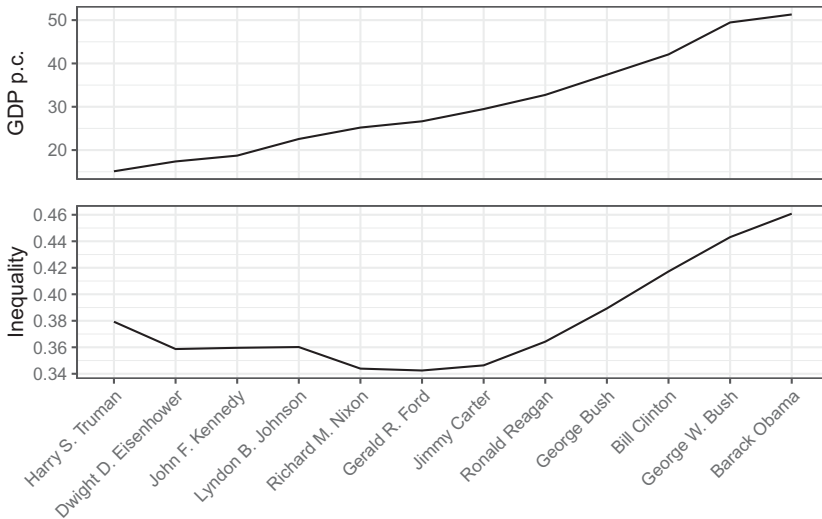


FIGURE 6.1. US GDP per capita (in 1,000 USD) and inequality by presidency.

6.6 SUMMARY AND OUTLOOK

In this chapter, we processed data from three sources to analyze trends in inequality and economic performance in the US across different presidencies. We did this using R's core functionality (with one exception: the `doBy` package). In particular, we imported data from text files such that they are available as data frames in R. This requires some caution, since data in text files may not be formatted according to standard conventions, and import errors can occur. Also, many standard text file formats do not explicitly specify the type of variables contained in a table, which is why R can only infer them (and this can go wrong). We merged our three datasets using R's `merge()` function, but also encountered the limitations of this process when dealing with more complex merges. The process of first creating the Cartesian product of the two tables, and then retaining the desired combinations is one way to bypass these limitations. Finally, we aggregated the data in R, applying a set of aggregation functions over groups of data.

Data processing using CSV files, data frames, and the functions I have presented so far is the standard workflow in R, and something you need to be familiar with. What we covered in this chapter is already a great improvement beyond a spreadsheet-based workflow: In R, you specify all your data operations in code. This way, you can correct, amend, and

re-run this code, but also share it with others. Still, R's basic data processing features do not necessarily constitute the optimal and most intuitive way to handle data in R. We therefore discuss an easier and, in several ways, better way to process your data in the next chapter: the tidyverse. Still, there are several lessons you should remember from this chapter:

- *Knowing base R is important:* Even though there are now several extensions of R's core data wrangling features (the tidyverse being the most prominent one), you still need to know your way around base R. Many important packages are not compatible with the tidyverse, and you often will have to work with R's core data structures.
- *Data frames as R's main tabular data structure:* For us as social scientists who mainly work with tables, it is essential to know the features and pitfalls of data frames. The syntax to extract rows or columns may often seem strange, but it corresponds to R's vector-based programming approach. As we saw in the chapter, R does maintain types for the columns in a data frame, but they can change dynamically as you add new data. This is something to watch out for, and it can make explicit type conversions (casts) necessary.
- *Make sure that imports work correctly:* Due to the implicit type conversions that can occur in data frames, it is necessary to check imported data carefully. Most text-based data files such as CSV do not preserve the column types of your data, which is why they must be inferred (or explicitly specified) during the import.
- *A few simple packages add standard data manipulation features:* Base R can do most basic operations on tabular data, but for some tasks, it is necessary to rely on external packages. In this chapter, we used the doBy package, which is one way to run basic aggregation operations on tabular data.

R and the tidyverse

In the previous chapter, we did some basic data processing with base R. Is this not enough to solve most of our tasks? Sure, but we can do better. Working with data in base R is oftentimes limited. Handling data frames can be difficult, and additional functions for data management must sometimes be added by importing external packages (e.g., the aggregation functions in the `doBy` package). Here, a set of packages, together referred to as the `tidyverse`, provides a much better and fully integrated way to work with data. It reflects our basic understanding of data handling very closely and also relies primarily on tables as the main data structure (variables as columns, rows as observations). The syntax, however, is much easier to remember, since the `tidyverse` uses natural names wherever possible and relies on verbs for actual operations to be carried out. Overall, this means that your code becomes more readable, not just for yourself but also for others trying to replicate it.

This chapter walks you through a simple application that demonstrates the use of the `tidyverse`'s data management features. Rather than a single R library, the `tidyverse` is actually a collection of several R packages for different purposes, which, however, use a common underlying logic and syntax. You may be familiar with the `ggplot2` package for producing graphics, but there are also several other extremely powerful packages that are part of the `tidyverse`. You can learn more about the entire `tidyverse` suite of packages at <https://www.tidyverse.org>.

All `tidyverse` packages are carefully designed, provide a wealth of useful features and are therefore highly recommended. In keeping with our focus on data management, however, we will only focus

on two of them. `tidyr` provides basic functionality to store data in rectangular (tabular) data structures, and `dplyr` offers powerful functions to manipulate data. To make these (and other) packages available in R, you need to install the entire tidyverse as described in Chapter 2 (unless you are using the pre-configured project environment) and then load it with:

```
library(tidyverse)
```

7.1 APPLICATION: GLOBAL PATTERNS OF INEQUALITY ACROSS REGIME TYPES

In the example for this chapter, we continue to explore the political determinants of inequality, but expand the scope of the analysis. While the previous chapter focused exclusively on the US, we now adopt a comparative perspective and study a global sample of countries over several decades. In particular, existing scholarship has suggested that inequality and regime type may be closely related (Acemoglu and Robinson, 2005; Houle, 2009). Our aim here is to create a dataset for analysis that allows us to track how patterns of inequality have developed over time in different political regimes.

Our main data source on inequality is again the World Inequality Database (WID, 2020), from which we obtain a cross-sectional time series dataset of income inequality estimates for many countries. We again rely on the full dataset, downloaded as a set of CSV files and merged into a single file for the exercises in this chapter. Again, we use the share of pre-tax income that goes to the richest 10% as our indicator for inequality (`p90p100`), which is available for many years and countries. The resulting data is available in the file `inequality.csv` in the data repository for this chapter.

In addition to the inequality data from the WID, we require data on the type of political regime that exists in a given country. Scholarship in political science has made different attempts to measure regime type along the dimension of autocracy vs. democracy. Our example in this chapter relies on the well-known Polity IV project (Marshall et al., 2015), which codes political regimes along a continuous dimension from -10 (full autocracy) to 10 (full democracy). Since political regimes change over time (e.g., by becoming more democratic or more autocratic), the Polity scores are provided as annual observations at the country level.

7.2 A NEW OPERATOR: THE PIPE

In data management, we often have to apply a series of operations to our data. We add and recode variables and filter selected cases, and merge our data set with others. The standard way of doing this in R is to apply a series of functions, creating intermediate datasets that are used as input at later stages of the process. Consider the following example of two artificial datasets with 20 annual observations, each of which contains a single additional variable (randomly assigned for simplicity):

```
dataset1 <- data.frame(year = 2000:2019, var1 = runif(20))  
dataset2 <- data.frame(year = 2000:2019, var2 = runif(20))
```

Let us assume we want to subset `dataset1` to observations that occurred after the year 2007 and merge it with `dataset2`. This is how to do this in base R:

```
dataset1_subset <- subset(dataset1, year > 2007)  
final_dataset <- merge(dataset1_subset, dataset2)
```

When we add more operations on our data, each of them generates a new intermediate result such as `dataset1_subset` and adds a new line of code. The `tidyverse` introduces a new operator that facilitates this process: The pipe `%>%` allows you to write your code in a more natural fashion, from left-to-right. What does this mean? In the following example, we again subset and merge the two datasets, but in a single line of code, and without intermediate results. Here, I demonstrate the use of the pipe with the same base R functions we used above – later, we will replace them with the appropriate ones from the `tidyverse`:

```
final_dataset <- dataset1 %>% subset(year > 2007) %>% merge(dataset2)
```

The idea of the pipe is straightforward: It takes a given dataset and sticks it into a new operation. As you can see in the example, we can chain several pipe operations and specify a complete “pipeline” in a single line of code. This code essentially says: “take `dataset1`, filter it such that it only contains the years after 2007, and merge the result with `dataset2`.” This code is easier to read, expresses the aim behind it more clearly, and the flow of the data is much more apparent. We also have to type less boilerplate code such as `subset(dataset1, ...)` or `merge(dataset1_subset, ...)`, because we are passing data directly from one operation to the next.

You may have noticed that when using the pipe operator, the input it sends to the next function becomes the first argument of that function;

for example, rather than writing `subset(dataset1, year > 2007)`, we can simply say `subset(year > 2007)` and the first input to the `subset()` function – the data that should be filtered – will be provided by the pipe. In the *tidyverse*, all functions are designed for this intuitive use of pipes, while many functions from outside the *tidyverse* are incompatible. So, the pipe is not a generic new operator in R; rather, it works only with the functions designed for it. If you would like to learn more, I recommend the chapter on pipes in Wickham and Grolemund (2016) and the documentation of the *magrittr* package.

7.3 LOADING THE DATA

As always, we need to load our data into R before we can start processing it. When working with the *tidyverse*, we use the `read_csv()` function for this. This is a new implementation of R's basic import function `read.csv()`, and you can use it in a similar way:

```
wid <- read_csv(file.path("ch07", "inequality.csv"), na = "")
```

The `read_csv()` function assumes that the fields in a row are separated with a comma – similar to the base R functions, you can use `read_delim()` for files with a different separator, which allows you to manually specify the field separator. For the WID dataset, it is important to specify the empty string (""), to indicate NA values, otherwise the function would interpret Namibia's two-letter code "NA" as NA. Let us now drop again the unnecessary columns in our data, such that we retain only the ones we need – the country identifier (a two-letter ISO country code), the year, and the value of the inequality indicator for the respective country and year:

```
wid <- wid %>% select(country, year, value)
```

Here you can see the pipe `%>%` in action: We take the original `wid` dataset and pass it to the `select()` function, which we ask to retain three columns. We store the result in `wid`, overwriting its original content. What is the result of this import? Let us take a closer look at the `wid` object by simply printing the first three entries. We do so again using the pipe operator, but stick the `wid` into a different function: `slice()`, which is used for subsetting datasets according to a row range provided:


```
wid %>% slice(1:3)

# A tibble: 3 x 3
  country year value
  <chr>   <dbl> <dbl>
1 AE     1990  0.593
2 AE     1991  0.595
3 AE     1992  0.597
```

When we print the dataset, we see that we are not dealing with a conventional data frame. Rather, the `read_csv()` creates something similar, called a “tibble.” A tibble is a modern version of a data frame. Tibbles serve the same purpose (which is to store tabular data), but with several tweaks that streamline and improve their use in practical applications. They have a much nicer default `print()` method (which is invoked when simply typing the name of the tibble): It reports the overall dimensions of the table, the names of the columns, and their types. More about tibbles can be found in the documentation of the `tibble` package.

Before we proceed, let us explore a bit more on how to work with tibbles in practice. Tibbles support all the basic operations you can do with data frames. For example, you can rename columns (which in fact is done much more elegantly as compared to data frames in base R):

```
wid <- wid %>% rename(p90p100 = value)
```

Apart from a nicer way to print, tibbles come with very useful features that make working with data much easier. In certain cases, however, you may have to explicitly convert a tibble to a proper data frame; this can be done using `as.data.frame()`. While you can use the `[]` and `$` syntax for extracting data from tibbles in a similar way as for standard data frames, I strongly recommend that you use the corresponding functions provided by the `tidyverse` for this if possible. They are designed to work with the pipe operator and improve readability of the code. We have already used the `select()` and the `slice()` functions, as well as the `filter()` function to extract rows based on a search condition. It is important to emphasize that these functions always return tibbles; this reduces confusion in comparison to the corresponding functions for data frames, which sometimes return a vector rather than a data frame (e.g., the `$` operator).

In addition to the data from the WID, we also require data on political regimes from the Polity IV project. This data is distributed both in Excel and SPSS format, and we choose the former. Since the Excel file contains a properly formatted data table, the import does not cause any issues. We

use the `readxl` package, which is also part of the tidyverse and hence creates a tibble:

```
library(readxl)
polity <- read_excel(file.path("ch07", "polity.xls"))
```

Since the Polity IV database contains many variables we do not need for our analysis, we only keep the main Polity indicator (`polity2`) in addition to the country (`ccode`) and year (`year`) identifiers:

```
polity <- polity %>% select(ccode, year, polity2)
```

We now have all the necessary data in two tibbles, `wid` and `polity`, which we use to generate a single dataset for our analysis.

7.4 MERGING THE WID AND POLITY IV DATASETS

Our next task is to merge the `wid` and `polity` datasets. Both contain annual observations at the country level, but merging them is complicated by the fact that there is no common country identifier yet. The WID refers to countries with a two-letter code, while the Polity database includes *Correlates of War* (COW) country codes, a system widely used in international relations and conflict research. Hence, we need to match the two-letter country codes in the WID to the COW codes. This task is greatly facilitated by the excellent `countrycode` library for R, which can translate between different codes and names for states. We can use the main translation function `countrycode()` from this package, which needs to know in which column the country identifier is stored that we want to translate (in our case, this is the `country` column). Also, it requires us to specify the coding system from which we want to translate (the ISO two-letter country code used in the WID, “`iso2c`”), and what coding system we want as output (“`cown`” is used to denote the numeric COW coding system). To store the result of the translation in a new column named `ccode`, we use the `mutate()` function:

```
library(countrycode)
wid <- wid %>% mutate(ccode = countrycode(country, "iso2c", "cown"))
```

Note that we get a warning from the `countrycode` function that two countries could not be merged. One of them is Palestine (two-letter code PS), which is not contained in the COW list of independent states. The second one is Serbia, where `countrycode` uses the old two-letter code for

Yugoslavia and therefore does not produce the correct COW code (345). We can fix the second issue by manually inserting the correct COW code for Serbia, using again the `mutate()` function to change the `ccode` variable:

```
wid <- wid %>% mutate(ccode = if_else(country == "RS", 345, ccode))
```

The `if_else()` function in the statement has three parts. It basically says: If the two-letter code is RS, use code 345 as the new value for `ccode`, otherwise use the existing `ccode` value as the new one. With a common country identifier, it is now straightforward to merge the two datasets based on `ccode` and `year`. Functions for merging in the tidyverse are called *join* functions, which is the technical term for combining tables in relational databases (we will learn more about joins in the next chapters). You may recall from the previous chapter that the default mechanism for joining datasets is to keep only those observations that have at least one match in the other dataset. This is called an *inner* join. Let us first try to use this function for merging Polity and the WID based on the `ccode` and `year` variables:

```
dataset <- polity %>% inner_join(wid, by = c("ccode", "year"))
```

The entire `polity` table has 17,562 observations, while the merged dataset has only 3,142. This is due to the fact that the WID only covers a subset of countries – if we now retain only observations from Polity with a match in the WID, all the countries that are contained in Polity but not in the WID are removed from the merged dataset. This is the standard behavior of all inner joins. If you need to retain all records from the first (the left) or the second (the right) dataset – similar to the `all.x` and `all.y` parameters of the `merge()` function in Chapter 6 – you could use a “left” or a “right” join, which can be executed with the `left_join()` and the `right_join()` function.

7.5 GROUPING AND AGGREGATION

The WID only covers a subset of all countries worldwide, and even for these, the inequality measure (`p90p100`) contains many missing values. We should first get a better overview of our dataset as regards the countries and time periods it covers, but also the countries/years for which we have valid observations from the WID. To generate some useful statistics to answer these questions, we use grouping and aggregation. Recall from Chapter 3 that data aggregation is the definition of different groups

or subsets of data, with an aggregation function applied to each of these groups separately. As we have seen in the previous chapter, in a conventional data frame these groups can be dynamically defined in the `summaryBy()` function.

In a tibble, however, this mechanism is slightly different. Tibbles allow you to define the grouping as a *feature of the tibble*, which is then used whenever grouping functions are applied to it. Grouping is enabled with the `group_by()` function:

```
dataset <- dataset %>% group_by(country)
dataset

# A tibble: 3,142 x 5
# Groups:   country [107]
  ccode year polity2 country p90p100
  <dbl> <dbl>   <dbl> <chr>   <dbl>
1     2  1913     10 US      0.423
2     2  1914     10 US      0.430
3     2  1915     10 US      0.422
# ... with 3,139 more rows
```

You can see in the output that the tibble now has the grouping by country enabled, and that there are 107 different groups (countries). We can now summarize the tibble, which will automatically be done separately for each of the groups. To see how many observations we have per country, we use the aggregation function `n()` that counts the number of cases in each group:

```
dataset %>% summarize(count_obs = n())

# A tibble: 107 x 2
  country count_obs
  <chr>     <int>
1 AE         27
2 AL         17
3 AO         28
# ... with 104 more rows
```

The output of this function creates a new tibble containing the summary statistics we computed. While we have 100 or more years' worth of data for countries such as France and the US, for many others the coverage is much more limited. For our analyses below, it would be useful to know since what year particular countries are covered in the WID, such that we can adjust our period of analysis accordingly. Therefore, we expand our summary such that it outputs the first year with an inequality estimate

during the observation period. To do that, we use the minimum as an aggregation function:

```
dataset %>% summarize(firstyear = min(year))
# A tibble: 107 x 2
  country firstyear
  <chr>      <dbl>
1 AE          1990
2 AL          1996
3 AO          1990
# ... with 104 more rows
```

The example shows that data aggregation in the tidyverse is very elegant, and is a considerable improvement over the mechanism we used in the previous chapter. One of the main advantages is that we can define aggregation functions only for particular columns they should be applied to, and have full control over the naming of the columns holding the aggregated values. As you can see in the output, for many countries, there are few inequality estimates for years earlier than 1990, which is why we restrict our analysis below to the years 1990 and later. Before we do that, however, we disable grouping of our main dataset with `ungroup()`, since the next operations on the dataset do not need grouping:

```
dataset <- dataset %>% ungroup() %>% filter(year >= 1990)
```

To track patterns of inequality by regime type, we need a dataset with average annual values of inequality, computed separately for democracies and autocracies. As a first step, let us introduce a new binary variable `democracy`, which identifies those countries that are democracies in a given year. Following the standard convention, we code country-years with `polity2 >= 6` as democracies. Since we have missing values in the `polity2` variable, we drop these observations before computing the aggregation:

```
dataset <- dataset %>%
  filter(!is.na(polity2)) %>%
  mutate(democracy = if_else(polity2 >= 6, T, F))
```

Since we need average inequality values for each year and separately for democracies and autocracies, we need two levels of grouping. We therefore enable grouping again with:

```
dataset <- dataset %>% group_by(year, democracy)
```

Now, we can summarize our data as introduced above and compute the average level of inequality, separately for the democracies/autocracies and each year in our sample:

```
data_agg <- dataset %>%
  summarize(mean_ineq = mean(p90p100))
```

The result of the aggregation is stored in a new dataset, `data_agg`, which we use in the next section.

7.6 RESULTS: GLOBAL PATTERNS OF INEQUALITY ACROSS REGIME TYPES

Figure 7.1 plots the aggregated values for democracies and autocracies over time. Keep in mind that the WID does not cover all countries worldwide, so this result must be treated with some caution. The plot shows there are notable differences in the level of inequality between democratic and autocratic countries. In democracies, around 40% of the income go to the richest 10% of the population, which is a large share. In autocracies, however, this share is even higher, with values of more than 50%. So clearly, democracies seem to be doing better than autocracies in creating a more equal society. However, the figure also shows that in democracies, the level of inequality is increasing over time, while it is slightly decreasing in democratic countries. We should mention though that by simply averaging over all countries, our simple comparison hides much variation *within* each of the two categories. In particular, there are considerable differences among democratic countries when it comes to inequality in the population.

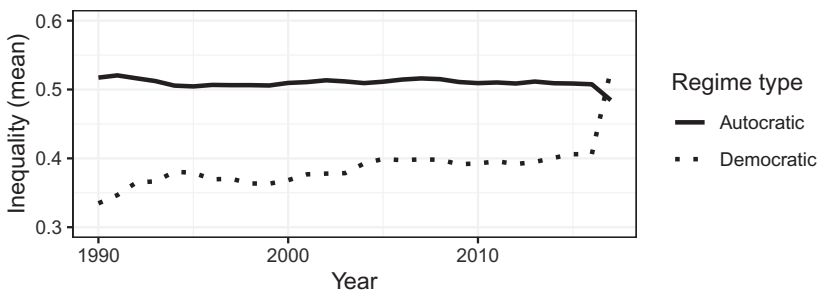


FIGURE 7.1. Trends in inequality over time, for democracies and autocracies.

7.7 OTHER USEFUL FUNCTIONS IN THE TIDYVERSE

Before we conclude this chapter, let us briefly review some other functions that can be really helpful to quantitative work in the social sciences.

7.7.1 Lags of Variables

The first of the functions help us create a lag of a variable (e.g., the value of that variable in the previous time period). Continuing with our example above, we may want to do a simple regression analysis of how democracy affects inequality. For analyses of this type, it is common to lag the main independent variable – in other words, this means that we predict the level of inequality with the *previous year's* democracy score for the respective country. For this, we need to extend our dataset such that in addition to the contemporary democracy scores in the `polity2` variable, we also have a new variable with the democracy scores from the previous year.

To create this variable, we use the `mutate()` function that you already know from above. In this function, we use the `lag()` function applied to the `polity2` variable, which is the variable we want to lag. This function also needs to know which variable specifies the temporal order of the data, in our case the `year`. Since we want to compute the lags separately for each country, we `group()` our dataset first, and `ungroup()` it after the operation is complete:

```
dataset <- dataset %>%  
  group_by(ccode) %>%  
  mutate(polity2_lag = lag(polity2, order_by = year)) %>%  
  ungroup()
```

With the lagged predictor `polity2_lag` now being a new variable in our dataset, we can, for example, run a simple linear regression to test again our above result that democracies tend to have lower values of inequality.

7.7.2 Converting between Wide and Long Tables

The tidyverse also contains functions to convert between “long” and “wide” tables. What was this again? Recall our discussion in Chapter 3, where we talked about the features of a well-designed table. Good tables are those where you can add data by adding more rows to the table. Our dataset above is such a table: If additional data about more countries and/or years became available, we could just add a new row for each country-year we want to insert. This type of table is also called a “long”

table. However, some of the data we use in social science projects comes in poorly designed tables. For example, for country-level data with annual observations, you sometimes encounter tables with one row per country, and a column for each year covered in the dataset. This format is called a “wide” table.

Let us use the data from the WID to illustrate how we can convert tables between long and wide formats. The original data is contained in `wid`, which is a “long” table. For illustration purposes, we simplify the table a bit and retain only three countries with observations from three years, and we also drop the COW code:

```
wid_simple <- wid %>%
  select(-ccode) %>%
  filter(year >= 2000 & year <= 2002) %>%
  filter(country %in% c("FR", "US", "DE"))
```

We can convert the table to a “wide” format with the `pivot_wider()` function from `tidyverse`. You need to specify the variable in the table that contains the values for the new header names (in our case, this is the `year` column), as well as the variable that contains the values you want in the converted table (in our case, the inequality levels in the `p90p100` column). It is useful to sort the table with `arrange()` beforehand, such that the new columns are properly ordered:

```
wid_wide <- wid_simple %>%
  arrange(year) %>%
  pivot_wider(names_from = year, values_from = p90p100)
```

This is what our new table looks like:

```
wid_wide
# A tibble: 3 x 4
  country `2000` `2001` `2002`
  <chr>    <dbl> <dbl> <dbl>
1 DE      0.316  0.316  0.317
2 FR      0.331  0.334  0.328
3 US      0.439  0.428  0.427
```

Since “wide” tables are usually difficult to deal with, we usually need to convert them to a “long” format rather than vice versa. This works with the `pivot_longer()` function:

```
wid_long <- wid_wide %>%
  pivot_longer(-country, names_to = "year", values_to = "p90p100")
```


This returns the data again in a long table, the format that should be preferred for most of the work we do in the social sciences:

```
wid_long
# A tibble: 9 x 3
  country year  p90p100
  <chr>   <chr>   <dbl>
1 DE     2000    0.316
2 DE     2001    0.316
3 DE     2002    0.317
# ... with 6 more rows
```

7.8 SUMMARY AND OUTLOOK

This chapter introduced the *tidyverse* framework for R, a collection of different R packages that integrate well with each other and use a consistent grammar. Although we have used the *tidyverse* only for simple data management operations here, its functionality goes well beyond this (e.g., with the *ggplot2* package for graphics). Much work in the *tidyverse* is done using a new operator, the pipe, which allows you to write code that is simple and intuitive to understand. I demonstrated how to work with tibbles, an extended version of the usual R data frame. The *tidyverse* offers a number of functions to perform standard data operations, such as selection, aggregation and merging of tables. It also has new and improved functions to import and export data from various different file types (see also the examples in Chapter 4).

In general, it is highly recommended to perform your data work with the *tidyverse* and its associated packages. It is elegant, powerful, and efficient, and allows your code to be easily understood and replicated by others. Although for some specialized types of data (e.g., spatial data), the integration with the *tidyverse* is not without pitfalls, all common tables with numbers and/or text can easily and conveniently be processed with it. It even interfaces well with relational databases, which we cover in the next chapters. Nevertheless, as an apt user of R, you should know both “worlds” well – base R, and how it differs from the *tidyverse*. You can then decide which one is the best choice for a given project. From this chapter, there are a number of recommendations for your work:

- *Use the pipe wherever possible:* The pipe operator allows for an improved, much more logical workflow for most data management operations. For example, in base R, users tend to create new R objects for every intermediate step of a data processing sequence. This can

be confusing and error-prone. Arranged as a pipeline with the pipe operator connecting the different steps, there are no intermediate results we need to deal with – all that matters is how we get from the input to the final result.

- *Try not to mix:* Being fluent both in base R and the tidyverse, it is possible to switch back and forth between the different approaches in a single script. You should try to avoid this. If you opt for the tidyverse in your script, try to stick with it and implement the entire workflow using its functions. This makes your code consistent and easier to follow for others.
- *Watch out for potential issues with the tidyverse:* Despite the considerable advantages that the tidyverse has for most data management tasks, there are some potential drawbacks you should keep in mind. The tidyverse includes a wealth of functions, which means that conflicts can occur if other packages include functions with the same name. You see a message alerting you to (usually uncritical) conflicts with base R functions when you load the tidyverse. Once other packages are loaded, these conflicts can be problematic. Also, due to its size, the tidyverse depends on a large number of other packages, so your R installation will grow considerably and installation issues can arise.
- *Remember the conversions between long and wide tables:* As we have seen in the chapter, the tidyverse offers a convenient way to convert between wide and long tables. This is a task you may encounter from time to time, since existing tables are often formatted for humans to look at (and may therefore be distributed in a wide format). You should resist the temptation to manually convert them, and instead rely on R to do this.

PART III

DATA IN DATABASES

Introduction to Relational Databases

All data processing we did so far in this book was file based. That is, our data was stored in files, and these files were read by our data management software (e.g., R). Using this software, we processed the data in various ways, and output the resulting datasets again to files. In this workflow, files are the basic containers of our data, which we use to store it persistently and to share and disseminate it. Due to its simplicity, flexibility, and versatility, file-based data storage is used for the vast majority of social science research projects. In this chapter, we go one step further. Rather than keeping our data in simple files, we use a kind of software specifically designed for storing and processing data: a database management system (DBMS).

There exist many different types of DBMS. Our focus here will be on what is probably the most common one: a *relational* DBMS. These systems are built on the idea that all data should be contained in tables that are linked to each other. This is an idea that should be straightforward to us, since we have dealt with tables from the beginning of this book. The concept of a relational database goes back several decades. In computing, this is a long time. Still, these databases continue to be around, in different forms and flavors, which attests to the power and flexibility of the concept. So, how can these database systems improve upon the standard file-based data management workflow?

- Organizing your data in a single file is simple, but quickly becomes difficult if your data is spread out across different files. If you follow the advice on a “good” table design in Chapter 3, you will probably require several tables to store data without redundancies; for example,

if you use data with annual estimates of the GDP for different countries, you will use one table for the variables at the country level that remain constant over time (such as the year of independence), and one table for those variables that change annually (the GDP estimates). Using a single table is not a good idea, since you would have to repeat the constant country-level variables for every annual observation. This would mean that part of your data is redundant, and it is something we should avoid. With file-based data processing, however, each table requires a new file, so your entire “database” consists of many files and becomes difficult to maintain. Relational database systems, in contrast, are designed to manage many different tables simultaneously. Each database contains all tables for a project, keeping them together in one place.

- Not only are data spread out across many files difficult to handle, but they can also become internally inconsistent. Imagine in your table with GDP estimates, you have an entry that refers to a particular country in the country table, for example, Switzerland. You would like to join the two tables to create a dataset for analysis. However, what if Switzerland is somehow missing from the countries table? With file-based data storage, there is no mechanism to ensure that tables that refer to each other are *consistent* – that is, that links from one table to another are indeed valid and point to actual data. Relational databases have different mechanisms to maintain this *relational integrity*, that is, the consistency of data across different tables as you add, delete, or update data.
- When your tables become large, the performance of data operations becomes an issue when relying on file-based data storage. Loading a small file and filtering particular observations from it is easy and fast, but takes more and more time the larger your table becomes. Relational databases are designed for fast and efficient processing of your data. Operations such as searching and updating your data are tuned for optimal performance, even if your data is so big that it cannot all be kept in the computer’s memory at the same time. All this complexity is safely hidden from you as the user of a DBMS – you tell the system what you want to do with your data, and the system internally uses whatever machinery is necessary to carry out these tasks.
- Finally, collaboration between different researchers is difficult in a file-based workflow. Different people would have to exchange different versions of files, while making sure that the content of these files remains consistent (see the second point above). Imagine two

researchers trying to update data in different columns of the same table: This is almost impossible in a file-based workflow, since both would have to work on a single copy of the same file (and possibly destroy the other's changes when saving it). Database management systems are centralized in a way so that many users can access the data at the same time. The different tables in a database can be modified by different users, according to the permissions they have. Each table exists only once in a database, rather than in different copies of a file.

While these advantages of relational DBMS will become clearer in this and the next chapters, using such a system in lieu of simple data files entails a technical overhead. Ultimately, it is up to you to decide what system or workflow you use for your project. If your project is small and narrow in scope, it will be perfectly fine to keep your data in files only. However, if your project involves several interlinked tables, some of which are large, or if more than one researcher works on the data, then you should consider using a database management system. In the following section, we introduce the general setup of such a system.

8.1 DATABASE SERVERS AND CLIENTS

Many database management systems are set up in client-server architecture. That is, the DBMS runs on a computer somewhere on the Internet (which is called the “server”), and so-called “clients” connect via the network to this server, send data processing instructions, and fetch data. Figure 8.1 illustrates this graphically.

The big circle represents the database management system. In the figure, this DBMS manages just one database, but in reality, there can

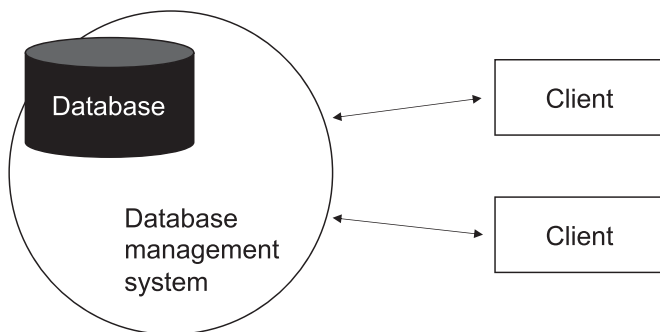


FIGURE 8.1. Interacting with a database management system.

be many of them. The clients, depicted as rectangles, interact with the DBMS: They connect to a particular database, can upload and retrieve data from it, or send instructions for data processing. When you connect to a database server from R (as we will do in this and the following chapters), your R instance is one of those clients. R, however, is not the only client software that can communicate with a database – there are many others. For example, most DBMS come with simple text-based clients, which you can run on your local machine to send commands to the server. Also, all the major programming languages and most statistical software packages have extensions that allow you to connect to a database server.

In many cases, these clients are based on computers other than the database server, and the communication between clients and the server is done over the network. While it may seem unnecessarily complicated, this separation is actually very useful. For once, it allows the database server to be operating on powerful hardware, which is necessary in particular if you deal with large datasets and/or complex calculations. Running these operations on your local workstation or laptop would be much slower and, in many cases, even impossible. Also, the shared client-server setup is well designed for multiple users accessing a single database, which is very useful for collaborative projects.

The communication with a (relational) database server is done with a language designed for this purpose, the Structured Query Language (SQL). Some pronounce it as “Sequel,” others prefer “Ess-Queue-El.” SQL, as the idea of a “relational” database in general, has a long history, and there are many different dialects of the language. In this book, we rely on the PostgreSQL database system as well as the SQL dialect it uses. PostgreSQL is free and open source, well-known, and many other programming languages and tools can interface to it. While other relational databases such as MySQL, Oracle or Microsoft SQL Server differ in the features they offer (and therefore also the SQL dialect they understand), this book introduces some general concepts of the relational approach and SQL that apply regardless of what system you work with.

To set up a client-server structure for the purpose of this book, we have installed the PostgreSQL server on your system in Chapter 2. This server should now be running. If not, you need to go back to Chapter 2 and the online installation instructions on the book’s companion website. Now is also a good time to follow the instructions in Chapter 2 to create a new database specifically for this chapter, if you have not done so already.

We use the name `dbintro` for this database, but you are of course free to choose any name you prefer.

R has a generic interface to communicate with relational databases called DBI, the “R Database Interface.” The `RPostgres` package we use is built on this interface. There are many different types of database servers, and using this standardized interface means that you connect to any of them in the same way. For these connections, you typically need to specify at least the type of server (PostgreSQL, MySQL, etc), the name of the database, and your username and password (remember again to change the username and password to match your setup). Here is how to do this for our server and the `dbintro` database:

```
library(RPostgres)
db <- dbConnect(Postgres(),
  dbname = "dbintro",
  user = "postgres",
  password = "pgpasswd")
```

If you get an error that says something like “could not connect to server,” the PostgreSQL server may not have been installed properly, or that you forgot to start it. In this case, I recommend that you go back to Chapter 2 and complete again the steps described there and in the online instructions.

Let us take a closer look at the database connection. The function to connect to the database server is `dbConnect()`. It returns a connection object, which we call `db`. We will use this object later to send all sorts of commands to the server. When we are done, we should close the connection properly (see the end of this chapter). To connect, we need to provide a few connection parameters to the function. First, this is the name of database we want to connect to, the `dbname`. Since a given server can host many different databases with different users and for different purposes, we need to specify which one we want to work with. In this chapter, this is the `dbintro` database, but we will use other databases in the following chapters. Second, you need to provide your username and password, so that the server knows who it is communicating with. This user-based authentication also allows you to later define different permissions for different users, for example, by giving some users read-only access to the database, while allowing others to modify the data. This is something we return to in Chapter 10. Note that in the above example, we omit several other connection parameters, for example, the name of the computer running the server. This is because you are running

PostgreSQL on your local machine, using default settings. It is easy to extend the above code to connect to servers on other computers, if the need arises.

8.2 SQL BASICS

Before we use the connection to our database to send commands and work with actual data, let us cover some basics of the SQL language. As we will see below, much of it is actually close to human (English) language, so it is not too difficult to understand. Unlike many other programming languages, SQL is case-insensitive, so it does not matter if you write `SELECT * from myTable`, or `select * FROM MYTABLE`. However, I strongly recommend that you follow the convention to spell SQL keywords in upper case, and names of tables, columns, and functions in lower case. The above statement then becomes `SELECT * FROM mytable`. I will follow this convention throughout the book.

In a relational database, all data is contained in tables, and the SQL language is designed around these tables. As social scientists, we refer to the data in our tables as *observations* or *cases*, each of which consists of different *variables*. In the database world, we prefer the terms *rows* (or *records*) and *columns* (or *fields*) of a table. Tables in relational databases have *typed* columns. This means that we need to define whether we want to stick text or numbers (or something else) into a given column, and the database system then ensures that only allowed data of the given type is stored in that column. This is similar to R's data frames (although R adjusts types dynamically, while a database does not), but very different from the non-standardized tables you can find, for example, in spreadsheets. Different database systems (and therefore, the different SQL dialects they use) vary in the column types they offer. In our discussion, we will not go into the details regarding these differences, but rather try to introduce SQL that also works beyond the PostgreSQL system we use for our exercises.

Relational databases employ a strict separation of data structure and the data itself. This is why in SQL, there are several dedicated commands to define and modify the structure of your tables: You can introduce new tables, define which columns they should consist of and what the types of these columns should be, and you can also delete columns or entire tables. This category of statements is called *data definition*. The second category of commands is for the updating of the data contained in the tables of a database: You can insert new rows or delete existing ones, or

update the information contained in particular fields of a table. These are examples of *data manipulation* statements. Finally, we ultimately want to extract data from the tables in our database, which is why we need *data extraction* commands.

In our example below, we use R as a client to connect to the database and to send SQL statements to it. While the R code and the functions we use for this are of course specific to R, the SQL code is not – you could send the same statements via a different client, and the database would do exactly the same. However, we will also be using some convenience functions for R that facilitate, for example, the loading of data into the database. These are features offered by R (or rather, its database interface DBI) and not by SQL.

8.3 APPLICATION: ELECTORAL DISPROPORTIONALITY BY COUNTRY

In democracies, the main way by which institutions aggregate the preferences of citizens is through elections. In an election, citizens cast their votes, and the result of election – for example, the composition of the national parliament – is supposed to reflect the distribution of voters. However, electoral systems vary tremendously in the way they translate the votes cast in an election into a particular distribution of seats, which has long been the focus of much research in comparative politics (see e.g. Grofman and Lijphart, 1986). One outcome that is associated with the voting system is the “disproportionality” of an electoral result. Disproportionality refers to the difference between the *share of votes* that a party achieves in an election, and the *share of seats* it ultimately gets in parliament. In a majoritarian system with its winner-takes-all logic (as in the UK, for example), disproportionality between vote and seat shares will be highest, since the votes cast for candidates that do *not* win a precinct ultimately do not count. Disproportionality is typically measured using Gallagher’s least squares index, which is defined as the square root of half the sum of squared differences between seat shares (S_i) and vote shares (V_i):

$$\text{LSq} = \sqrt{\frac{1}{2} \sum_{i=1}^n (V_i - S_i)^2}$$

What is the disproportionality in actual elections? To find out, we use data from the *ParlGov* database, a comprehensive resource with

information on election results, political parties, and the composition of governments in EU and OECD countries (Döring and Manow, 2018). Importantly for our purpose, ParlGov consists of a set of different tables, since it is already structured internally as a relational database. In this chapter, we are going to use only one of these tables, which is the one with data on elections; in the next chapter, we will extend our work with ParlGov and add another table from the database. These tables we use are not the original ones. They have been revised slightly for the purpose of our exercises: I dropped variables we do not need and elections with missing data, kept only parliamentary elections, and retained only European countries.

8.4 CREATING A TABLE WITH NATIONAL ELECTIONS

We have a PostgreSQL database and can connect to it, but so far we do not have any tables. As we have seen above, relational databases require us to specify the structure of the data (i.e., the tables) first, before we can add data. This sounds more difficult than it actually is – all we need to do is tell PostgreSQL what the name of the new table should be, what columns it should contain, and what the types of these columns are. First, let us take a look at the election data we have from ParlGov. If you open the file `elections.csv` in a text editor, this is what you should see:

```
election_id,country_name,election_date,party_id,vote_share,seats,seats_total
402,Austria,1945-11-25,1013,49.8,85,165
402,Austria,1945-11-25,973,44.6,76,165
402,Austria,1945-11-25,769,5.4,4,165
```

The structure of the table is not difficult to understand. Each line contains an election result for a political party. Each election has its `election_id` and is linked to a country. The data also contain the `election_date`. The next three columns store the party-specific information about the election result: the party (identified by the `party_id`), the `vote_share` it achieved, and the number of seats it obtained. Finally, the last column contains the total number of seats that were filled in the respective election.

Let us now create a table structure in SQL, so that we can import the ParlGov election data. There are a few things we need for this. First, recall that we created our database connection above, and can access it via the `db` connection object. We tell R to use this connection whenever we send an SQL command to the database. The `dbExecute()` function is what we need for this. It has two parameters: First, the connection (this is our `db`

object); second, a string with the SQL command we want to send to the database. Creating a new table in SQL is done using the `CREATE TABLE` statement:

```
dbExecute(db,
  "CREATE TABLE elections (
    election_id integer,
    country_name varchar,
    election_date date,
    party_id integer,
    vote_share real,
    seats integer,
    seats_total integer)")
```

As mentioned above, we write the SQL keywords in upper case. What exactly happens in this statement? We provide the name of the new table (`elections`), followed by the list of columns and their types. The set of columns and types needs to be enclosed in parentheses and separated by commas. In the example, we use four different types of columns:

1. Integer numbers, given as `integer`.
2. Text, given as `varchar` (a set of characters with variable length).
3. Date values, given as `date`, which will later help us order elections by calendar date as well as conduct other date-based calculations.
4. Decimal values. Here, we use `real` numbers, which is one of PostgreSQL's decimal number types.

If you need more information about the column types PostgreSQL can handle, take a quick look at the documentation at <https://www.postgresql.org/docs/current/datatype.html>. To reiterate our point from above: The code we present here mixes SQL code (the long text starting with `CREATE TABLE`) with R code (`dbExecute()`) to send it to the database. If you were to use a client other than R to connect to the database, you would need the SQL part, but not the R function calls. To check whether the table was successfully created, R's DBI interface has a useful function that prints out a list of all tables in a database:

```
dbListTables(db)
[1] "elections"
```

As the output shows, we currently have one table in the database, which is the `elections` table we created. Now, we have completed the definition of our table structure and can fill it with data. This is done

using the SQL `INSERT INTO` command. To insert some data for the 1919 elections in Austria, you use the following statement:

```
dbExecute(db,
  "INSERT INTO elections
  VALUES (1030, 'Austria', '1919-02-16', 97, 40.75, 72, 170)")
```

Here we specify which table we want to add our data to, and provide in parentheses the values the respective columns should have. Note that we have to surround string values such as `Austria` with single quotation marks (`'`). The same holds for date values, which will automatically be recognized as a date if they are formatted in a standard way. The above example inserts data for *all* columns in the table, and the new values have to be provided in the exact same order in which the columns appear in the table (which is what we defined above). If we want to insert data for only specific columns, and possibly in a different order, we have to specify the columns we want to insert into – let us take an election from Belgium as an example:

```
dbExecute(db,
  "INSERT INTO elections
  (election_id, country_name, election_date, vote_share, party_id)
  VALUES (872, 'Belgium', '1908-05-24', 22.6, 2422)")
```

This omits the two fields with the party's seats and the total number of seats, and uses the party id as the last value. To check whether this worked, let us move from data manipulation to data extraction. We use another very important SQL command: `SELECT`. In the simplest form, `SELECT` can be used to retrieve all data from a table, without any filtering or transformations. Let us do this for all data we have in `elections`, which at this point are only two elections from Austria and Belgium:

```
dbGetQuery(db, "SELECT * FROM elections")
```

	election_id	country_name	election_date	party_id	vote_share	seats	seats_total
1	1030	Austria	1919-02-16	97	40.75	72	170
2	872	Belgium	1908-05-24	2422	22.60	NA	NA

Since this is an SQL statement, which, unlike `dbExecute()` above, *returns* data rather than just manipulating it, we need to use a different R function that fetches data from the database: `dbGetQuery()`. For simplicity, we simply output the result of this function – a data frame – to the console, but in a regular script, you would store it in a new R object for later use. The asterisk `*` in the SQL code stands for “all columns,” and we need

to provide the name of the table we want to extract from. You can see that the `seats` and `seats_total` values for Belgium are correctly stored as `NA`, since we chose not to provide them when we inserted the data for Belgium. Here, `NA` is R's convention to represent missing data. In relational databases, missing values are usually encoded as `NULL` (this is not a string, therefore no quotes), and the DBI functions take care of mapping R's NAs to `NULL` values in the database.

`SELECT` is probably the most powerful command in SQL, and we can cover only some variations of the above statement. Let us assume we want to see only a subset of the columns in a table. This is possible by specifying the columns names explicitly, rather than using the wildcard character `*`:

```
dbGetQuery(db, "SELECT country_name, election_date FROM elections")
```

	country_name	election_date
1	Austria	1919-02-16
2	Belgium	1908-05-24

We can also extract just a subset of all data with the `WHERE` keyword:

```
dbGetQuery(db,
  "SELECT country_name, vote_share
  FROM elections
  WHERE vote_share > 40")
```

	country_name	vote_share
1	Austria	40.75

It is also possible to dynamically compute new columns in a `SELECT` statement, for example, to output the vote share as a proportion rather than a percentage:

```
dbGetQuery(db,
  "SELECT country_name, vote_share / 100 AS vote_share_prop
  FROM elections")
```

	country_name	vote_share_prop
1	Austria	0.4075
2	Belgium	0.2260

In this statement, we transform the vote share by dividing it by 100, and output the result in a new column called `vote_share_prop`. This new column appears only in the result – it does *not* change the original table in any way. There is much more we can do with `SELECT` statements, some of which is shown below after importing the entire ParlGov table into our database. But before we do so, we first need to remove the data we have added to our table with a `DELETE` statement:

```
dbExecute(db, "DELETE FROM elections")
```

As with any operation that removes data, you need to be very careful: This statement deletes *all your data in the table* but keeps the table structure. Now that we have an empty table, we can insert all of the data from our election data frame into the table. Writing separate INSERT statements for each row would be very cumbersome. Luckily, there are several ways in which you can easily load data into a PostgreSQL table. In the example below, we use a function from R's DBI interface, which takes an R data frame and sends it to a database. The file `elections.csv` in the data repository contains the elections data. We load the data into R and then use `dbAppendTable()` to append it to the empty `elections` table that we created above:

```
elections <- read.csv(file.path("ch08", "elections.csv"))
dbAppendTable(db, "elections", elections)
```

As all database functions, `dbAppendTable()` first takes the database connection to be used. Second is the name of the database table that the records should be appended to. Last, we specify the data frame that should be appended to the given table. If the import is successful, you can again use the SELECT command to browse some of the data (the LIMIT keyword restricts the output to a certain number of rows). Note that in a database table, the data has no fixed ordering; the following SELECT statement returns two rows, but on your system, these may be different from the ones you see in the example:

```
dbGetQuery(db, "SELECT * FROM elections LIMIT 2")
```

	election_id	country_name	election_date	party_id	vote_share	seats	seats_total
1	402	Austria	1945-11-25	1013	49.8	85	165
2	402	Austria	1945-11-25	973	44.6	76	165

The second, and slightly more convenient way to load data is through the DBI's `dbWriteTable()` function. This function takes a data frame and a table name, and sticks the data into a given database table. If the table does not exist, it can even generate a new table structure before inserting the data. To try this, we first delete the entire table:

```
dbExecute(db, "DROP TABLE elections")
```

Now, we want to add the entire elections table to the database in a single step. Before we can do this, we need to make sure that all the columns in the data frame have the correct type. This is not the case for

the election date, which is still a string variable. Therefore, we first convert it to a date, and then upload the entire table in one step:

```
elections$election_date <- as.Date(elections$election_date)
dbWriteTable(db, "elections", elections)
```

Again, the `dbWriteTable()` function is convenient, since it creates the new table in the database according to the structure of the data frame, and then uploads the data contained in the data frame to it. You can check again with a `SELECT` statement that the import was done successfully.

After the successful import of the `election` table, we create a new field for the year of an election with `ALTER TABLE`, to make future operations with yearly aggregations easier. In PostgreSQL, you can extract some part of a date (e.g., the year, the day, or the month) with the `extract()` function:

```
dbExecute(db, "ALTER TABLE elections ADD year integer")
dbExecute(db,
"UPDATE elections
SET year = extract(year from election_date)")
```

8.5 COMPUTING ELECTORAL DISPROPORTIONALITY

With our table successfully imported into our relational database, we can now proceed to compute the Gallagher index of disproportionality in SQL. We do so for each election, using data on vote shares and seat shares. Our `elections` table from ParlGov already contains information about each party's vote share (in the `vote_share` variable, in percent). We need a separate field for the seat share, which we can simply compute as the fraction of the actual seats for the respective party (`seats`) and the total number of seats in parliament (`seats_total`). Following the convention of separating data definition from data manipulation, we first need to create an (empty) new field for the seat share:

```
dbExecute(db, "ALTER TABLE elections ADD seat_share real")
```

We again use a `real` type for this variable, since it will contain decimal numbers. The `ALTER TABLE` command can not only be used for adding new columns, it can also delete (`DROP COLUMN`) them or change their type (`SET DATA TYPE`). Now, we can fill the new column by computing the percentage of the total seats that the party received. If we were to do so by simply dividing `seats` by `seats_total` and multiply it by 100 (to obtain a percentage), we would get the wrong result: The result of dividing two

integer numbers in PostgreSQL is again an integer, which is why the result would have the decimal places removed. To fix this, we multiply it with a decimal number (100.0) and not an integer number (100) – as a result, PostgreSQL will carry out the computation with decimal numbers, which is what we want:

```
dbExecute(db,
"UPDATE elections SET seat_share = 100.0 * seats / seats_total")
```

The next step is to calculate the difference between the vote share and seat share for each party in each election, square it, and then compute the sum over all these squared differences for a given election. Let us start with the first part, the squared differences between vote shares and the seat shares. We can simply include it as an additional field in a SELECT statement, something we have already introduced above. When performing this operation, we need to make sure to convert the ParlGov vote share from a percentage to a proportion, to make it comparable to the seat share. The power() function in SQL performs the exponentiation; alternatively, you could use the ^ operator for this:

```
dbGetQuery(db,
"SELECT power(vote_share - seat_share, 2) AS squared_diffs
FROM elections LIMIT 2")
```

	squared_diffs
1	2.941746
2	2.133375

Note that we are computing these squared differences only for illustration purposes; they are simply displayed, but not stored for later use. Next, we amend our SQL statement to compute the sum of these squared differences across all parties in an election. You recognize that what we need is simply an aggregation operation with grouping, similar to what we have done in previous chapters: We combine all squared differences with the same election_id and aggregate them by summing them up. In SQL, aggregation is yet another thing you can do with a SELECT statement: All you need to do is specify (one or more) aggregation functions, as well as the grouping levels with the GROUP BY keyword. We also divide the sum of squared differences by two, and take the square root:

```
dbGetQuery(db,
"SELECT
election_id,
sqrt(0.5 * sum(power(vote_share - seat_share, 2))) AS lsq_index
FROM elections")
```

```
GROUP BY election_id LIMIT 2")
election_id lsq_index
1          828  2.621663
2          938  2.389561
```

In the first part of the `SELECT` statement, we define what we would like to get out: First, this is the grouping variable `election_id` itself, so that we know which election a result refers to. Second, this is our above computation of the squared differences, but wrapped in the `sum()` function. This is the aggregation function that the database applies to each group as defined by the grouping variable. This sum is then multiplied with `0.5`, and the square root function is applied to it. As above, we use the `FROM` keyword to tell the database which table to use for this calculation. Finally, we need to define the grouping variable using the `GROUP BY` keyword (the `LIMIT` keyword again limits the output to two rows, which is simply for presentation purposes).

8.6 RESULTS: ELECTORAL DISPROPORTIONALITY BY COUNTRY

We are almost ready to create a graph with the index values by country. To do this, we make two adjustments to our previous SQL statement. First, we include the country name in the grouping, so that we know which country an election occurred in. This does not change our result, since a particular election is always linked to exactly one country. Second, we use only those elections from our table that were held after World War II. This is done by specifying a filter condition with the `WHERE` keyword we have used above:

```
dbGetQuery(db,
"SELECT
  election_id, country_name,
  sqrt(0.5 * sum(power(vote_share - seat_share, 2))) AS lsq_index
FROM elections
WHERE year >= 1946
GROUP BY election_id, country_name LIMIT 2")
election_id country_name lsq_index
1          429      Norway  4.062591
2          466      Greece  6.958688
```

This is the data that we need to generate our plot. For each country and each election, Figure 8.2 shows the disproportionality scores that were computed above. You can clearly see considerable differences

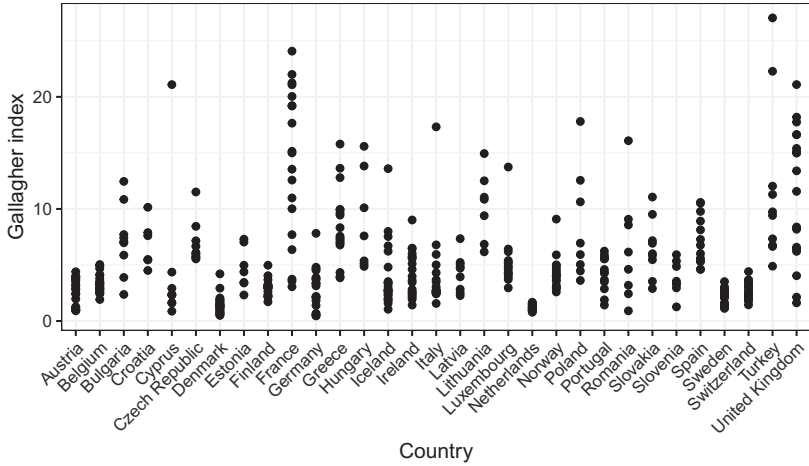


FIGURE 8.2. Gallagher index values for elections in different countries.

between countries when it comes to the disproportionality in the different elections. For example, some countries such as Sweden consistently have highly proportional outcomes, which means that the composition of the national parliament matches the distribution of votes across the different parties very well. This is not the case for other countries, however. The United Kingdom, for example, has several elections with high values of the Gallagher index. This is not surprising, given the electoral system in the UK: The country has a majority voting system, where in each district, only the candidate who gets most of the votes in that district wins. As a result, the votes for other parties are not reflected in the composition of the UK parliament.

Finally, once we are done working with our database, we need to close our database connection with:

```
dbDisconnect(db)
```

8.7 SUMMARY AND OUTLOOK

Database systems offer an alternative to standard, file-based data storage that is predominant in the social sciences. Databases are systems designed not just for data storage, but also for easy and efficient data manipulation and retrieval, possibly by several users in collaboration. They are often set up in a client-server fashion, where a central server keeps the data that

can be accessed from different clients over the network. In this chapter, we introduced a relational database system, which is the most frequently used type of database. At the core of a relational database is the table as the main structure to store data. The interaction with a relational database happens via the Structured Query Language, SQL.

We covered three different types of SQL commands. *Data definition* statements such as `CREATE TABLE` help us set up the structure of our database by defining the tables and their columns. *Data manipulation* statements allow us to populate our tables with data, and to change and delete it, and *data extraction* commands retrieve data from our database for further use (e.g., in R). In this chapter, we started our exploration into the world of database using a single table only. This is obviously too limited, which is why we will add more tables in the next chapter.

Besides the focus on SQL, there are several lessons you can take away from this chapter.

- *Client-server setups are useful for many applications:* As explained in the chapter, the client-server setup we use with PostgreSQL allows you to outsource particular tasks to other machines, and R simply interacts with these servers as a client. This is useful for many other applications beyond PostgreSQL, for example, other types of database servers, or servers executing large computing tasks. It is now possible to obtain access to these servers, for example, through universities, which means that you do not have to manage such a system yourself.
- *Recognize the difference between SQL and R:* Obviously, R and SQL are designed for different tasks, but there is an important difference in the philosophy underlying these languages. In R, you give the R engine a precise set of instructions on what it should do. This is called *procedural* programming. In SQL, you say what you want as a result, but not how to get there. This is called *declarative* programming. Declarative programming is convenient for us, since we do not have to worry about handling large amounts of data on a disk – the database system does this for us.
- *Distinguish between R's built-in database functions and SQL:* We have seen that R offers a number of convenient database functions that make your work easier, such as `dbWriteTable()`. These functions internally generate SQL code, which is then executed by the server. Of course, while this may be convenient for you, it also gives you less control over these operations. For example, you can bypass the explicit step of creating a table, and let `dbWriteTable()` do all the work. For this, it

is important to check the result in the database, for example, whether your automatically created table has the correct structure.

- *Different functions for sending commands and getting data:* It is important to remember the difference between data extraction (with `SELECT`) and the other operations that can be performed on the server. A `SELECT` statement returns data, the other statements do not. This is why there are two different functions in R for these different types of statements. `dbGetQuery()` is used only for data extraction, and requires an SQL statement that returns data (typically, `SELECT`). All other kinds of operation are done with `dbExecute()`.

Relational Databases and Multiple Tables

In the previous chapter of this book, I gave an introduction to a relational database system and the SQL language that we use to interact with it. We defined a new table, populated it with data, and extracted and aggregated the information contained in it. For illustration purposes, this introduction used a single table only; however, as I emphasized repeatedly, the power of relational databases lies in their ability to manage many different, interlinked tables simultaneously. This is why in this chapter, we are adding more tables to our database.

At this point, let us quickly go through the motivation again for distributing data across multiple tables. In Chapter 3, we discussed good and bad designs: Ideally, you should set up your tables such that they avoid data redundancy – each piece of information should be stored only *once* in the database. In our example about *elections* and the *parties* participating in these elections, how could redundancy possibly occur? Imagine for a moment that we were to store elections and parties in one table:

country_name	election_date	vote_share	party_name_short	family_name
Austria	1919-02-16	40.75	SPÖ	Social dem.
Austria	1920-10-17	35.99	SPÖ	Social dem.
Austria	1923-10-21	39.60	SPÖ	Social dem.

The first three columns in this table contain information about election results: The country they are held in, the date, and the vote share of the given party. The remaining two columns contain the party information: The short name, as well as the party family. This short example shows that we have redundant data: The short name and the party family are

repeated every time a party – in our case, the Austrian Social Democrats (SPÖ) – participates in an election. This is why the ParlGov project splits up their entire database into multiple tables. By separating data on election results from the data on political parties, we can reduce redundancy in the database. This is what our above example looks like in the actual ParlGov database: We have one table on election results (which is the one we used in the previous chapter):

country_name	election_date	vote_share	party_id
Austria	1919-02-16	40.75	973
Austria	1920-10-17	35.99	973
Austria	1923-10-21	39.60	973

and a second one on political parties (all shortened for presentational purposes):

party_id	party_name_short	family_name
973	SPÖ	Social dem.

By storing the party information in a separate table, we end up with *one* record for each party, rather than repeating this information for every election the party participates in. If we want to add new variables for parties (e.g., whether they have been coded as populist), we can do this by updating *one* row for each party. This facilitates the management of your data significantly and reduces errors. The above example also shows how we can link entries across tables: The parties results table has a `party_id` column, which we use in the elections table to identify the party that the given result belongs to. The use of these references is crucial, since we deal with different tables whose entries are linked to each other. In the world of relational databases, we often use integer numbers for this purpose. A unique identifier for a record in a table – such as `party_id` in the parties table – is called a *primary key*. A reference in a table that points to a record in a different table – such as `party_id` in the elections table – is called a *foreign key*. Much of the work we do below deals with these keys.

9.1 APPLICATION: THE RISE OF POPULISM IN EUROPE

In this chapter, we continue our work with election results, but extend it in a new direction. Over the recent decade, the Western world – and Europe in particular – has seen a strong rise in populism. Cas Mudde

defines populism as a political discourse or even an ideology based on the “relationship between the people (good) and the elite (bad)” (Mudde, 2004). In this chapter, we want to track the rise of populism over time. Specifically, we do this by measuring the electoral success of political parties that have been defined as “populist.” In this example, we do not differentiate between different types of populism, as for example, right- and left-wing populism – readers that are interested in only one or the other can easily modify the example.

For this exercise, we need two tables in addition to the election results we used in the previous chapter. So far, we only used data on elections from *ParlGov* to compute a Gallagher index of disproportionality. In the elections table, however, parties are only referenced with an internal identifier (the `party_id`), which is why we need to bring in a separate table on political *parties* to obtain the names of the parties as well as other information about them. Since our goal is to measure the success of populism by the vote share of populist parties, we need to know whether a party is considered a populist party or not. For this, we rely on the PopuList database, a list of populist parties in Europe (Rooduijn et al., 2019). As of Version 2.0, the PopuList dataset can easily be linked to parties from ParlGov: Each party in the PopuList has a `parlGov_id`, which corresponds to the `party_id` in ParlGov. Combining data from ParlGov and the PopuList ultimately allows us to track the success of populist parties over time in parliamentary elections.

9.2 ADDING THE TABLES

Let us now do some practical work to see tables and the references between them in action. Do not forget to create a new database, following the instructions in Chapter 2. We use the `dbadvanced` database for this chapter and connect to it:

```
library(RPostgres)
db <- dbConnect(Postgres(),
  dbname = "dbadvanced",
  user = "postgres",
  password = "pgpasswd")
```

We first add the elections table from the previous chapter, using the corresponding function from R’s DBI interface: `dbWriteTable()`. This function simplifies the import, since it automatically creates the table structure for us. This is convenient, but you have to make sure that the column types in the initial R data frame have the correct types, as they will be used to

specify the columns in the database table (see the previous chapter). We also add the year again as a separate column:

```
elections <- read.csv(file.path("ch09", "elections.csv"))
elections$election_date <- as.Date(elections$election_date)
dbWriteTable(db, "elections", elections)
dbExecute(db,
  "ALTER TABLE elections ADD COLUMN year integer")
dbExecute(db,
  "UPDATE elections SET year = extract(year from election_date)")
```

Our next step is to add the ParlGov table with political parties to our database, using again the functionality provided by R's DBI extension:

```
parties <- read.csv(file.path("ch09", "parties.csv"))
dbWriteTable(db, "parties", parties)
```

Since ParlGov does not provide information about whether a party is considered populist or not, we rely on the PopuList data described above. Before we can later merge this data to our parties table, we need to also import it as a table, using the file `populist.csv` in the repository for this chapter:

```
populist <- read.csv(file.path("ch09", "populist.csv"))
dbWriteTable(db, "populist", populist)
```

You should now have three tables in your database: the `elections` table from the previous chapter, and the `parties` and `populist` tables that we just created. Let us check if this is the case:

```
dbListTables(db)
[1] "elections" "parties" "populist"
```

The structure of the `parties` table should be obvious. Most importantly, as already mentioned above, each party has a `party_id`, which corresponds to the `party_id` in the `elections` table and helps us link each election result to the party it belongs to. This is similar for the `PopuList` table (or rather, the reduced version I have prepared for this chapter), where each party has a `parlGov_id`, along with information on whether it qualifies as a “populist” party according to the `PopuList` dataset, and whether it is considered to be a party on the far left or the far right:

```
dbGetQuery(db, "SELECT * FROM populist LIMIT 3")
```

	parlgov_id	populist	farleft	farright
1	1536	1	0	1
2	50	1	0	1
3	669	1	0	0

9.3 JOINING THE TABLES

Before we work with all three tables, let me demonstrate the linking of tables using the two tables from ParlGov only. To briefly repeat, we have a table with data on election results (`elections`), and a `parties` table with data on parties. Each entry in `elections` refers to a party from `parties` by means of a party identifier, called `party_id` in both tables. In the database world, this is called a “one-to-many” relationship between the two tables, since each party belongs to several election results – it usually participated in several elections. The combination of two tables that contain corresponding data is called a “join.” Joining two tables is a temporary operation – in contrast to a merge operation, we do not end up with a new, persistent table that contains the linked records. Rather, a join creates a *temporary* dataset with the corresponding records, which we can use for further data operations, or export for later analysis. The *storage* of our data, however, is still done in separate tables, which helps us avoid redundant data in our database.

So, how do we join tables in SQL? Again, we use a `SELECT` statement for this. All we need to change is the `FROM` part of the statement, such that it does not select from a single table, but from a set of two joined tables. This is indicated by the `JOIN` keyword:

```
dbGetQuery(db,
"SELECT
  elections.country_name, election_date, party_name_short, family_name
FROM elections JOIN parties ON elections.party_id = parties.party_id
LIMIT 3")
```

	country_name	election_date	party_name_short	family_name
1	Denmark	1915-05-07	RV	Liberal
2	Denmark	1953-09-22	GrFa	no family
3	Greece	1977-11-20	EDA	Communist/Socialist

It is not difficult to understand what this statement does: `elections` should be joined to `parties`, by linking entries where the `party_id` in

elections (which is a foreign key) corresponds to the `party_id` in `parties` (which is a primary key). It is not a requirement that the join attributes in the two tables have the same name, but we often follow this convention to make the relationship more obvious. The variables we select – the country name, the election date, etc. – are specified in the first part of the `SELECT` statement. Since `country_name` appears both in the `elections` and the `parties` table, we need to tell SQL which one we want, by specifying the name of the table before the name of the field (`elections.country_name`).

The type of join that is carried out with the simple `JOIN` keyword is called an *inner* join – in fact, you could write `INNER JOIN` instead and get the exact same result. An inner join links all pairs of entries from the two tables that have the same value in the join attribute. That is exactly what we want in the vast majority of cases. Although much less frequently used, there are other types of joins that retain *all* records from one of the tables, but only the matching records from the other (the `LEFT JOIN` and the `RIGHT JOIN`). Even though the join of the two tables is only temporary, we can use it in the `SELECT` statement as if it were a new, big table. For example, we can count the number of records:

```
dbGetQuery(db,
  "SELECT count(*)
  FROM elections JOIN parties ON elections.party_id = parties.party_id")
count
1 5247
```

Alternatively, we can run aggregations on it. Here is an example that makes use of the `party_family` variable contained in `ParlGov`: We compute the average vote share of social democratic parties per year, to see the ups and downs in their electoral success:

```
dbGetQuery(db,
  "SELECT year, avg(vote_share)
  FROM elections JOIN parties ON elections.party_id = parties.party_id
  WHERE family_name = 'Social democracy'
  GROUP BY year
  ORDER BY year
  LIMIT 3")
year      avg
1 1900 12.7500
2 1901 17.0600
3 1902  9.4025
```

In this statement, you recognize all the different parts of a data aggregation, as introduced in the previous chapter: the grouping variable `year`

(computed by extracting the year from the election date), and the aggregation function (the average over the `vote_share` values for a given year). Importantly, we filter out the social democratic parties with the `WHERE` keyword, since these are the parties we are interested in. Finally, we order the result by year, and truncate it for display purposes using the `LIMIT` keyword – if you would like to see the entire time series, just remove this last part of the statement.

9.4 MERGING DATA FROM THE POPULIST

In the previous section, we joined the elections and the parties tables. Joining means that the two tables are dynamically combined within a query, while the original data remains in separate tables. Is this what we should also do when linking parties from ParlGov with data on populist parties from the PopuList? We could do a simple join on the party identifier:

```
test <- dbGetQuery(db,
  "SELECT *
  FROM parties JOIN populist ON parties.party_id = populist.parlgo_id")
nrow(test)
[1] 199
```

As per the logic of an inner join, we only get the matching records from both tables – this is why the result of the join contains only 199 entries, which is a small subset of the almost 1,300 parties from ParlGov. It is easy to see why: Unlike ParlGov, which goes back more than a century, the PopuList covers only recent years. Also, it identifies only populist and eurosceptic parties, which is why it contains only a subset of recent parties.

Our parties table and the data from the PopuList are coded at exactly the same level – both contain information about political parties as unit of observation. In other words, the relationship between the two is a *one-to-one* relationship rather than the *one-to-many* relationship we have for parties and elections. While it is technically possible to use SQL joins whenever we want to combine information from two tables, in this case it may be more useful to merge the variables from the PopuList to our parties table. Again, merging means that we amend the parties table, such that it *persistently* stores the additional variables from the PopuList. We can then simply access the information about whether a party is considered as populist in the parties table, rather than having to join it with populist every time.

To merge the PopuList coding to the existing parties table, we first add a new column:

```
dbExecute(db, "ALTER TABLE parties ADD COLUMN populist integer")
```

The default value of this new column is `NULL` (the SQL value for missing data). We then use an amended version of an `UPDATE` statement, which uses a second table to update the values in the given table. More precisely, it links the two tables similar to a join, and copies the values of the `populist` variable from the `populist_parties` table to the `parties` table:

```
dbExecute(db,
"UPDATE parties
SET populist = populist.populist
FROM populist
WHERE parties.party_id = populist.parlgov_id")
```

Again, the logic of this statement is not difficult to understand. We update the `parties` table and want to set the values of the `populist` field to the corresponding ones from the `populist_parties` table. In the `WHERE` clause, we need to specify – similar to the join above – what attributes the two tables should be linked on. Importantly, this updates the values only for the parties contained in the PopuList data, because these are the only ones that can be matched. For all other parties, the default values (missing, or in the database terminology: `NULL`) remain.

With the new variable `populist` now being part of our table with political parties, we can modify the above aggregation query such that it counts, for example, the number of populist parties per year that participated in elections:

```
dbGetQuery(db,
"SELECT year, count(*) AS num_parties
FROM elections JOIN parties ON elections.party_id = parties.party_id
WHERE populist = 1 AND year >= 1998
GROUP BY year
ORDER BY year DESC
LIMIT 5")
```

	year	num_parties
1	2017	19
2	2016	18
3	2015	23
4	2014	13
5	2013	18

This statement is very similar to the one above, where we computed the average vote share of social democratic parties by year. We change the

aggregation function to output the count of elections, join the two tables as above, and restrict the combined result to parties that are populist (`populist = 1`) and elections in 1998 and later, since this is the first year for which there is data from the PopuList.

9.5 MAINTAINING REFERENTIAL INTEGRITY

When introducing relational databases, we discussed some of their advantages for data management and processing. One of them was that databases can help us avoid data redundancy, but at the same time ensure that our data remains consistent. For example, by splitting up the data on election results and the parties participating in these elections, we can avoid that information on parties is repeatedly stored every time a party participates in an election. Splitting data into several tables may be useful for eliminating data redundancy, but at the same time creates other problems. As we have seen above, every row in the `elections` table has a pointer to the corresponding row in the `parties` table. This is implemented by means of an integer number – `party_id` in `elections` points to the corresponding `party_id` in `parties`. The latter is a primary key in the `parties` table – a field that uniquely identifies an entry. The former is a foreign key in the `elections` table – a field that references an entry in another table.

Problems can now arise if the pointer to the entry in the other table is invalid – in our example, this would mean that we have a row with `party_id = 1556` in `elections`, but no corresponding entry with `party_id = 1556` in the `parties` table. In other words, we would have an election result for a party that does not exist in our database, and our data would therefore be inconsistent. In database terminology, this is called a violation of *referential integrity*. Referential integrity applies if every reference between tables is valid, that is, if it points to an existing entry in the respective table. Of course, we want referential integrity at all times, since otherwise we would have major gaps in our data – in this case, an election result we cannot link to a party. How can we ensure that errors of this kind do not arise?

At the moment, the database does nothing to help us address this challenge. We could, for example, delete any party from the `parties` table, leaving a number of invalid foreign keys in the `elections` table. Why? The reason is that our database does not “know” yet that one field in one table references entries in another table. Let us proceed step by step to define this relationship in SQL. First, we need to introduce `party_id` as a primary

key in the `parties` table. Again, a primary key is a field (or in some case, a combination of two or more fields) that uniquely identifies each line in the table. It is common practice to use positive integer values for this – luckily, we already have such a field in our table and only need to define it as primary key. We do this using the `ALTER TABLE` statement again, but this time without adding a new field:

```
dbExecute(db, "ALTER TABLE parties ADD PRIMARY KEY (party_id)")
```

When we define a primary key, the database does different things. Most importantly, it introduces logical checks, for example, by ensuring that no single value of the primary key occurs more than once. For example, try adding a new record with 1739 as the value for the primary key:

```
dbExecute(db,
"INSERT INTO parties (party_id, party_name_short)
VALUES (1739, 'New Party')")
```

This value already exists in the table, which is why PostgreSQL refuses to add the new entry. We get an error message telling us that the value 1739 already exists as a primary key.

Rather than using an existing field as primary key, you can also have the database create and maintain one for you. Simply add a new field of the type `serial`, and you will get an integer variable that automatically increments when new records are added to the table (you do not need to provide values for it). If you define this field as primary key, you never have to worry about duplicate key values anymore.

We now have a primary key for the `parties` table, and PostgreSQL ensures that the key does what it is supposed to do: uniquely identify parties in our database. The second step to have the database check and maintain referential integrity of our data is to define the `party_id` field in `elections` as foreign key. We again use an `ALTER TABLE` statement to do this:

```
dbExecute(db,
"ALTER TABLE elections
ADD FOREIGN KEY (party_id) REFERENCES parties (party_id)")
```

Using this statement, we tell the database that `party_id` in `elections` points to `party_id` in `parties`. This means that all party IDs used in

the elections table must be present somewhere in the parties table. Since PostgreSQL created the foreign key without any error messages, we know that this is the case. However, once we attempt to delete a party from parties, the database blocks this operation if this party is referenced from elections. Try this statement:

```
dbExecute(db, "DELETE FROM parties WHERE party_id = 1739")
```

Now, the database refuses to delete party 1739, since this would leave some election results without a corresponding party. So in essence, by specifying in our database which attributes are primary keys and foreign keys, the database helps us maintain the consistency of our data and ensures that referential integrity is not violated. Using these mechanisms, distributing data over multiple tables becomes much more manageable.

9.6 RESULTS: THE RISE OF POPULISM IN EUROPE

We can finally put our data together and create a dataset for our analysis of the rise of populism in Europe over time. In the following code example, we again join the parties and elections tables, the latter now amended with the PopuList coding. We aggregate the joined tables by country and election date, which allows us to plot the success of populist parties per country, as measured by the vote share in the respective election:

```
populism_ds <- dbGetQuery(db,
  "SELECT
    elections.country_name,
    election_date,
    sum(vote_share) AS total_vote_share
  FROM elections JOIN parties USING (party_id)
  WHERE populist = 1 AND year >= 1998
  GROUP BY elections.country_name, election_date
  ORDER BY country_name, election_date")
```

The plot in Figure 9.1 shows that in particular in Eastern Europe, populist parties have been gaining ground in the recent decade. In several countries, they now achieve vote shares of up to 50% and more.

As a last step, we need to close the connection to our database:

```
dbDisconnect(db)
```

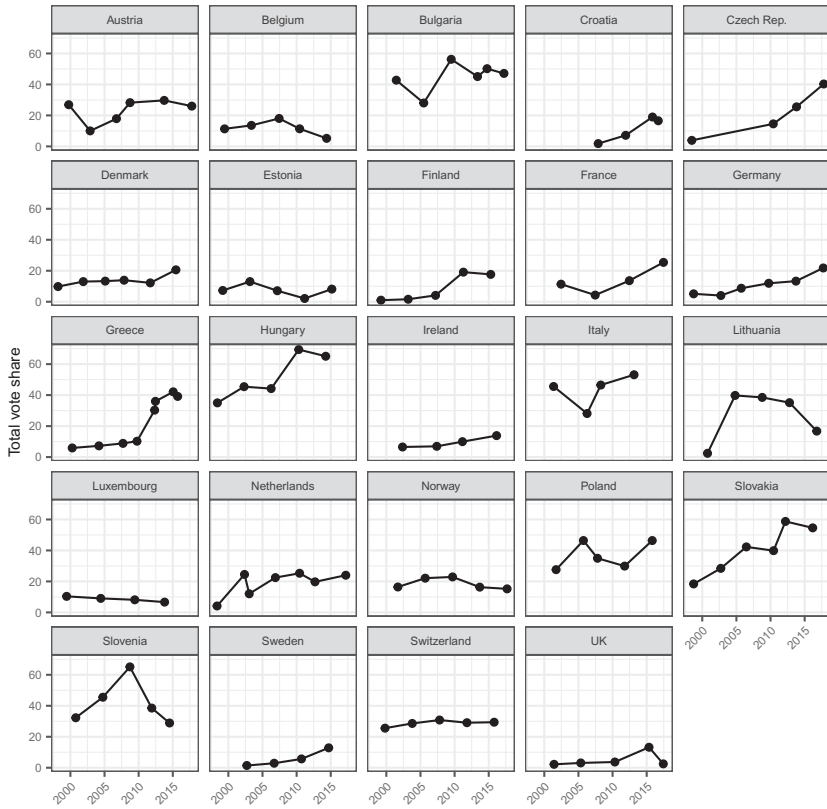


FIGURE 9.1. Vote shares of populist parties in different countries.

9.7 SUMMARY AND OUTLOOK

The art of working with relational databases necessarily involves multiple tables. In this chapter, we extended the single-table example from the previous chapter such that it uses two tables. More precisely, we added a second table with data about political parties to the existing elections table, such that we have more information about the parties themselves. We supplemented the latter table with data from the PopuList project, which identifies populist parties in Europe. Using our data, we were able to plot the electoral gains of populist parties in Europe over the recent years.

Spreading information out over several tables in a relational database involves different challenges. First, we need to think about the structure of our data: What tables do we need, and what variables are they supposed

to contain? These are conceptual questions about our database, and they relate directly to what we discussed in earlier chapters of this book (e.g., designing a database such that it avoids storing redundant data). When we use existing datasets, we often do not have a choice and have to use the data in the way it is provided to us. However, when designing databases for our own projects, taking some time to think about the data structure is important. Database designers have even developed an entire modeling approach for this purpose, which is based on the definition of real-world *entities* and the *relationships* between them. These “Entity-Relationship” models can then be used to define the actual tables in a relational database. For most applications in the social sciences, however, this conceptual step is not required, as the complexity of the data is limited.

Also, there are technical challenges we need to overcome when working with multiple tables. The first we discussed is the dynamic combination of data from different tables. While stored across multiple tables, matching entries from them can be joined in SQL to perform various tasks such as aggregation, or can be exported for analysis. Importantly, joins are dynamic, and the original data are still kept in their original tables. The second challenge the database can solve for us is to keep our data consistent across different tables. For example, if a table has a *foreign key* that refers to a *primary key* in another table, the database can make sure that corresponding entries for the latter exist in the second table. This way, we can automatically ensure *referential integrity* of the database and prevent operations that would violate it.

While we now know a lot about databases already, we still need to explore two more features that can be really useful for our work: the ability for multiple contributors to jointly work on datasets, and to quickly search large amounts of data. The next chapter addresses these two questions, and wraps up the basic introduction of relational databases in this book. Before we proceed, here are some recommendations from this chapter:

- *Think about the structure of your data:* This came up repeatedly in the book, and here it is again. Choosing a good structure for your database first requires a good understanding of what is in your data: What real-world entities are described, and what are their features? How do these entities relate to each other? Once you have answered these questions, it becomes easier to design a structure for your data.
- *All tables need a primary key:* For a smooth operation of a relational database, it is absolutely necessary to have sensible primary keys for

all your tables. A good choice is a single integer number. Some datasets already have a primary key, for others you can easily create one in your database with a `serial` field.

- *Make use of the integrity checks in a DB:* In the chapter, we saw how PostgreSQL can help you maintain referential integrity and make sure that the data is consistent across tables. I recommend using these features, in particular when your database becomes more complex. Without these checks, errors and missing data can occur without you noticing.
- *Merge only when you have to:* While joins are the standard operation to combine data from different tables in a relational database, it is also possible to merge tables by copying data from one to the other. This is something you should only do when it is really necessary, since it can violate the principle of avoiding redundant data.

Database Fine-Tuning

In Chapters 8 and 9, we covered the basics of relational databases – tables as the main containers of data, and how they can be created, populated, and joined with SQL. In these chapters, we dealt exclusively with conceptual questions about data and how it is stored in a relational database. In this final chapter on databases, we move on to two more operational questions. Recall that data structure (facilitating the use of multiple tables, avoiding redundancy) was only one of the reasons for storing data in a database. There are at least two more reasons that can make databases such as PostgreSQL a useful choice for social science projects. First, databases can handle large datasets much more efficiently than a file-based workflow, and second, databases permit data processing shared by multiple users, such that it is possible to give some users write access to certain parts of the data, while others can only read it.

Most database systems do not solve these issues automatically. Rather, they require some fine-tuning by the user, but fortunately, none of this is very complicated. To show how database systems such as PostgreSQL deal with these challenges, we cover two topics in this chapter. First, we discuss the use of so-called search *indexes* that allow the database to quickly look up particular entries in a table based on one or more of the fields. Indexes are not only used in relational databases; rather, they are data structures that you can find in many systems dealing with large amounts of data (although they are often hidden from the user). Second, we introduce PostgreSQL's multi-user capabilities, where you can add several users to a database and equip them with particular privileges for data access and data manipulation.

<i>Person No.</i>	<i>City</i>
18665	3
16889	8
17443	9
14662	8
16881	8

FIGURE 10.1. A table without an index.

In this chapter, we are not going to work with another real-world example to demonstrate indexes and multi-user features. Rather, we will use an artificial dataset, which makes it possible for us to create large tables without the need to import them from a file.

10.1 SPEEDING UP DATA ACCESS WITH INDEXES

An index is an additional data structure added to your database that allows the database system to quickly locate records in a given table, based on one or more of the fields in the table. You can think of database index very much like the index of a book: A book's index is essentially a list of important keywords and topics covered in the book, and it provides you with the page number(s) that contain relevant information about the respective topic. Here is an example that briefly illustrates how an index works. In Figure 10.1, you can see a table with data on persons located in three cities (3, 8, and 9). Now imagine that you want to select all persons in a particular city, for example, those in city 8. Without a search index, the database system needs to go through all records in the table sequentially, test whether the city attribute is equal to 8, and retain those where this condition is satisfied.

This would of course be a very fast operation for our small table, due to the fact that it only has five rows. However, as the size of the table grows, so does the retrieval time: In computer science, this time is typically measured in relation to the number of rows in the table. Our simple, non-indexed table requires retrieval times that scale *linearly* with the number of records: If we double the number of records, the retrieval time doubles, too. This becomes a real issue when we deal with large tables and require many repeated lookups. Luckily, we can solve this issue with the help of indexes. A search index is created to speed up the retrieval of records based on a particular search attribute. Figure 10.2 shows our example again, but with a search index on the city field added.

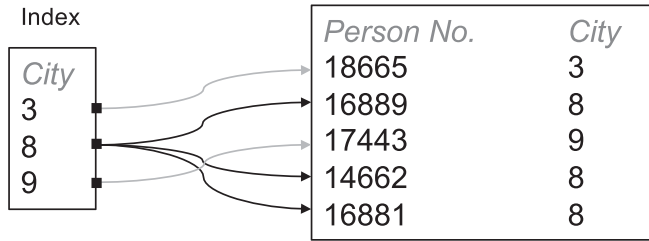


FIGURE 10.2. A table with an index.

The index stores pointers to the different values in the search attribute. So if we want to look up the persons living in city 8, all the database system needs to do is locate the value 8 in the index, and follow the pointers to the three records available for city 8. This is much faster compared to the simple traversal of the entire table as in our above example: The retrieval time using an index is usually *logarithmic* in the number of records in the table. This is a tremendous speedup: A table with 5 million entries would require 5 million steps for a naïve, sequential search, compared to less than 25 steps for a search using an index (binary logarithm).

This speedup, however, has certain costs. An index is an additional data structure that needs to be stored somewhere, so the size of your database on disk will become larger (which is something that, in most cases, you can simply ignore since the gains for retrieval are significant). An issue that may be more relevant arises when we insert new records into the table (or when we update the indexed column for some of the records): With an index in place, simply adding the new record to the table is not enough; the database system also needs to update the index such that it contains a pointer to the new record. If you insert many records, or update the existing ones frequently, this will become slower in comparison to a non-indexed table. In a typical workflow in the social sciences, however, this is less likely to happen. Therefore, you can usually create an index *after* all the data has been inserted, which avoids this problem.

Thankfully, indexing functionality for different kinds of data is readily built into PostgreSQL (and many other database systems), so as a user you do not have to worry about any of the inner workings. To demonstrate the speedup we can gain from an index, let us perform a little experiment. We create a large table both in R and in a relational database, and measure how long it takes for a certain subset of the table to be retrieved. For this experiment, we slightly expand the above example. We create a table with persons that are located in cities, and are observed annually

(e.g., in a longitudinal survey). The `expand.grid()` function is useful here, which creates a data.frame out of all possible combinations of the values in the given vectors. We add a randomly generated `result` variable to the data frame, which holds the value measured for the respective person in the respective year. Finally, we randomly shuffle the order of the entries in our table, to exclude any effects from our data having been inserted in a particular ordered sequence.

```
survey <- expand.grid(person = 1:100, city = 1:1000, year = 1970:2020)
survey$result <- runif(nrow(survey))
survey <- survey[sample(nrow(survey)), ]
```

We can now simulate a simple data retrieval operation from our table, where we extract all records for city 80 and the years 2000 and later. When we do this, we measure the time the system takes to carry out this task, by computing the difference between the system time immediately before the data retrieval and immediately after. In the following code chunk, we enclose the three lines of code in curly brackets, such that they are executed immediately after one another:

```
{start_time <- Sys.time()
nrow(subset(survey, city == 80 & year >= 2000))
extime_R <- Sys.time() - start_time}
```

The time used for extracting the relevant records is 52.09 milliseconds. On your system, this value will be different, since execution times depend on a multitude of factors; however, the precise value is not important since we are interested in relative differences between R and PostgreSQL. Now, let us do the same experiment in a relational database. As always, we create a new, blank database for this chapter (see Chapter 2 for instructions), and connect to it:

```
library(RPostgres)
db <- dbConnect(Postgres(),
  dbname = "dbtuning",
  user = "postgres",
  password = "pgpasswd")
```

We again create our artificial dataset, this time using an SQL statement. The `generate_series()` function in PostgreSQL makes the generation of the numeric sequences easy. All possible combinations of these values are generated by referencing them in the `FROM` part of the statement. Again, we add a `result` value randomly:


```
dbExecute(db,
  "CREATE TABLE survey AS
  (SELECT *, random() AS result FROM
   generate_series(1,100) AS person,
   generate_series(1,1000) AS city,
   generate_series(1970, 2020) AS year)")
```

This table has exactly the same content and size as the data frame we used above. At present, it is just a simple table in the database, without any indexes to facilitate data retrieval. This is how we perform the same query as above in SQL, again measuring the execution time (note again the curly brackets):

```
{start_time <- Sys.time()
dbGetQuery(db,
  "SELECT count(*) FROM survey
  WHERE city = 80 AND year >= 2000")
extime_pg1 <- Sys.time() - start_time}
```

Without an index, the search in our survey table takes longer than the one in the R data frame: 196.99 milliseconds. Clearly, without an index, R's basic data structures perform even better than a relational database. Does an index solve this problem? Adding an index to a table is easy. All you need is a `CREATE INDEX` statement, where you specify the table and the column that should be indexed. As a rule of thumb, it is advisable to index those columns that are typically used to retrieve records from the table, and to create a separate index for each of them. In our case, these are the city and year columns in the survey table. If you have a primary key in the table and it has been explicitly defined as such, there is no need to create an index, since PostgreSQL does this automatically:

```
dbExecute(db, "CREATE INDEX ON survey (city)")
dbExecute(db, "CREATE INDEX ON survey (year)")
```

If we now run our query again with the same statement as above:

```
{start_time <- Sys.time()
dbGetQuery(db,
  "SELECT count(*) FROM survey
  WHERE city = 80 AND year >= 2000")
extime_pg2 <- Sys.time() - start_time}
```

we see a considerable performance improvement. Now, the query only takes 1.79 milliseconds, which is much faster than the previous SQL query on the non-indexed table (by a factor of about 110), but also about

29 times faster than R's data frame. So if your data processing includes large datasets with many repeated data retrievals, it is absolutely essential that you properly index your data. If you do not need the additional features of a database system and use a purely file-based data workflow, you can consider using alternatives to R's basic data frames. For example, the `data.table` package provides a tabular data structure that can be indexed, and has a much better retrieval performance, in particular for large tables.

10.2 COLLABORATIVE DATA MANAGEMENT WITH MULTIPLE USERS

One of the main benefits of managing data in a centralized, server-based setting is the possibility for many users to access the database. Imagine a situation in which a team of researchers collaboratively works on a new dataset (thereby actively modifying it), and another group of researchers prepare initial analyses on this dataset (with read-only access to the data). In a file-based workflow, the second team of researchers could be provided with regular snapshots of the data, shared as files. However, it is difficult to collaboratively edit and update data in a team of contributors if the data is stored solely in files. The reason is that changes by one person can easily be overwritten by another person, similar to what happens if several people edit a single text document at the same time.

Relational database systems have fine-grained mechanisms of access control, which makes it possible to fine-tune read and write privileges for many users of a database. These features can be useful to resolve issues arising in collaborative scenarios such as the one above, but also many others. In this section, I present a brief introduction to user privilege management in SQL. We continue to use our survey table created above, but will simulate access to this table by two users. The first one of them uses the connection we have initiated above, with the standard username and password. This user is a "super-user," owns the database `dbtuning` and can make any modification in it. The connection object we have created is `db`, which we will continue to use. For our exercise, however, we also need a second user. Therefore, our super-user first needs to create this second user in the database system:

```
dbExecute(db, "CREATE USER other WITH ENCRYPTED PASSWORD 'pgpasswd1'")
```

Note that we create the user through the `db` connection, which is the super-user connection with the privilege to add and modify users. The new

user is called `other`, and we create this user with an encrypted password and not a plain text one (`WITH ENCRYPTED PASSWORD`). In PostgreSQL, users are defined at the level of the entire database server, not at the level of individual databases. Therefore, the new user is in principle available for all databases hosted on our server. At this point, however, there is not much this user can do, because no access privileges to databases and tables have been defined. Still, we can already connect to the server as the new user. We do so with a new connection object `db1` that we will use for all operations that user `other` will perform later.

```
db1 <- dbConnect(Postgres(),
  dbname = "dbtuning",
  user = "other",
  password = "pgpasswd1")
```

We are connected to the `dbtuning` database, so everything we send over the `db1` connection will be executed within this database. Let us try to select a few rows from the table:

```
dbGetQuery(db1,
  "SELECT avg(result) FROM survey
  WHERE city = 80 AND year = 2000")
```

As expected, this fails with an error message. The reason is simply that user `other` does not have any privileges for the database, which means that the user can neither read nor modify any data in it. Our super-user can enable this. The following statement (executed as user `postgres` through the `db` connection) allows user `other` to perform `SELECT` queries on the `survey` table:

```
dbExecute(db, "GRANT SELECT ON survey TO other")
```

Now, the user can successfully execute the above query:

```
dbGetQuery(db1,
  "SELECT avg(result) FROM survey
  WHERE city = 80 AND year = 2000")

  avg
1 0.5139814
```

In some cases, it may be necessary to let users update the data in a table. We can do this by granting update privileges for the entire table, but it is even possible to do this for individual columns only. Let us assume that the new user is supposed to update the survey results. We allow this by

providing the user with the right to update the table, but only a specific column. In the case of our survey, we can use this functionality to let other users change the measured value in the result column, but not the basic structure of the survey with person IDs, cities, and years:

```
dbExecute(db, "GRANT UPDATE (result) ON survey TO other")
```

Now, the user can change the results, for example to correct a wrong entry

```
dbExecute(db1,
  "UPDATE survey SET result = 0.789
  WHERE person = 3 AND city = 80 AND year = 2000")
```

but the user cannot update data in the other columns of the table, which is exactly what we want. The following command fails with an error message:

```
dbExecute(db1, "UPDATE survey SET year = year + 1 WHERE year = 2000")
```

We only granted the user the privilege to update some data, but other manipulations (such as dropping some records) are not possible. If we want our new user to update the table or delete records from it, we need to expand the user's privileges. You can either specify these new privileges explicitly (e.g., `GRANT UPDATE`), or simply give the user all privileges on the survey table.

```
dbExecute(db, "GRANT ALL PRIVILEGES ON survey TO other")
```

Finally, we also show how to remove certain privileges after they have been granted. This is done with a `REVOKE` statement, which works similar to the `GRANT` statement used above. Again, you can specify the privileges you would like to drop explicitly, or simply revoke them all:

```
dbExecute(db, "REVOKE ALL PRIVILEGES ON survey FROM other")
```

As always, at the end of the chapter, we close our database connections:

```
dbDisconnect(db)
dbDisconnect(db1)
```

The examples above gave you an idea of how to fine-tune user access to the tables in your database. Access control works at the level of tables. By default, users (that do not own the table) have no access to a table. You can change this by granting read-only access to the table, or allow

users to make changes to the data in the table (or only certain columns). These features allow for databases to be used in a collaborative setting, where multiple researchers jointly access a single database remotely.

10.3 SUMMARY AND OUTLOOK

Most of our discussion about databases in the previous chapters centered around questions of data structure and content. In this chapter, we looked into operational questions arising in the work with relational database systems. Databases shield a lot of technical complexity from users; all you need to do is define your tables and populate them with data, and the database system takes care of saving this data in a physical storage. While PostgreSQL and other systems have a lot of internal mechanisms to process the data as efficiently as possible, some fine-tuning may be necessary, depending on the context in which you use the database. I showed above how the retrieval of data in large tables can be sped up by several orders of magnitude through the use of indexes. Since these indexes also have certain costs (e.g., they make data insertions slower), the database system does not create them automatically, which is why you need to do this yourself using the respective SQL statements.

A second topic we covered in this chapter is the multi-user features of PostgreSQL. Due to the client-server setup of most database systems, it becomes possible for your data to be accessed by different people and from different places, something that is difficult and error-prone if your data is stored in files. However, collaborative access means that you need to think about who should get access to the data in the first place, and what the other users can do with the data: Are they supposed to only have read access, or can they even make modifications to it? The user privileges in PostgreSQL allow you to define this. You can create new users, and grant them permission to select data from a particular table, or modify (change and delete) it. These features make databases a powerful and convenient tool for storing and processing research data. Overall, there is a number of lessons learned in this chapter:

- *Index your large datasets:* When your datasets grow large, it is essential to work with indexes to speed up search and retrieval. As mentioned above, indexing features are not just available in relational databases; instead, this is a general technique to handle large amounts of data efficiently. You can also equip individual tables in R with an index using the `data.table` package, but of course without the other advantages offered by relational databases.

- *User privileges allow for transparent data access:* Larger projects in the social sciences can easily involve several collaborators accessing a database. The user privilege system provides a way to regulate access to your data, with different access levels for different people. In particular when it comes to sensitive data, it is essential to define who can see, update, or delete what kinds of data in your project.
- *Tracking changes remains difficult:* In many projects (e.g., those that involve human coding), it is often desirable to track changes to the data made by users. This is difficult, regardless of whether your data is stored in files or in a database. One way to do this is to keep regular copies (snapshots) of your data. Alternatively, in PostgreSQL, you can use the `pgaudit` extension, which logs all database operations to a logfile.
- *Direct access to the DB:* Sometimes, it is useful to quickly browse a database, for example, to check whether the data was imported correctly. For this purpose, you can use a graphical database client, such as the free *pgAdmin* tool (<https://www.pgadmin.org>), or the commercial, but highly recommended *Postico* software (<https://eggerapps.at/postico/>). Using these tools, you can browse a database, look at some records from a table, or even make small updates manually.

PART IV

SPECIAL TYPES OF DATA

Spatial Data

In all the previous chapters, we were dealing with the most common data format in the social sciences: tables. These tables usually contain numbers and text. We discussed how you can store these tables in files, read and process them in R, or use relational databases to manipulate data distributed over several tables. For some applications, however, we need to go beyond this simple model. There are special types of data for which the tabular data model is insufficient. In this part of the book, we take a look at three of them. This chapter introduces *spatial* data, which are observations that come with geographic coordinates. In other words, with spatial data, each observation has a particular location on the globe assigned to it. In later chapters, we will cover text as data, followed by the final applied chapter on network data.

11.1 WHAT IS SPATIAL DATA?

Spatial data are closely linked to the world of Geographic Information Systems (GIS), which is the software to collect, process, and analyze data with spatial coordinates. There are two major types of spatial data: *vector* data and *raster* data. In this chapter, we discuss only the former. You can think of a vector dataset as a table similar to the ones we have used so far, but where each row has some geographic information attached to it. Take a look at the example in Figure 11.1: You can see a standard table on the right, which contains information about cities. This is the same type of data model we have used so far.

However, in a vector GIS dataset, this tabular information (called the “attribute table”) is amended with spatial coordinates. As you can see

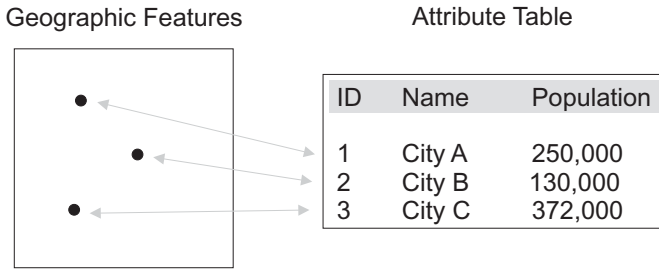


FIGURE 11.1. A table with cities (right), each of which is associated with spatial coordinates (left).

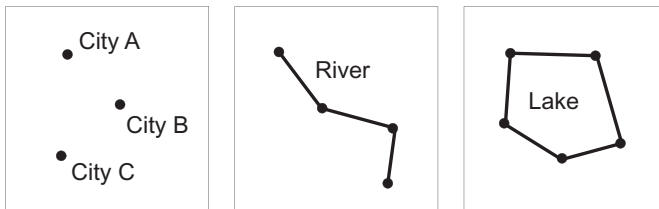


FIGURE 11.2. Different types of vector data.

in the example, each city in the table corresponds to a point on a map (left), which denotes the location of the respective city in the geographic space. Thus, a vector dataset is closely related to a standard table, with the only difference being that there is a new type of column with geographic information. This column contains what we typically call the *geometry* of a given entry. In our example above, this geometry is simply a two-dimensional *point*, in other words, a pair of (x, y) coordinates. However, GIS systems also allow more complex geometry types. Figure 11.2 shows a *line* geometry, which can be used to represent, for example, a river or a road. Finally, a *polygon* geometry is used to represent closed areas, such as a lake, or the borders of a country. A line is a sequence of (x, y) points, and a polygon is simply a closed line.

A question we cannot cover in depth is how we get from a three-dimensional surface (the earth) to a two-dimensional map, so I just convey some basic intuition here. There are two basic approaches to do this. The first one is to define a coordinate system for the (roughly spherical) surface of the earth (see Figure 11.3, left). This is what we do when using longitude and latitude coordinates: The equator has latitude 0, and locations north (south) of the equator have positive (negative) latitudes, each measured in radial coordinates. Longitude 0 is defined as going through

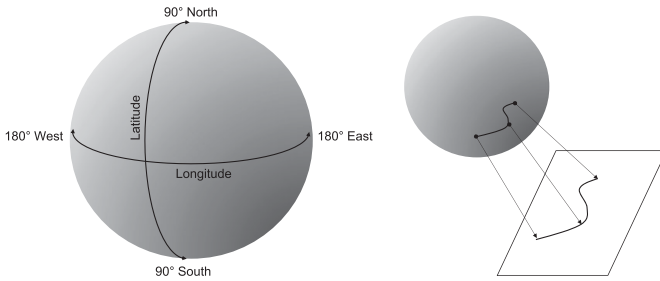


FIGURE 11.3. Coordinate systems for spatial data: A geographic (unprojected) coordinate system using radial coordinates on a spherical earth (left), and a projected coordinate system (right).

Greenwich, and longitudes increase as we move east, again measured in radial coordinates. Thus, each point on the globe can now be uniquely identified by its longitude (x) and latitude (y) values, which is called a *geographic coordinate system*. Importantly, we need to be careful what we can and cannot do with this system. Distance calculations, for example, can be tricky, since the distance on a sphere with radial coordinates cannot be computed similar to a planar surface.

The second approach to turn a three-dimensional surface of the earth into a two-dimensional map is to *project* it (see Figure 11.3, right). A projection is essentially an instruction for mapping points on the globe to corresponding ones on a map. There are many different ways for doing this, some of which are designed for particular purposes (e.g., they allow you to compute the distances between points correctly, thus avoiding problems such as the one described for geographic coordinate systems).

While the features of different projections are not important here, you need to keep in mind that the coordinate system and the projection of a geographic dataset are an important parameters you need to know when working with spatial data. Certain operations on spatial data (e.g., measuring the area of a polygon) only produce valid results if they are performed in a suitable projection. If you want to read up on this topic, I highly recommend the University of Boulder's *Earth Data Analytics* online course, which contains an entire chapter about spatial coordinate systems (Wasser, 2020).

GIS software is designed to handle spatial data – it allows you to read, modify, analyze, or visualize it. There are many different GIS systems available: *ArcGIS* is one of the most widely used commercial ones. If you want to try out an open-source GIS, I recommend *QGIS*, which is available free of charge at <https://www.qgis.org/> for all major operating

systems. In this chapter, we do not rely on specialized GIS software. Rather, we use extensions to the tools we introduced in the previous parts of the book: The R statistical software can in fact be turned into a powerful tool to process and analyze spatial data. Also, PostgreSQL has a spatial extension called PostGIS, which allows us to carry out spatial operations in SQL in combination with all the existing benefits of relational databases. However, before we discuss spatial data management with R and PostGIS, let me briefly introduce the applied example we use in this chapter.

11.2 APPLICATION: PATTERNS OF VIOLENCE IN THE BOSNIAN CIVIL WAR

In the practical exercises for this chapter, we examine a dataset about violent incidents in the civil war in Bosnia (1992–1995). The break-up of the Yugoslav Federation went along with an outbreak of violence between the three major population groups. Much of this violence happened in the (now independent) republic of Bosnia and Herzegovina, which is what we focus on in this chapter. Specifically, we will analyze the distribution of violence in Bosnia over space, to identify where it was most severe. Although we will not do a full explanatory analysis to study the drivers of this violence, the approach we present here is usually similar for any kind of statistical analysis that involves spatial data.

We use data on violent events from the *Geo-referenced Event Dataset* (GED), collected and maintained by the Uppsala Conflict Data Program (Sundberg and Melander, 2013; Högladh, 2019). The GED is part of a family of datasets related to political violence, and the current version of the data as well as much additional documentation and information can be found on their website. The GED is an *event dataset*, which means that it provides information at the level of individual incidents. In the case of the GED, each of these incidents is a violent, lethal confrontation between two of the conflict parties. The dataset records the date of the incident, the actors involved, the number of casualties, as well as several additional variables. The following is a (shortened) single entry from the dataset:

id	side_a	side_b	source_article	date_start	longitude	latitude
200416	Gvt. of BH	Civilians	BBC Monit.	1993-10-26	17.28	44.55

Each incident has an `id`, and it identifies the participants in the event (`side_a` and `side_b`). In the above example, the dataset records

an incident of violence against civilians, perpetrated by government forces. This information was obtained from an article published in BBC Monitoring (`source_article`). The incident took place on October 26, 1993, at the location with the given geographic coordinates (`longitude` and `latitude`). As we can see from the latter coordinates, each entry in the GED corresponds to a point on the map, so we can treat the entire collection as a GIS vector dataset in our application below.

To examine the spatial pattern of violence, we use Bosnia's pre-war administrative divisions. The smallest administrative unit was the municipality (*opština*), and Bosnia had 109 of them (with the capital Sarajevo divided into five municipalities). Our goal in this exercise is to compute the level of violence for each of the municipalities over the course of the war. Using administrative divisions as spatial units of observation is only one way to conduct a spatial analysis. Although we do not do this here, you could use this approach to relate an outcome we want to explain (in our case, violence) to particular socio-demographic variables measured at the level of administrative divisions, for example the ethno-national composition of a municipality (Weidmann, 2011). Alternative approaches for spatial analysis include the use of artificial grid cells as a unit of observation (Tollefsen et al., 2012), or no fixed spatial unit at all, as in point process models (Warren, 2015).

Our task in the analysis below is, therefore, to combine the vector dataset of violent events (points) with a dataset of Bosnia's municipalities, which are represented by polygons. We do so by identifying those violent events that took place within a municipality. In other words, we use the spatial coordinates of events and municipalities to link them to each other. In the GIS world, this is called an "overlay" operation, but we can also refer to it as a "spatial" join – the joining of data based on a spatial relationship (in our case, a point being located in a polygon). In the next section, we process spatial data using R and some extension packages, before introducing the PostGIS spatial database as an alternative workflow.

11.3 READING AND VISUALIZING SPATIAL DATA IN R

While most spatial data are usually processed in specialized GIS systems, R has grown into a powerful and versatile GIS itself, due to the development of new extension libraries. In this section, we use R's *Simple Features* (`sf`) library, a relatively new generic library for vector data. The term "feature" is often used in the GIS world to refer to the computational

representation of real-world objects, such as houses or lakes. The `sf` is the latest addition to R's spatial libraries and will likely supersede older ones, such as the `sp` package. As always, to use the functionality of the package, we have to load it first:

```
library(sf)
```

As described above, a vector dataset is essentially a data table with a new type of column that contains the spatial representation of the respective entry. This column is referred to as its geometry, and geometries can be points, lines, or polygons. The `sf` package follows exactly this approach. It uses R's standard data frames, but adds a new column type for geometries. So in other words, a spatial dataset in `sf` is just a regular table with a specific column for spatial information. Let us take a look at how to create such a table in R. We first load the GED data on violent events. This dataset comes as a regular CSV file:

```
events <- read.csv(file.path("ch11", "ged.csv"))
```

I removed many columns from the dataset to make the exercises below easier to follow. Compared to the original version, the reduced dataset contains only events for Bosnia, and only those events for which the exact location is known. Also, the dataset has a limited set of columns, namely, those with the unique ID of each event, the date it occurred, the location of the event (stored in the `longitude` and `latitude` columns), and the number of casualties (the best estimate provided by the GED):

```
summary(events)
```

	id	date_start	latitude	longitude
Min.	:199077	Length:1136	Min. :42.71	Min. :15.78
1st Qu.:	199690	Class :character	1st Qu.:43.85	1st Qu.:18.10
Median :	200136	Mode :character	Median :43.85	Median :18.38
Mean :	200106		Mean :44.11	Mean :18.16
3rd Qu.:	200503		3rd Qu.:44.54	3rd Qu.:18.38
Max.	:200874		Max. :45.19	Max. :19.54
	best			
Min.	: 0.00			
1st Qu.:	1.00			
Median :	3.00			
Mean :	17.29			
3rd Qu.:	6.00			
Max.	:8106.00			

For now, R treats the `events` table as a regular data frame and does not know that each entry has spatial point coordinates attached to it.

Therefore, we need to “spatially enable” the dataset, which we can simply do by converting it to a spatial object of type `sf`. The `st_as_sf()` function takes a regular data frame, and requires you to specify the names of the columns where the spatial coordinates are stored. In addition to the names of coordinate columns, we need to specify what spatial reference system is used for the dataset.¹ Longitude/latitude coordinates indicate a geographic coordinate system (see Figure 11.3), which has the ID 4326. If your data uses a different reference system or if it is projected, it is important to correctly specify the coordinate system here:

```
events <- st_as_sf(events, coords = c("longitude", "latitude"), crs = 4326)
```

Now you will see that `events` is no longer just a data frame, but much more:

```
print(events, n=2)

Simple feature collection with 1136 features and 3 fields
Geometry type: POINT
Dimension: XY
Bounding box: xmin: 15.78194 ymin: 42.71194 xmax: 19.53556 ymax: 45.18944
Geodetic CRS: WGS 84
First 2 features:
  id date_start best geometry
1 199885 1993-02-01 6 POINT (18.80833 44.87278)
2 199767 1994-03-03 1 POINT (15.91861 44.84444)
```

What do we see in this output? Our dataset contains a total of 1,136 features, each of which is a conflict event. The data uses *points* as geometries, in a two-dimensional space (hence the dimension `XY`). We also see the overall spatial extent of the dataset, referred to as its “bounding box” – this is the rectangle defined by the minimum and maximum coordinates along the `x` and `y` axes.

`sf` provides plotting functions specifically designed for spatial data. The easiest approach is to plot only the events as points. This is done by extracting the geometry from the dataset using the `st_geometry()` function, and by sticking it into the plot function as follows:

```
plot(st_geometry(events))
```

You can see the result in Figure 11.4. However, often we may want to color/style the plotted features according to some quantity associated

¹ Spatial reference systems were defined by the *European Petroleum Survey Group* (EPSG); online catalogues can be accessed, for example, at <https://spatialreference.org> or <https://epsg.io>.

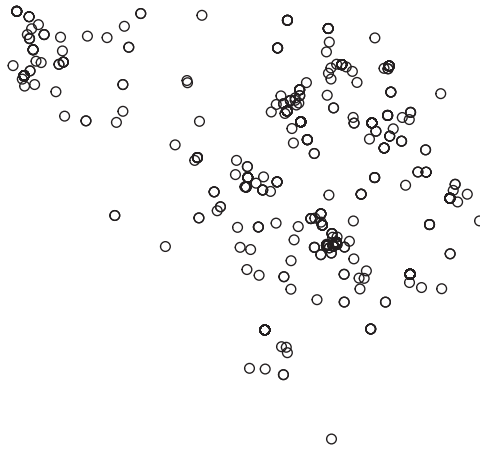


FIGURE 11.4. A simple plot of the conflict events from the GED.

with them. For example, we could color the dots according to the severity of the event, or according to the year in which they took place. In `sf`, this can be done by subsetting the data to the variable you want to use for coloring, and using again the `plot()` function, as for example in `plot(events["best"])`. There are many other options for tuning the plotting of spatial features, and you should consult the `sf` manual if you are interested in learning more.

We have now imported the events data from a CSV file, and converted it to a spatial dataset with point geometries. The second dataset we need contains the municipalities for our spatial analysis. The borders of each municipality are stored as a polygon, which is why we cannot simply store their coordinates in two columns of a CSV table. Instead, the dataset of municipal borders is provided in the *shapefile* format, an old legacy format for GIS vector datasets. As you can see in the data repository, a shapefile actually consists of at least three files with the same name, but different endings (`.shp`, `.shx`, and `.dbf`). Due to the fact that this format is still widely used, all GIS tools including `sf` are able to import it. When we do this, we have to manually set the coordinate reference system to the standard longitude/latitude system using the `crs` parameter, as above:

```
municipalities <- st_read(file.path("ch11", "bosnia.shp"), crs = 4326)
Reading layer `bosnia' from data source
  `Users/nilsw/Books/DataManagement/dmbook/ch11/bosnia.shp'
  using driver `ESRI Shapefile'
Simple feature collection with 109 features and 2 fields
```

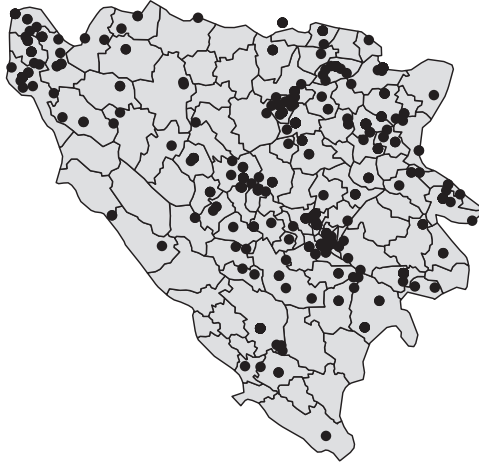



FIGURE 11.5. Municipality boundaries and conflict events.

```

Geometry type: POLYGON
Dimension:      XY
Bounding box:  xmin: 15.74059 ymin: 42.56583 xmax: 19.61979 ymax: 45.26595
Geodetic CRS:  WGS 84

```

With both the events and the administrative borders imported in R, let us take a quick look at a combined plot. For the map in Figure 11.5, we first plot the underlying municipalities, and then place the events on top of them with the `add=T` parameter:

```

plot(st_geometry(municipalities), col = "lightgrey")
plot(st_geometry(events), pch = 16, add = T)

```

We see that there are some areas with lots of events, while others experienced no violence. Still, this is difficult to tell exactly: Events occurring at the same location have the same coordinates and are therefore plotted on top of each other, which makes it impossible for us to keep them apart. Therefore, we proceed with our exercise and count the number of events per municipality, which helps us gauge the spatial distribution of violence over the entire country.

11.3.1 Overlaying Different Spatial Datasets in `sf`

Rather than just plotting our two datasets – the municipalities and the events – on top of each other, we now need to link events to their respective

municipalities, such that we can count them. Before we do so, a short comment on terminology is in order. In the GIS world, a spatial dataset used in a project is typically called a *layer*. In our example, therefore, we have two layers – a layer of municipality boundaries, and a layer of events. Linking the events to the municipalities based on their location is an example of what is called an *overlay* operation in GIS terminology. However, from a relational database perspective, we can also think of these layers as *tables with spatial coordinates*. This way, an overlay operation is equivalent to *joining* tables based on location: Rather than linking records based on a common attribute (which is what a regular join does), we link them by location. In our example, we want to combine each event with the municipality it occurred in. In other words, an overlay of this kind is simply a *spatial join*, and I now demonstrate how this is done in R and sf.

The `st_join()` function is used to carry out a spatial join, so let us take a look at what it does:

```
joined <- st_join(events, municipalities)
print(joined, n = 2)
```

Simple feature collection with 1136 features and 5 fields
 Geometry type: POINT
 Dimension: XY
 Bounding box: xmin: 15.78194 ymin: 42.71194 xmax: 19.53556 ymax: 45.18944
 Geodetic CRS: WGS 84
 First 2 features:

	id.x	date_start	best	id.y	name	geometry
1	199885	1993-02-01	6	115	Breko	POINT (18.80833 44.87278)
2	199767	1994-03-03	1	26	Bihac	POINT (15.91861 44.84444)

A spatial join is very similar to a regular, non-spatial join: It links the records of one table to the corresponding records from another table. So for each conflict event, the function appends the attributes of the corresponding municipality the event is located in. In the above example, the event with ID 199767 occurred on March 3, 1994 in municipality 26, which is Bihac. The joined dataset is again a spatial one, as you can see from the geometry column, since it retains the spatial coordinates of the first dataset (the events).

Using the `st_join()` function in this way hides much of the power and complexity of spatial joins. Without an additional specification, layers are joined based on intersecting geometries; that is, an event is linked with

a municipality if its geometry *intersects* with the municipalities geometry (the polygon). This does exactly what we want. However, things can become more complicated with different types of geometries or different relationships between them. For example, we can spatially join tables if geometries touch (but not intersect) each other, or if they are located within a certain distance to each other. To illustrate the basic idea of a spatial join, however, we do not go into more detail here.

We have now joined the two layers, such that each event is linked to the corresponding municipality. To create a map of the intensity of violence across Bosnia, there are two steps left to do: First, we need to aggregate the joined datasets by municipality and count the number of events for each of them, and, second, we need to append this information to our original dataset of municipalities, so that we can plot it as a map. Let us start with the first step. For convenience, we use the tidyverse approach for aggregation and merging (see Chapter 7), but it is of course also possible to do this in base R:

```
library(tidyverse)
eventcounts <- joined %>%
  as.data.frame() %>%
  group_by(id.y) %>%
  count(name = "num_events")
print(eventcounts, n = 3)

# A tibble: 79 x 2
# Groups:   id.y [79]
#   id.y num_events
#   <int>     <int>
1     1         1
2     2         7
3     4         1
# ... with 76 more rows
```

The above code first converts the joined spatial dataset to a regular, non-spatial data frame with `as.data.frame()`, since we no longer need the geometries of the municipalities. Using the standard tidyverse approach, it then groups the data using the `id.y` variable (which is the municipality identifier), and counts the records for each of them. This way, we get a list of municipality IDs (`id.y`) with the number of events that occurred in these municipalities. It is important to notice that the municipalities that do not contain any events do not show up in this list. `eventcounts` only contains event counts for 79 municipalities, which means that

30 out of 109 municipalities did not experience any conflict events according to the GED.

What is left for us to do is to merge this information with the existing municipalities dataset. Here, we need to be careful, since the data frame with the event counts is much shorter than the complete list of municipalities. Therefore, we use a left join, which preserves the entire list of municipalities:

```
municipalities_sf <- municipalities %>%  
  left_join(eventcounts, by=c("id" = "id.y"))
```

At the end of the chapter, we will use this new dataset to plot the distribution of violence across the different municipalities in Bosnia.

11.4 SPATIAL DATA IN A RELATIONAL DATABASE

In the first part of this chapter, we relied on spatial data stored in files, which we then imported in R for processing. This file-based approach is only one way to work with spatial data. Similar to relational databases for non-spatial data, we can also use these databases to store and process data with spatial coordinates. There are several advantages of the latter, for example, a more efficient processing of large datasets, but also concurrent access to the data by multiple users.

In this chapter, we rely again on the PostgreSQL relational database system. As you know, PostgreSQL is a great tool to work with tabular data – this is something we discussed at length in the previous chapters. As it turns out, however, PostgreSQL can also process spatial data, once we enable the PostGIS spatial extension. The combination of PostgreSQL/PostGIS (which from now on, we simply refer to as PostGIS) then becomes a powerful spatial database that is an ideal tool for more complex spatial data operations. You interact with PostGIS in the same way as we did with PostgreSQL alone. This means that you need to have the database server running and set up a new database (which we call `spatialdata`) for this chapter, as described in Chapter 2. We then connect to our database exactly as we did in the previous chapters:

```
library(RPostgres)  
db <- dbConnect(Postgres(),  
  dbname = "spatialdata",  
  user = "postgres",  
  password = "pgpasswd")
```

We now have a blank PostgreSQL relational database that keeps data in tables. For us to be able to work with spatially referenced data, however, we need to spatially enable our database by switching on the PostGIS extension:

```
dbExecute(db, "CREATE EXTENSION postgis")
```

Let us check if PostGIS is working correctly. The following code should output a single line similar to what you see below, which indicates the PostGIS version installed on your system:

```
dbGetQuery(db,
  "SELECT name, installed_version
  FROM pg_available_extensions
  WHERE name = 'postgis'")
  name installed_version
1 postgis          3.1.5
```

We are now ready to import the spatial datasets into PostGIS. Again, we use the conflict events and the municipalities data for Bosnia that you are familiar with from the exercises above. Let us start with the events. We first read the CSV file and write it as a simple table in the database:

```
events <- read.csv(file.path("ch11", "ged.csv"))
dbWriteTable(db, "events", events)
```

This step is exactly the same as for a non-spatial table. So far, our table is not yet “spatially enabled,” which means that PostGIS does not know yet that each event is actually associated with a point with x and y (or rather, longitude and latitude) coordinates. This is why, similar to the R-based workflow above, we need to explicitly create a column in the events table for the spatial location associated with each row. As with the *sf* package for R, this column is called a *geometry* column, and we create it using an ALTER TABLE statement. Recall that we have used different column types in PostgreSQL before, such as integer for numbers or varchar for text (see Chapter 8). With PostGIS enabled, we can now define columns of type geometry. For a geometry column, you need to specify the type of the geometry (a point, a line or a polygon), as well as the coordinate reference system (the EPSG ID we used above):

```
dbExecute(db,
  "ALTER TABLE events ADD COLUMN geom geometry(point, 4326)")
```

We can now use the data in the `longitude` and `latitude` fields of our table to update the geometry column with newly created point geometries. Here, the `st_point()` function creates the point, and the `st_setSRID()` function defines the spatial coordinate system (which you already know from the previous section):

```
dbExecute(db,
  "UPDATE events
  SET geom = st_setSRID(st_point(longitude, latitude), 4326)")
```

Done! Our `events` table now has the spatial coordinates of all events stored in a new column, which we can later use for spatial computations. Next, we need to import the municipalities to our database. Here, we follow a slightly different approach. We first import the shapefile using the `st_read()` function from the `sf` package (see above), rename the geometry column to `geom` for consistency, and then send the (spatial) table to PostGIS. The latter is done using the `st_write()` function from `sf`, which is essentially the spatial equivalent to the `dbWriteTable()` we used in previous chapters. The function requires you to specify the database connection (this is the `db` object), as well as a name for the layer you would like to create in the database.

```
municipalities <- st_read(file.path("ch11", "bosnia.shp"), crs = 4326) %>%
  rename(geom = geometry)
st_write(municipalities, dsn = db, layer = "municipalities")
```

Let us count the records in our two spatial tables to make sure that the import was successful:

```
dbGetQuery(db, "SELECT count(*) FROM municipalities")
  count
1  109

dbGetQuery(db, "SELECT count(*) FROM events")
  count
1 1136
```

11.4.1 A Spatial Join with PostGIS

The two tables were correctly imported: We have 109 municipalities and 1,136 events in our database. We can now proceed to spatially join them, using the geometry columns from the two tables. Let us recall first what a join of two tables does: It links the records of one table to those of another, based on a defined relationship between attributes of the two tables. In a conventional join, we usually require that an attribute from one table

be equal to the attribute from another table. For example, in the previous chapter, we joined parties to elections based on the party ID attribute.

In a spatial join, we no longer match records based on common attributes, but based on their spatial coordinates. Specifically, we want to join an event with a municipality if the former *is contained in* a given unit. Hence, we simply replace the join condition in a conventional join (which usually requires two given attributes to be equal) with a spatial condition (namely, that one geometry is contained in another). Let us take a look at how this is done in the context of a SELECT statement:

```
dbGetQuery(db,
"SELECT municipalities.id, events.id
FROM municipalities JOIN events
  ON st_contains(municipalities.geom, events.geom)
LIMIT 3")
  id id..2
1 115 199885
2  26 199767
3  70 200575
```

This query demonstrates how we join the two tables based on their geometries. The basic structure of this query should be familiar: We specify what fields we want to see (in our case, the IDs of the municipalities and the corresponding events), and which tables we want to select from. Here, the JOIN keyword is used to link municipalities and events. The part that is new is the join condition. Here, we use the PostGIS function `st_contains()`, to require that we only want to retain those pairs of records where the municipality geometry (which is a polygon) contains the event geometry (which is a point).

So each entry we see in the output above is a pair of municipality ID and event ID that satisfies the join condition, that is, where the municipality contains the event. Since the municipalities are non-overlapping, each point can be linked with at most one municipality, so the maximum number of records this table can have is 1,136 (the number of events). Joining municipalities and events is only the first step. Again, as in the R-based example above, we need to aggregate this table such that it counts the number of events per municipality. You should be familiar with SQL's aggregation – all we need to do is specify the aggregation function (`count(*)`) as well as the grouping level (`GROUP BY`):

```
dbGetQuery(db,
"SELECT municipalities.id, count(*) as num_events
FROM municipalities JOIN events
  ON st_contains(municipalities.geom, events.geom)
```

```
GROUP BY municipalities.id
LIMIT 3")

id num_events
1 1 1
2 2 7
3 4 1
```

We now have the information that we want – for each municipality, we have the number of conflict events that occurred there. The final step is to add this information to our `municipalities` table, such that we can access it along with the existing data we have on municipalities. For this, we first add a new column to this table, which we later use to store the event counts:

```
dbExecute(db, "ALTER TABLE municipalities ADD COLUMN num_events integer")
```

Recall that in our above example, we first stored the event counts in a separate table, which we later merged with the main `municipalities` data frame. In SQL, we can do something similar. However, rather than creating a new table for the event counts that we later have to delete again, we use a “temporary” table within our statement. This table is created on the fly as we run the query, but exists solely for the purpose of this query and is later deleted. You can define such a temporary table using the `WITH` keyword. In the statement below, we define a temporary table called `eventcounts` using exactly the same SQL code as in the previous example. We then use this table in an `UPDATE` statement, where we link it to the main `municipalities` table using the municipality ID:

```
dbExecute(db,
"WITH eventcounts AS (
  SELECT municipalities.id, count(*) as num_events
  FROM municipalities JOIN events
  ON st_contains(municipalities.geom, events.geom)
  GROUP BY municipalities.id)
UPDATE municipalities
SET num_events = eventcounts.num_events FROM eventcounts
WHERE municipalities.id = eventcounts.id")
```

Only 78 municipalities are updated, since the others do not contain any events. Finally, we export the `municipalities` table as a spatial dataset to R, such that we later draw a map of the violence in Bosnia. The `st_read()` function can not just read data from files (as above), but also from a database connection. Once we have done that, we can close the database connection.


```
municipalities_pg <- st_read(dsn = db, layer = "municipalities")
dbDisconnect(db)
```

11.5 RESULTS: PATTERNS OF VIOLENCE IN THE BOSNIAN CIVIL WAR

We computed event counts per municipality in two ways: First, using the `sf` package for R and, second, using the PostGIS spatial database. This gives us two spatial datasets: `municipalities_sf` is the result of the former approach, while `municipalities_pg` is the result of the latter. Both have the same structure: Each entry corresponds to a municipality, and the `num_events` column contains the event count for the respective municipality, with NA values for those municipalities without events. We can now use either of these datasets to draw a map of the distribution of violence in the Bosnian war as in Figure 11.6.

This maps shows us the municipalities that were hit hardest by violence in the civil war, as measured by the number of events. Part of the city of Sarajevo (displayed in black) experienced most of the attacks, but there are other areas in the north and the northwest of the country for which the GED recorded many conflict events. Of course, we can debate whether

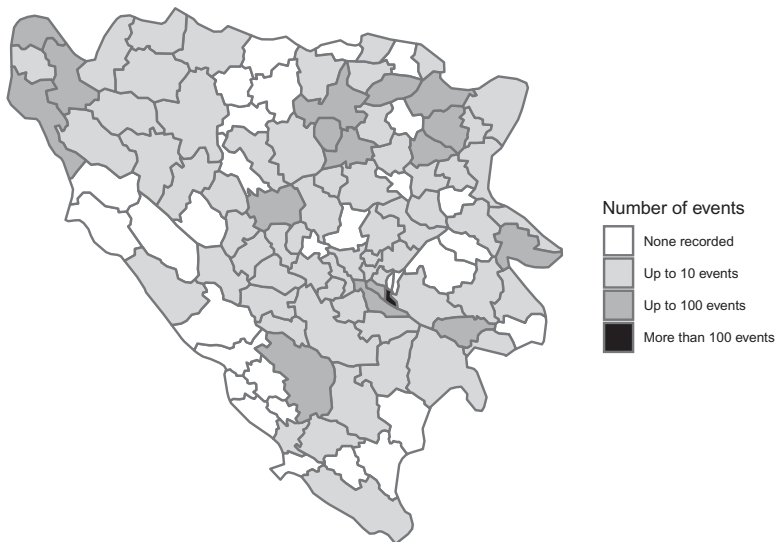


FIGURE 11.6. Civil war violence in Bosnia, as measured by the number of conflict events in the GED.

counting the number of events is a valid approach to approximate the patterns of violence. Alternative ones are possible, for example, by using the GED's casualty statistics.

11.6 SUMMARY AND OUTLOOK

Spatial data are empirical observations tagged with geographic coordinates, such that they can be assigned to particular places on Earth. In this chapter, we have focused on vector data, which can be of different types: *points* to represent single locations, *lines* for rivers or roads, and *polygons* for political units. A GIS vector dataset usually consists of a set of geographic features, each of which is linked to additional data contained in an attribute table. When working with vector data in R or GIS, we use an amended version of the standard tabular data structure with special column types, namely, those that store the corresponding spatial features.

We discussed how to create these spatially enabled tables both in R (using the *sf* package), but also in PostgreSQL's PostGIS extension. For the exercises, we used two spatial datasets: a point dataset of violent events in the civil war in Bosnia, and a polygon dataset of municipality boundaries. With these two datasets, we performed a spatial overlay operation, where the points are superimposed on the boundaries to find out which events occurred in each municipality. In database terminology, this is a *spatial join*, where we link entries from two tables based on a spatial relationship they have (in our case, whether a point is located within a polygon).

In this and the following chapters, we use both file-based and database-driven workflows to process our data. You are now experienced enough to decide whether one or the other is more suitable for your project: Using R and its extension packages to process data stored in files is easier as regards the technical infrastructure you need, but may not be ideal for projects involving several collaborators and/or large datasets. Spatial operations can be time-consuming, which is why it is often useful to perform them in a spatial database such as PostGIS. In particular, while not necessary in our above example, you can use indexing as described in the previous chapter also for spatial columns, which, in many cases, will give you significant performance improvements. Beyond the topics we covered in the chapter, here are some suggestions that can be helpful to you:

- *Explore the world of spatial data further:* If this chapter sparked your interest in spatial data, there is much more to explore. In our examples, we only used a limited set of operations with one type of spatial data – vector data. In particular in combination with raster data, there are many other useful applications for spatial analysis in the social sciences, and other, more focused introductions can help you make progress (see, for example, Lovelace et al., 2019).
- *Visualize your spatial data whenever possible:* When working with spatial data, it is often useful to plot your data. Looking at a map helps you understand your data better and lets you identify errors or artifacts that would otherwise be difficult to spot. This is why I recommend that you visually explore your data, using R's plotting features or an interactive GIS program such as QGIS (see next point).
- *Use QGIS to easily browse GIS datasets:* Oftentimes, it is useful to take a look at a spatial dataset to explore its structure or browse its contents. For this purpose, a graphical GIS can be useful. QGIS is a powerful, open-source GIS tool, which is available free of charge for all major operating systems from <https://www.qgis.org/>. With QGIS, you can even connect to a PostGIS spatial database and explore the different spatial tables visually.
- *Do not despair, GIS data formats can be confusing:* For GIS, there is a wealth of different file formats. Many of them are legacy formats (such as shapefiles), which continue to be used in the field. Also, the mix of two very different approaches (vector and raster data) adds to the complexity. As you make progress in spatial analysis, you will encounter many more file formats, and it is useful to consult the various online references for more information about their specifications.

Text Data

So far in this book, we worked almost exclusively with data structured in tabular form. In the previous chapter, we saw how simple tables can be amended with spatial coordinates, such that each entry in the table is linked to a location on the globe. In this chapter, we cover a type of data with considerably less structure: texts. A text can be any written statement or report, but also spoken words that were transcribed. Text data is important to understand a wide variety of phenomena that are of interest to social scientists: What issues are discussed in parliamentary debates? How do certain hash tags travel on social media? How do journalists frame particular social issues in news reporting? All these applications require us to deal with text data.

In recent years, text analysis has become extremely popular in the social sciences. While traditionally the domain of (computational) linguistics, several types of text analysis have now become part of the social science toolkit. Linguists focus more on in-depth analysis of text where they try to identify, for example, the structure of sentences, and the subject or the object of an action. In the social sciences, and in political science in particular, text analysis has been done with simpler approaches, for example, analyzing word frequencies, or searching for the occurrence of particular keywords in texts.

Whatever type of analysis we plan to apply to textual data, we will have to obtain, manage and store these texts first. This is what we focus on in this chapter. The variety of methods for text analysis is so large that this chapter cannot even provide a sufficient overview. Rather, we discuss what text data look like, how they are typically stored as files, and how we can process and manage these data both in R and in a relational database.

Country	USA	Country	HUN
Year	1978	Year	1997
Debate	34	Debate	52
<p>We meet in this General Assembly on the threshold of a new decade. It will be a time of complex challenge, a period in which, more than ever, co-operative endeavours among nations are a matter not only of idealism but also of direct self-interest. [...]</p>		<p>I am very pleased to see the Foreign Minister of neighbouring Ukraine assume the prestigious post of President of the General Assembly at its fifty-second session. In fulfilling your challenging tasks you may rest assured of the support and cooperation of the delegation of Hungary. [...]</p>	

FIGURE 12.1. Two documents with associated metadata.

In doing so, and in line with the general approach of this book, the focus is on the handling of text data *before* it is used for some kind of text analysis; therefore, we deal with the representation and storage of textual data, and how we can search and query them. Once we know how to do this, the data can later be used for all the different methods and tools that exist for text analysis, such as topic modeling, sentiment analysis or more advanced natural language processing approaches (Grimmer et al., 2022).

12.1 WHAT IS TEXTUAL DATA?

Textual data (or “text as data,” as it is often called) usually comes in the form of *documents* as the basic units. In its simplest form, a text dataset is a collection of documents, where each document corresponds to what we would call a “case” or an “observation” in a standard social science dataset. A collection of text documents is often called a “corpus.” Not surprisingly, each document in a corpus is much longer than the short strings we have seen in standard tables (e.g., country names). In many text datasets, individual documents are tagged with additional information. For example, in a corpus of political speeches, each speech can be labeled with the name of the speaker or the date it was held. Figure 12.1 illustrates what this looks like for speeches held during the United Nations General Debates, which will be introduced in more detail below.

The figure shows two speeches, one held by the US in the 34th General Debate in 1978, one by Hungary in the 52nd debate in 1997. The main data is the text of the speeches, and each of them is tagged with metadata. This means that the entire corpus of textual data can actually be represented in a tabular structure, where each of the metadata fields and the text itself correspond to a column. Figure 12.2 shows what this looks like

<i>Country</i>	<i>Year</i>	<i>Debate</i>	<i>Text</i>
USA	1978	34	We meet in this General Assembly [...]
HUN	1997	52	I am very pleased to see [...]

FIGURE 12.2. The two documents stored in a table.

for the two speeches above. Therefore, as in the chapters before, we can transform yet another type of data to a tabular format, and use many of the data operations we are already familiar with.

While we can use a *structured* data format such as a table to store an entire corpus, text data is typically considered to be *unstructured*. Here, “unstructured” refers to the main part of the data, which is the text. Unstructured means that the text does not follow a particular pattern or model, such that it is difficult to locate particular pieces of information in it. For example, it is likely that each of the texts in our example above contains information about the country holding the speech. You can see this in the speech from Hungary, which pledges the support of the “Hungarian delegation.” Still, it is not straightforward to extract this information, as the country information is not explicitly flagged as such in the text, and is likely to be phrased differently in speeches of other countries. Compare this to the structured part of the dataset (the metadata). Here, we can simply look up the respective column to find out about the country holding the speech. Hence, not surprisingly, unstructured data require different and more complex methods to extract information compared to structured data such as tables.

How do we store text data digitally? As for spatial data, there exist numerous options and file formats. A first distinction we have to make is whether we store an entire corpus with different documents in a single file or as a collection of files. For the latter, we can simply use one text file for each document (see Chapter 4 for some basics about text files). When following this approach, each text file contains only text, not a CSV-encoded tabular structure. This means that there are some potential issues. For example, recall what we discussed about text encoding in Chapter 4. If text contains special characters that exist for many languages worldwide, we have to make sure that we choose an appropriate encoding. Also, we have to decide where to store document metadata if the text files themselves contain only plain text. As we will see below, this is often done by encoding metadata in file names or folder names.

A different approach for storing a document collection is to keep all documents in a single file. Again, there are many different ways for doing

this. One that you are already familiar with is the CSV format, where each document and its associated metadata corresponds to a single line. When using CSV for collections of long texts, we have to be particularly careful about how to deal with commas and line breaks in the texts. Since these characters have a particular function in CSV files (they separate fields and lines), we have to make sure that the texts containing them are properly encoded in the CSV file, for example, by enclosing them in double quotes (see Chapter 4).

12.2 APPLICATION: REFERENCES TO (IN)EQUALITY IN UN SPEECHES

In 2015, the United Nations adopted the *2030 Agenda for Sustainable Development*, a plan for improving economic, social, and environmental conditions worldwide. At the core of this agenda is a set of 17 *Sustainable Development Goals* (SDGs) that should be achieved by the year 2030. Goal No. 10 is reduction of inequality, both at a global scale between countries and also between different groups within countries. More details about SDG No. 10 are provided on the UN website at <https://www.un.org/sustainabledevelopment/inequality/>. Inequality and its reduction have not always been a high priority for the UN. For example, during the Cold War era, much of UN politics was about international security and the avoidance of violent conflict.

How can we trace the salience of different topics in the UN over time? How can we find out whether and when (in)equality became an issue of concern for deliberations at the UN? This is the kind of question that can be analyzed using statements from political actors. In our application for this chapter, we focus on speeches by UN member states at the UN General Debate, held once a year at the beginning of each session of the General Assembly at the UN Headquarters in New York. At the General Debate, each state is usually represented by its head of government. The first General Debate took place in 1946, the 2020 General Debate was held in September 2020 and commenced the 75th session of the General Assembly.

General debates last for several days and consist of a series of speeches by the representatives of the member states. In this chapter, we rely on the UN General Debate Speech Corpus (UNGDC, Baturo et al., 2017), a collection of the General Debate speeches between 1970 and 2018. As Baturo et al. (2017) note, the speeches are used by the different governments to comment on particular events and developments in the past year,

but also to emphasize pressing issues in world politics. Therefore, the collection provides us with an interesting opportunity to find out when and how inequality was mentioned in these speeches over time.

Rather than placing all the speeches into a single file, the UNGDC is distributed as a compressed archive, where each speech is stored in a single plain text file. Speeches are stored in separate directories, one for each year. The names of the text files contain information about the country holding the speech, the session (starting with 25, which corresponds to the 1970 General Debate) and the year. This is what the data structure looks like for the 25th debate in 1970:

```
Session 25 - 1970
├── ALB_25_1970.txt
├── ARG_25_1970.txt
├── AUS_25_1970.txt
└── ...
```

The dataset uses ISO three-letter codes to denote countries. In each file name, the different metadata fields are separated with an underscore. For the purpose of illustration, and to limit the computational complexity of the code examples in this chapter, we focus only on speeches by the US as one of the dominant countries in the UN. Our simple task in this chapter is to locate mentions of terms related to (in)equality in the US speeches over time. In line with the previous chapters, we do this first in R only, and later also in PostgreSQL.

12.3 WORKING WITH STRINGS IN (BASE) R

As you know, R data frames have columns with different types, one of which is character vectors for text. The character sequences (strings) we have used so far are short, such as country or party names. The texts below are much longer, but in principle can be treated exactly as the short strings we encountered so far. As mentioned above, a collection of texts can be stored in a tabular file format such as CSV, in which case you can import them to R as any other CSV file. If, however, the texts are stored as separate files, this becomes a bit more difficult. This is the case for the UNGDC that we use in this chapter, where each file contains only the text of a speech, and the metadata is encoded in the file name. Luckily, there is a useful package called `readtext` that makes the import of these file collections easy. `readtext` is a companion package to the powerful

quanteda text analysis framework, which we take a closer look at below. With `readtext` installed, we load the package:

```
library(readtext)
```

The data repository for this chapter contains the UN General Debate speeches for the US, with one speech per file. The `readtext()` function is designed to import collections of files. Rather than just specifying a single file to be read, you can use the wildcard character `*` to specify a *pattern* of directories and files that the function should use. In our case, this patterns consists of the folder in which the text files are located (`ch12`), and the file name pattern `*.txt`. The function will then process all files ending in `*.txt` in the given directory. In addition, the function needs information about where the document metadata are stored. In *quanteda* terminology, these metadata are “document variables” or “docvars.” In our case, the country, the session, and the year of the respective speech are part of the file name, which is why we set the `docvarsfrom` parameter accordingly. Finally, we need to specify what metadata fields are encoded in the file name. If you omit this parameter, `readtext()` will assign standard names for these variables. Our speech files are named such that the different document variables are separated with an underscore, which the function recognizes by default. A different separator can be set with the `dvsep` parameter.

```
docs <- readtext("ch12/*.txt",
  docvarsfrom = "filenames",
  docvarnames = c("country", "session", "year"))
```

The `readtext()` function can process many more types of text files, including PDF or MS Word. Also, it can handle different ways of storing metadata, for example, in CSV format. Let us take a closer look at what the function does. If the import is successful, the function returns an (amended) data frame, where each speech corresponds to one row (49 in total). We can use standard R syntax to output a single document, such as this one:

```
docs[1,]

readtext object consisting of 1 document and 3 docvars.
# Description: df [1 x 5]
  doc_id      text                country session year
* <chr>      <chr>                <chr>    <int> <int>
1 USA_25_1970.txt "\1.\t It is \"...\" USA      25 1970
```

This document is the speech given by the US in 1970, at the beginning of the UN's 25th session. Each document has a unique `doc_id`, generated from the file name. The metadata (country, session, and year) are contained in the respective columns, exactly as we specified above. The most important column is `text`, which contains the text of each speech. This is a standard character variable, and we can use R's basic functions to work with it. Before we turn to our research question and study how inequality is referenced in the speeches over time, let us examine the texts in more detail. Take a closer look at the first speech by outputting the beginning of the first two paragraphs with the `substr()` function that returns a subset of a string between the given positions:

```
substr(docs[1,]$text, 1, 22)
[1] "1.\t It is my privilege"
substr(docs[1,]$text, 957, 970)
[1] "2.\tDuring this"
```

It seems that each paragraph in this speech is numbered, followed by a tab character (`\t`). Recall that the tab is one of the invisible characters we discussed in Chapter 4. If you open the corresponding file `USA_25_1970.txt` in RStudio's text editor, you can verify that the numbering of paragraphs continues in the same fashion (digits, followed by a dot, followed by a tab character). These numbers may be problematic, since they are not part of the actual speech, but also are not used consistently throughout the dataset. For example, the speech from 1996 no longer has numbered paragraphs. Therefore, it is best to clean up the texts by removing the paragraph numbers. How can we do this?

Manually searching for (and replacing) individual numbers such as "1.", "2.", etc. is not an option, as there are dozens of numbered paragraphs in some speeches. Also, it would violate one of our core rules for data processing, which is that data manipulations should be transparent and replicable, and therefore be defined in code. For these reasons, we need a better search method, where we can flexibly define a search pattern. This is what so-called *regular expressions* (in short, *regex*) allow us to do. Regular expressions are extremely powerful and not just limited to R; in fact, they constitute a standard feature of many other programming languages as well. Relational databases can handle regular expressions too, as we will see below.

We start by first developing a pattern to locate the paragraph numbers, and later use this pattern to eliminate them from the speeches. Before we

use the real speeches, I demonstrate the use of regular expressions using some toy examples. In R, the most important functions to be used together with regular expressions are `grep()` and the closely related `grep1()`. They require two parameters: A regex pattern and a vector of strings in which to locate the pattern. `grep()` then returns the index for each string in which it was able to locate the pattern. For simplicity, we use `grep1()`, which returns a vector of the same size as the input vector, where each entry indicates whether the search pattern occurs in the respective input string. Let us try this with a simple example. A regex pattern can be a single character, for example, the character `a`:

```
grep1("a", c("data", "management", "book", "2022."))  
[1] TRUE TRUE FALSE FALSE
```

The character `a` occurs somewhere in the first two input strings, but not in the last two. If we refine our pattern such that it looks for the sequence `ag`, we get only one match, since this pattern only occurs in the string `management`:

```
grep1("ag", c("data", "management", "book", "2022."))  
[1] FALSE TRUE FALSE FALSE
```

Rather than particular characters, you can also search for *classes* of characters, such as all lowercase letters, or all digits. Let us try the latter. Before we do this, we need to briefly look at how R deals with strings and special characters within them, since this can interfere with how some regex patterns are defined. Some characters in R have a special meaning. For example, as you recall from Chapter 4, a line break is denoted by `\n`, which uses the special character `\`. Another example is single (`'`) or double quotes (`"`), which are used to denote the beginning and the end of a string, and therefore cannot occur within the string itself unless we remove their special meaning. To do this, you need to “escape” them with a backslash `\`. For example, to generate an actual backslash in a string, you write `\\`. To try this, you can use the `writelnLines()` function in R to output the *real* content of a string, not how it is represented in R. For example, `writelnLines("\\\\")` generates the output `\`.

Similar problems arise if we use a notation with a backslash (or other special characters) to define search patterns in regular expressions. In a regex pattern, the shortcut `\d` denotes a single digit (between 0 and 9). Since we need to escape the backslash, `\d` now becomes `\\d`. The added

backslash tells R to treat the next character as is, and not as one with a special meaning. Let us use this to locate digits in our toy example:

```
grep("\\d", c("data", "management", "book", "2022."))
[1] FALSE FALSE FALSE TRUE
```

Oftentimes, we want to detect *repetitions* of particular patterns, for example, a sequence of exactly four digits to search for years. This can be done with a regex *quantifier*, which indicates that a particular pattern must occur at least (or at most) a certain number of times. In its most generic form, this is denoted with curly brackets around the exact number of occurrences we want. The following example demonstrates this and we correctly locate the four-digit number in the last string:

```
grep("\\d{4}", c("data", "management", "book", "2022."))
[1] FALSE FALSE FALSE TRUE
```

There are different variations of the notation. For example, $\{n,m\}$ matches the preceding patterns at least n , but at most m , times. If you would like a pattern to be present at least once (but possibly more than that), you can also use the $+$ operator. Searching for at least one digit then simply becomes $d+$. Let us use this to search for a sequence of digits followed by a dot. The latter is again a special character and needs to be escaped to be interpreted as a full stop (dot).

```
grep("\\d+\\. ", c("data", "management", "book", "2022. "))
[1] FALSE FALSE FALSE TRUE
```

We are now already very close to a regex pattern that allows us to clean up the UN speeches. Recall that we need to search for a sequence of digits (at least one), followed by a dot, followed by a tab character. The latter is a literal character in a regular expression and does not need to be escaped, so we simply amend our above pattern with a $\backslash t$ before we can apply it to the speeches. To verify whether this works, we use the `grep()` function that returns the indexes of those texts where it was able to locate the pattern.

```
grep("\\d+\\. \\t", docs$text)
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

As we already suspected above, the numbering of paragraphs is not consistently applied in all speeches. Rather, the output of the `grep()` function shows that this pattern is only found in the first 15 speeches, which means that it stops after 1984. The `gsub()` function helps us remove these paragraph numbers. It takes as input a regex pattern (which we already have), a replacement text (which in our case is an empty string "", because we simply delete the paragraph numbers), and a vector of strings that should be modified. The latter is our collection of speeches, and we write the result back to our existing data frame:

```
docs$text <- gsub("\\d+\\.t", "", docs$text)
```

This example gave you an idea of how to work with regular expressions for searching and manipulating texts. Regular expressions are a widespread and extremely powerful technique to process strings, and we have only scratched the surface of the functionality and flexibility they offer. If you want to learn more about them, you will find many useful tutorials online or in more comprehensive R introductions. As mentioned above, regex are not a feature specific to R. While the general ideas apply also to other programming languages and regex implementations, there are different dialects with slight differences in the syntax. Even though we covered regular expressions for our work with political texts, their use is by no means limited to human text and language. In fact, regular expressions can be very useful to fix issues in data files, for example, in malformed CSV files.

12.4 NATURAL LANGUAGE PROCESSING WITH QUANTEDA

Base R and regular expressions can help you get a lot of tasks done when it comes to the management and processing of text data, as we have seen above. These tools are not specifically designed for the processing of natural language – you can use them for any types of strings. However, social science applications of text analysis mostly deal with text produced by humans, which is why we need extension libraries with features designed specifically for the processing of natural language. With the growing popularity of these approaches, a variety of software tools are now available for text analysis, including several ones for R.

Natural language processing (NLP) can be done at very different levels of sophistication. Simple approaches such as the one we use in this chapter are based on frequencies of words. More complex ones explore relationships between words, for example, by grouping them together

such that documents can be assigned to the topic(s) discussed in them. Even more sophisticated techniques for the processing of texts aim to get at the semantics of texts and their constituent parts. This usually involves the parsing of sentences to detect different classes of words such as nouns or verbs, or the extraction of subject and object in a sentence. All these more advanced methods are beyond the scope of this book, but they require the same type of input that we are dealing with in this chapter.

Almost all text analysis methods require some basic processing steps. In this section, we perform these steps with the `quanteda` library, one of the most advanced text analysis packages for R. In line with the scope of the book, however, we do not explore `quanteda`'s analysis features in depth. If you would like to learn more about the package, the online guide at <https://quanteda.io/articles/pkgdown/quickstart.html> is a good place to start.

Basic processing of text data involves a number of clean-up steps. One of the first is the splitting of texts into tokens, which usually correspond to words or word stems. This may seem straightforward: In English, and many other languages, words are separated by white spaces, so we can use these to separate words. However, in addition we need to deal with punctuation, which means that full stops, commas, or quotes also need to be taken into account. To do this, the computer needs instructions for what sequences of characters to assign to the same word, and when to start a new word. `quanteda` can deal with all this, so let us take a look. Once you have installed the package, you can load it with:

```
library(quanteda)
```

Since `quanteda` and `readtext` come from the same developers and are designed to work together, we can easily create a corpus from the documents imported above, from which we already removed the paragraph numbers:

```
speech_corpus <- corpus(docs)
```

Once the text is converted to a corpus, `quanteda` can easily split the texts into words and sentences. Take a look at the output of `summary(speech_corpus)` to find out how the length of the speeches (measured as the number of words or the number of sentences) varies in the speeches over time. For our application, however, we can get a first view by looking at the words we are interested in, and the context in which they occur with the `kwic()` function (“keywords in context”). We first use the `tokens()`

function to split the text into its constituent parts, and pass these tokens on to the `kwic()` function (the output is restricted to the first three occurrences for presentational purposes). With the `pattern` parameter, we define the term(s) we are interested in. Here, it is possible to specify a single term that will be matched exactly. In our example, we use the wildcard character `*`, which matches zero or more characters. This means that we can search for “equality” but also “inequality” with this simple pattern. Note that the pattern is matched against the *individual tokens* in the text which were generated when creating the corpus. The `window` parameter specifies the context that should be displayed with each match of the pattern – in our example, we show the two tokens left and right:

```
kwic(tokens(speech_corpus), pattern = "*equality", window = 2)[1:3,]
```

Keyword-in-context with 3 matches.

```
[USA_25_1970.txt, 1287] and human | equality | ; fifth
[USA_25_1970.txt, 3336] of racial | equality | . The
[USA_25_1970.txt, 3484] justice, | equality | and self
```

The example shows you what context our keywords occur in, and what the tokens in the text look like. Try removing the restriction to the first three lines to display the entire output, and you will see that we are capturing the right words that we are interested in. “Equality” and “inequality” occur together with references to justice or race. Oftentimes, these references are made with political goals (“combating inequality”). You can try to increase the window size in the above example to see more words before or after the target terms. The output also shows that the tokenization in *quanteda* does not remove anything from the text by default – punctuation characters such as `;` or `.` are included as individual tokens.

In the next steps, we create a data structure that is very common in text analysis: a document-feature matrix (DFM). This matrix has a column for each token (*feature*) in our corpus. Each row corresponds to a *document* in our corpus, and it contains the number of times that a token occurs in that text. Recall that we have a number of rather useless tokens in our corpus. This is why we first remove punctuation and numbers from our tokens before passing them on to the `dfm()` function. Later, we also remove so-called stopwords (words such as “a,” “the,” etc) from our DFM. For these stopwords, it is necessary to select the language (“en”), since stopwords are obviously specific to each language:

```
speech_dfm <- dfm(
  tokens(speech_corpus, remove_punct = T, remove_numbers = T))
speech_dfm <- dfm_remove(speech_dfm, pattern = stopwords("en"))
```

As you can find out with `featnames(speech_dfm)`, by default the DFM converts all tokens to lowercase, otherwise the different spellings of the same word (as in “Products” and “products”) would be counted as two different words. A DFM has usually lots of columns (10059, in our case), and many entries are zero because the corresponding terms do not occur in the respective document. If you print the DFM for our corpus, you can see this in the output (not shown here):

```
print(speech_dfm)
```

`quanteda` has a number of useful functions for exploring DFMs. One of them displays the most frequent words in the corpus, the “top features.” Not surprisingly, for our corpus of UN speeches, the top two words are “united” and “nations” (output again restricted to the first three for presentational purposes):

```
topfeatures(speech_dfm)[1:3]
```

united	nations	world
1604	1577	979

The DFM already contains the information we need for our applied example in this chapter. Recall that we want to count the mentions of (in)equality in the different speeches given by the US at the UN General Debate over the years. Our DFM gives us the number of times that *any* term in the corpus appears in the respective speech. Therefore, all we need to do is select the relevant terms from our DFM and extract the counts. We achieve this by creating a new, restricted DFM that only contains terms related to inequality. The `dfm_select()` function does this for us, and it accepts the same type of filter pattern as the `kwic()` function above.

```
dfm_ineq <- dfm_select(speech_dfm, pattern = "*equality")
```

By default, the function keeps the specified terms, which is exactly what we want. All that is left for us to do is to compute the row sums of the reduced DFM `dfm_ineq`, which gives us the number of times that either “inequality” or “equality” is mentioned in the speech. We add this count as an additional variable to our corpus:

```
speech_corpus$ineq_count <- rowSums(dfm_ineq)
```


At the end of the chapter, we create a simple plot using the year field and the newly created `ineq_count` field from the corpus.

12.5 USING POSTGRESQL TO MANAGE DOCUMENTS

In the final part of this chapter, we will show how to use a relational database to store and manage text documents. As we saw in previous chapters, the main focus of databases is the storage and efficient retrieval of data, not the analysis. This is why PostgreSQL is very limited when it comes to the processing of natural language; if your workflow involves text data stored in a database, you will typically export it for further processing in a specialized package such as `quanteda`. Nevertheless, it is useful to take a brief look at how PostgreSQL deals with text data, and how you can query the data with natural language searches. As in the previous chapters, we assume that you use a new database for this chapter, called `textdata`. We connect to our database with

```
library(RPostgres)
db <- dbConnect(Postgres(),
  dbname = "textdata",
  user = "postgres",
  password = "pgpasswd")
```

and import the UN speeches into a new table `speeches`. As you know from above, `readtext()` returns an extended data frame. We cannot directly send it to the database, which is why we convert it to a real data frame beforehand:

```
docs <- readtext("ch12/*.txt",
  docvarsfrom = "filenames",
  docvarnames = c("country", "session", "year"))
dbWriteTable(db, "speeches", as.data.frame(docs))
```

The table we created has five fields: the `doc_id` (the file name of the corresponding text file); the three `docvars` `country`, `session`, and `year`; and the `text` column that contains the text of the speech. As a next step, we again remove the line numbering from the speeches. Above, we have seen how to do this with regular expressions. Regex are not a feature specific to R; they also exist for PostgreSQL with a notation similar to the one described above. This is why we can use the search pattern `\d+\.\t` – which matches one or more digits followed by a dot, followed by a tab

character – also in our database. Let us first use the regexp search operator, the tilde `~`. This operator performs pattern matching. In the following query, we count the number of documents from our collection where the text field matches our search pattern (i.e., contains it at least once). Note that we again have to escape the backslashes properly, so that R passes them correctly to the database:

```
dbGetQuery(db, "SELECT count(*) FROM speeches WHERE text ~ '\\d+\\.\\.t'")
  count
1     15
```

The result is the same as above: 15 documents match our search pattern, which are the first 15 speeches in the dataset. How can we clean up the texts of these speeches? For this, we use the `regexp_replace()` function in PostgreSQL. It takes a string, a search pattern, and a replacement string, and returns a new string in which all occurrences of the search pattern have been replaced with the replacement string. The pattern we need is the same as above, and our replacement string is the empty string `'`, since we only want to delete the line number. In addition, the function takes optional control flags. Here, we must set the `g` (global) flag to extend the search/replace to *all* instances of the pattern, and not just the first one. There is no danger in applying this function to *all* texts in our sample; since the pattern is only found in the first 15 speeches, the others will remain unaffected:

```
dbExecute(db,
  "UPDATE speeches
  SET text=regexp_replace(text, '\\d+\\.\\.t', '', 'g')")
```

Having done some basic clean-up, we can now proceed to explore how to select documents from our database according to particular words in the text. One way to do this is the `~` operator, which allows us to specify a regex search pattern. Oftentimes, however, we do not really need regular expressions, which is why there is a simpler way to search text fields: the `LIKE` operator. Here, the syntax for the search pattern is much simpler. `LIKE` expects normal strings, which can contain two types of placeholders: the underscore (which matches any single character) and the percentage sign (which matches an arbitrary sequence of characters). How does this work in practice? First, we try to use `LIKE` with a search string that does not have any placeholders:

```
dbGetQuery(db, "SELECT count(*) FROM speeches WHERE text LIKE 'equality'")
  count
1     0
```

This query does not find any matching documents – why? The reason is that the search pattern `equality` is matched against the entire `text` column, so a match would only occur for any document where the *entire text* of the speech is “equality.” Of course, there is no single speech with this content in our collection. This is why we need the `%` placeholder, which matches an arbitrary sequence of characters. In this query

```
dbGetQuery(db,
  "SELECT count(*) FROM speeches
  WHERE text LIKE '%equality%')

count
1    11
```

we allow an arbitrary number of characters to occur before and after “equality.” This matches any speech where “equality” occurs at least once in the text, which is the case for eleven of them. Similar to `grep()` and its related functions in R, string matching operators such as `LIKE` are designed to work with strings in general. If you use them with natural language, they have no knowledge of what a word is, or that particular characters such as the dot can have a special meaning in language. This is why we need special extensions. PostgreSQL has a built-in set of functions for “full text search,” which help us process natural language. However, as the name suggests, it is designed primarily for searching natural language documents, so its applicability is much more limited compared to packages such as `quanteda`.

Recall that the first step in dealing with natural language is usually the clean-up of the text: We identify the tokens in the text and remove stop-words and punctuation. PostgreSQL does this by converting a document to a text search vector (`tsvector`), that contains a reduced form of the text. Similar to our document-feature matrix above, this vector contains the list of tokens in the text as well as the positions where they occur in the text. Therefore, once processed in this way, it is much easier to search for natural language terms in the text, since the DBMS only needs to go through this vector rather than the entire text. The creation of this vector is done with the `to_tsvector()` function, which converts a given string to a text search vector. Let us try this first with a simple example before applying it to the UN speeches:

```
dbGetQuery(db,
  "SELECT to_tsvector('english', 'The problems the world faces today')")

to_tsvector
1 'face':5 'problem':2 'today':6 'world':4
```

The output shows you what a vector looks like. It contains four tokens, along with the positions of these tokens in the text. For example, `world` is the fourth token in the text. There are a few things to note here. Most importantly, tokenization is language specific, so we need to specify `english` as the language. Now, take a look at the second token in the text, `problems`. This token is included as `problem` in the vector, with the “s” removed. The reason is that text search vectors trim the words to their word *stems*, so `problems` and `problem` are reduced to the same stem. Also, stopwords such as `the` have been removed from the index, and all the tokens are lower case.

The creation of text search vectors is computationally costly, so it is good practice to compute them once and save them in a separate column, such that you can use them later when searching the documents. The following code creates a new column of type `tsvector`, computes the vectors, and indexes the column using a special type of index (`gin`) to speed up data retrieval (see Chapter 10):

```
dbExecute(db, "ALTER TABLE speeches ADD COLUMN tokens tsvector")
dbExecute(db, "UPDATE speeches SET tokens = to_tsvector('english', text)")
dbExecute(db, "CREATE INDEX ON speeches USING gin(tokens)")
```

How do we use the text search vectors in practice? In PostgreSQL, we can now run a text search *query* against the vectors we created. A text search query uses a very simple syntax, similar to web search engines. In its simplest form, such a query is just a single word, but you can also connect different words with logical AND (&) and OR operators (!). We create a query with the `to_tsquery()` function. Importantly, PostgreSQL internally reduces the query in the same way as a text search vector. The advantage is that we do not have to worry about the different forms of a word – for example, `inequality` and `inequalities` are internally reduced to the same form:

```
dbGetQuery(db,
"SELECT
  to_tsquery('english', 'inequality'),
  to_tsquery('english', 'inequalities')")

to_tsquery to_tsquery..2
1 'inequ' 'inequ'
```

Now let us apply a simple query to our documents. For text searches, there is a special operator, `@@`, which determines whether a given text search vector matches a text search query. Here, we count the number of documents that match the query `inequality`:

```
dbGetQuery(db,
  "SELECT doc_id FROM speeches
  WHERE tokens @@ to_tsquery('english', 'inequality')")
  doc_id
1 USA_33_1978.txt
2 USA_53_1998.txt
3 USA_70_2015.txt
4 USA_71_2016.txt
```

If you go through the four speeches that match our query (the speeches for 1978, 1998, 2015, and 2016), you will see that not all of them contain the word *inequality* or some other form of it. The 1978 speech only talks about “international inequities and poverty” – however, since the text search applies stemming to the tokens, *inequality* and *inequities* are reduced to the same stem *inequ*, which is why we get a match also for the 1978 speech. This is exactly what we want in this case, since the 1978 speech talks about inequality between countries, which is what we are interested in. However, in other cases, the results of text searches can be too inclusive. Consider the following example, where we amend our pattern such that it searches of *inequality* or *equality*. Now, we get 34 matching documents:

```
dbGetQuery(db,
  "SELECT count(*) AS num_ineq FROM speeches
  WHERE tokens @@ to_tsquery('english', 'inequality | equality')")
  num_ineq
1      34
```

The reason is that *equality* is reduced to *equal*, which occurs frequently without any connection to *inequality*, for example, in sentences such as “Equally important, we hope that [..].” Hence, while text search can be a powerful and flexible tool to explore natural language, you have to be aware of the uncertainties when doing so. Before we proceed to show the result of our applied example, we close the database connection properly:

```
dbDisconnect(db)
```

12.6 RESULTS: REFERENCES TO (IN)EQUALITY IN UN SPEECHES

The reduction of *inequality* has been one of the UN’s *Sustainable Development Goals*, and the purpose of our exercise is to track the use of this

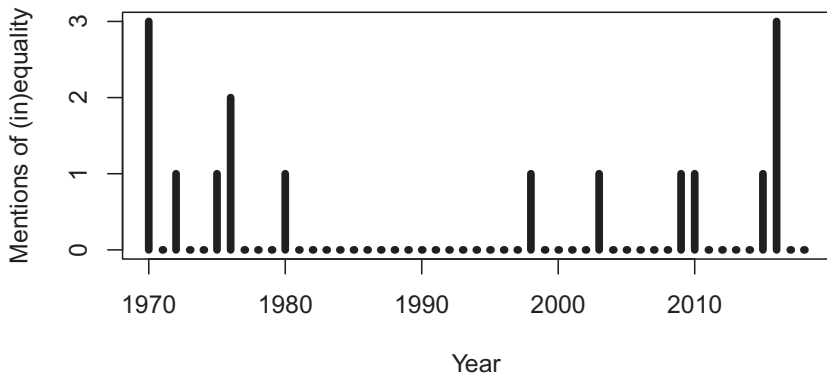


FIGURE 12.3. Number of mentions of (in)equality in UN General Debate speeches by the US over time.

and related terms over the years. We now use the statistics we computed with the help of the `quanteda` package. Above, we extracted the number of times that `inequality` or `equality` appear in the speeches from the DFM. In the simple plot in Figure 12.3, we use the two fields `speech_corpus$year` and `speech_corpus$ineq_count` for plotting.

We can see that in the early years of the sample, the US made several references to (in)equality in the UN General Debate speeches. This may be partly due to the civil rights movement in the US and political attention it had triggered to issues of inequality. The 1970s, however, were followed by a period without any mention in the 1990s, before the term came up more often again during the 2000s and later. The peak at the end of the study period (2016) coincides roughly with the adoption of the SDGs in 2015. Thus, as we can see based on this simple example, there is considerable variation in the salience of inequality in international politics over time, although it is not very prominent throughout.

12.7 SUMMARY AND OUTLOOK

Much research in the social sciences now relies on natural language data. In this chapter, we covered text as data, and how it can be processed in R and in PostgreSQL. Even though documents and their metadata can be stored in a tabular (structured) format, the texts themselves are examples of “unstructured” data. That is, *within* a text, we usually have no explicit structure, and the format and content of texts varies between documents. In this chapter, we used two different approaches to process text data. First, you can treat text simply as long strings, and use standard

string functions. Among these, we have introduced regular expressions, a very powerful method to search strings for particular patterns and replace them. Regular expressions are available in R and in PostgreSQL, but also in almost any other programming language. While useful when processing text data, they can be of great help for other tasks with strings, for example, when fixing malformed data files.

The second, more sophisticated approach we have illustrated in this chapter is to treat texts as natural language, and apply specialized methods for this. For R, the `quanteda` package is a good choice, but there are also other options such as `tidytext`, which integrates nicely into the `tidyverse` universe. These packages can perform a variety of NLP tasks, such as the splitting of a text into tokens, the elimination of stopwords and punctuation, or the reduction of words to their stems. For all of this, specialized knowledge of the particular features of a language are required, for example, how different sentences are separated, or how words can be trimmed to their stem. The relational database PostgreSQL has some basic functionality for doing this, which can come in handy if you keep a collection of documents on a centralized server and need to do fast and flexible lookups. For more advanced analysis of text, however, it is usually required to export the documents to R and use a text analysis package such as `quanteda`, whose functionality is much more advanced. For your future work with text data, here are some useful pointers:

- *Practice the use of regular expressions:* We began this chapter with an introduction to some standard string operations, which are available both in R and PostgreSQL. Among these, regular expressions are particularly powerful, but at the same time remain challenging even for experienced programmers. If you plan to work more with text data in the future, I recommend that you practice the use of regular expressions, since there are many operators and shortcuts we did not discuss in this chapter.
- *Compression is highly effective for text data:* Remember that we discussed the use of file compression in Chapter 4? This is particularly important if you store text datasets in files, since they can become very large. File compression works well with text files, and you can reduce the required disk space considerably for text data projects with tools such as `zip` or `gzip`. It is up to you to implement compression in a file-based workflow; PostgreSQL enables it automatically for text data.
- *Large files can be loaded directly into PostgreSQL:* So far, we have used R's DBI functions (such as `dbWriteTable()`) to import data into our

database. For very large CSV files, there is another way to this, which bypasses R completely. The PostgreSQL server can read files on your disk directly, which is much faster – for details, see the PostgreSQL documentation at <https://www.postgresql.org/docs/current/sql-copy.html>. There are some restrictions, however: This feature can only process tabular data in CSV (or similar) formats, and it becomes more difficult to use if you connect to PostgreSQL running on a remote server.

- *What if you need more flexible fuzzy string matching?* In the chapter, we discussed a number of ways in which you can specify search patterns to be located in strings. These approaches allow you to use different wildcard characters. Another way to implement non-precise, fuzzy string matching is by means of the “Levenshtein distance,” which is the number of characters that need to be changed when transforming one string into another. This is a common measure of similarity between strings, and you can use it in R with the `adist()` and the `agrep()/agrep1()` functions. In PostgreSQL, you can use the `levenshtein()` function, which is part of the `fuzzystrmatch` extension.

Network Data

In this final chapter on advanced data types, we discuss another kind of data that is frequently used in the social sciences: networks. Until now, we focused on data collections that cover separate entities: people, countries, elections, or conflict events. Now, we extend this perspective to examine *relationships* between them, in addition to the entities themselves. We represent these relationships as network structures.

13.1 WHAT IS NETWORK DATA?

A network is a structure consisting of entities (or nodes) and the relations between them. In more formal language, networks are often referred to as “graphs,” and the nodes as “vertices” with “edges” connecting them. For example, to represent an airline network, airports constitute the entities of the network, and the direct flight connections between these airports are the relations linking these entities. Graphs can also be used to represent social networks, where individuals are the nodes of the network, and edges exist between those individuals that know each other personally. Figure 13.1 (left panel) shows a simple network consisting of four nodes (A–D), and a total of four edges.

In the simple network above, any pair of nodes can either be connected with an edge or not. This is what we call an “undirected” graph, since the edges do not point one way or the other – they only connect a pair of nodes as in Figure 13.1 (left panel) above. Undirected graphs have many applications; for social network analysis, they can be used to connect individuals that have a symmetric relationship, for example, those that have coauthored a scientific publication (Newman, 2004) or have

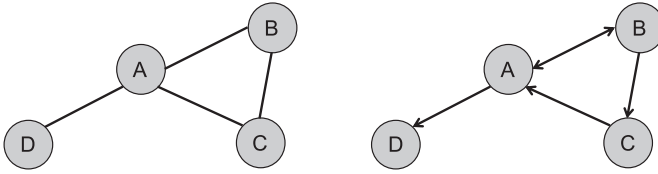


FIGURE 13.1. An undirected (left) and a directed graph (right).

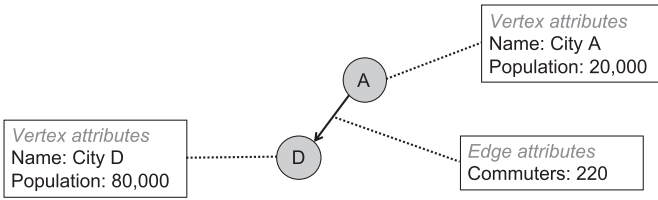


FIGURE 13.2. A graph with vertex and edge attributes.

co-starred together in a movie (Albert and Barabási, 2002). For many other applications, however, the links between the nodes in our network must be directional, and call this a “directed” graph. This simply means that we give each of our edges a direction. In Figure 13.1 (right panel), for example, you can see that there is an edge running from A to D, but not vice versa. Between A and B, however, we have edges in both directions (visualized as a two-directional arrow). Directed networks can be used to represent flows of some kind, for example, trade flows between countries (Barbieri and Keshk, 2017), or foreign direct investment of one country in another (Lee and Mitchell, 2012).

In many cases, simply having a network with vertices and edges is not enough, and we need to store more information about both. For this, we can amend the simple graph model, such that additional information is attached to the vertices and edges in our graph. Figure 13.2 illustrates this. We have a simple network with two nodes, A and D, each of which corresponds to a city. A directed edge from A to D represents the commuters from A who go to work in D every day. This network contains additional data, so-called “attributes,” about cities and commuting links (shown as boxes in the figure). We have information about the name and the population of each city in the vertex attributes, and the number of commuters in the edge attributes.

So far, a graph with edges and attributes was a conceptual data model – an idea of what our data looks like. How do we work with this data model in practice? In other words, how do we physically store network

data in files? There are two ways for doing this: as an adjacency matrix or an adjacency list. In both cases, we map our network to something that looks like a table. Once we have done that, we can use these tables exactly as we did in the previous chapters of this book: store them in files, or put them into a database for tabular data. But let us first look at how adjacency matrices and lists work.

The idea of an adjacency matrix is very simple: We create a quadratic table such that there is one row and one column for each vertex that exists in the graph. The entries in this table are then used to store information about the edges in the network, such that the rows correspond to the nodes where edges start, and the columns to those where they end. Let us illustrate this for the directed network from Figure 13.1 (right). Our adjacency matrix has four rows and four columns. To indicate that there is an edge running from A to D, we put a value of 1 in the first row, fourth column (and correspondingly for the other edges in our graph):

	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	1	0	0	0
D	0	0	0	0

While the adjacency matrix format is easy to understand, there are two major downsides to it. First, remember the advice I gave in Chapter 3: Tables should grow down, not sideways. The adjacency matrix violates this rule, since adding vertices to a graph means adding columns to the matrix. Second, adding attributes is very difficult when working with adjacency matrices. As soon as we want to store different edge attributes, this becomes impossible with a single matrix, since it holds exactly one value for each connection. Therefore, adjacency lists are preferable for most applications.

The adjacency list format follows a different approach. Rather than mapping out the entire set of possible pairs of edges and then indicating which ones are connected, an adjacency list simply contains only those pairs of edges that are connected in the graph. For the directed network in Figure 13.1, the adjacency list looks as follows:

From	To
A	B
A	D
B	A
B	C
C	A

The graph has five directed edges, each of which corresponds to a single line in our table. This format is very flexible: If we want to add more vertices or edges, we can do so by inserting more rows into the table. Also, if we want to add edge attributes to our graph, we can do so by storing them in separate columns, in addition to the `from` and `to` columns we have in our table. While edge attributes can easily be accommodated in this way, vertex attributes are typically stored in a separate vertex list, which is what we will do below.

We are now equipped with sufficient knowledge about the concept of networks and how we can store them in tables. Let us now take a quick look at the applied example we work on in this chapter.

13.2 APPLICATION: TRADE AND DEMOCRACY

International trade patterns constitute a central question in international political economy. When studying if and how much countries trade with each other, our main interest is not in single countries and their characteristics, but rather in the interactions between them. Therefore, the data that we need to study this is best represented as a network, where states constitute the nodes and the trade links between them are the edges. In our example, we study an important question: How does the level of democracy of states determine the volume of trade between them (Bliss and Russett, 1998)? The data for our analysis comes from two different sources. The first one is a dataset on bilateral trade, initially presented by Barbieri et al. (2009) and later updated until 2014 by Barbieri and Keshk (2017). The dataset draws on different sources and records (among other variables) the annual trade volume between pairs of states for the period 1870–2014. Therefore, rather than a single, static trade network, the dataset captures the temporal evolution of international trade, with annual observations.

The file `trade.csv` in the data repository contains a simplified version of the trade data. The format corresponds to the adjacency list we discussed above, so each trade link between two states constitutes an observation. States are coded according to the Correlates of War (COW) coding system (Correlates of War Project, 2008), which assigns independent states a

unique identifier, the COW code. For each link in our trade network, the states it connects are stored in the `ccode1` and `ccode2` variables. Due to the fact that we have annual observations, each line in the data also has a year variable. The last three variables provide information about the trade volume between the two states in the given year: `smoothflow1` is the total volume of the first country's imports from the second country (in millions of US dollars), and `smoothflow2` is the trade flow in the opposite direction. Both values are smoothed over time (see Barbieri and Keshk, 2017). Finally, `smoothtotrade` is the smoothed total volume of trade in the given year between the two states, independent of the direction.

Our second dataset for this chapter provides us with information about the states themselves, which we later use to explain the volume of trade between them. We rely on a subset of the large *Varieties of Democracy* (V-Dem) database, a project at the University of Gothenburg (Coppedge et al., 2019). Most importantly for our purpose, V-Dem provides aggregated expert assessments of many aspects of a country's political system, which we can use to examine how the level of democracy affects trade between two states. You can find a simplified version of the V-Dem data in the repository. It contains annual observations of states, each of which is coded with a `cowcode` and a year. Note that there are many missing values for the COW code, since V-Dem also tracks political units that are not considered to be independent states by COW and therefore have no COW identifier. The variable that we will be using to measure a state's level of democracy is `v2x_polyarchy`, which codes "electoral democracy" on a range from 0 to 1 (for more details about this and the other variables in the dataset, see the V-Dem codebook). In the file, we also have the world region the country belongs to (`e_regiongeo`) as well as the GDP per capita (`e_migppc`).

Together, the trade and V-Dem datasets constitute the (longitudinal) network we will be analyzing in this chapter. The trade dataset is an adjacency list with additional edge attributes, while the V-Dem data is a vertex list with vertex attributes. Vertices are identified by COW codes, so that we can link both datasets easily.

13.3 EXPLORING NETWORK DATA IN R WITH IGRAPH

When dealing with networks, we need a toolkit that is able to handle graph structures and allows us to conduct network analysis with them. R's base functions do not allow us to do that, as they are designed to mainly work with tabular data. However, several of R's extension libraries provide functionality to process network data. One of the most powerful

ones is the `igraph` package that we use in this chapter. `igraph` can read and export different data formats for network data, and allows us to manipulate and analyze networks in R. As always, we first need to load the package:

```
library(igraph)
```

There are different ways in which you can load a network into `igraph`. We use a function where we provide an edge list (the trade data) and a vertex list, both as simple R data frames. Before we can do this, we first need to import both datasets into R. Let us start with the trade network. Here, we need to keep in mind that the trade dataset stores missing values as `-9`, which is why we need to explicitly define this during the import. Also, we remove missing values and entries with a bilateral trade volume of 0, since they indicate that no trade is taking place and the states are therefore not connected:

```
trade <- read.csv(file.path("ch13", "trade.csv.gz"), na.strings = "-9")
trade <- subset(trade, !is.na(smoothtotrade) & smoothtotrade > 0)
```

If we take a look at the summary of the trade dataset, you will notice several things. Not surprisingly, even with the missing links removed, the dataset is large and contains around half a million observations. This is due to the fact that we observe pairs of states with annual estimates. To get started, we only use a subset of the trade data to simplify our exercise. We restrict coverage to 2014, and only keep those links with a total trade volume of 100 million dollars or more:

```
trade <- subset(trade, year == 2014 & smoothtotrade >= 100)
```

Next, we turn to the vertex list, the V-Dem data. We load it as a data frame, restrict it to 2014, and remove observations with missing COW codes since we do not need them in this exercise:

```
vdem <- read.csv(file.path("ch13", "vdem.csv"))
vdem <- subset(vdem, year == 2014 & !is.na(cowcode))
```

Before we can use the V-Dem data in `igraph`, we need to arrange the columns in the data frame. For the edge list, `igraph` expects the first two columns to contain the node identifiers – this is what we already have in the trade data frame. For the vertex data, `igraph` requires the first column to be the node identifier, which is why we need to make the COW code the first column in `vdem`:

```
vdem <- subset(vdem,
  select = c("cowcode", "country_name", "year",
            "v2x_polyarchy", "e_regiongeo"))
```

Now, both data frames should be in the right format so that we can create a network from them in `igraph`. Let us see how this works. We use the `graph_from_data_frame()` function, which is one among many different functions in `igraph` to construct a network. It takes an edge list as its main argument, and (optionally) a vertex data frame with additional data on the vertices. We also define the network to be undirected by setting `directed` to `FALSE`:

```
tradenetwork <- graph_from_data_frame(trade, directed = F, vertices = vdem)
```

This does not seem to work: `igraph` is complaining about some vertices in the trade data not being listed in `vdem`. The reason is that we do not have V-Dem codings for some states in the trade data – many of them are micro-states and are not covered by V-Dem. Therefore, we restrict our trade network to those pairs of states where V-Dem data is available for both of them:

```
trade <- subset(trade, ccode1 %in% vdem$cowcode & ccode2 %in% vdem$cowcode)
```

Now, let us try to construct the network again:

```
n <- graph_from_data_frame(trade, directed = F, vertices = vdem)
```

We now have an `igraph` network `n`, and can apply network-specific functions to it. First, we take a look at the summary:

```
summary(n)
IGRAPH 5ad0bc2 UN-- 174 3456 --
+ attr: name (v/c), country_name (v/c), year (v/n), v2x_polyarchy
| (v/n), e_regiongeo (v/n), year (e/n), smoothflow1 (e/n), smoothflow2
| (e/n), smoothtotrade (e/n)
```

Our network is undirected (UN) and has 174 vertices and 3,456 edges – if you want to check, you can compute these numbers with `vcount(n)` and `ecount(n)`. The summary also displays the attributes we have defined for our network. For example, `country_name` is a vertex attribute (v) of type character (c), while `smoothflow1` is an edge attribute (e) of type numeric (n).

While `igraph` defines its own methods for accessing and modifying a network, many of them work in ways that are similar to R. For example,

`V(n)` gives you access to the entire list of nodes in the graph, and you can retrieve any one of them simply by indexing. The following statement returns the first vertex in the graph:

```
V(n)[1]
+ 1/174 vertex, named, from 5ad0bc2:
[1] 700
```

Similar to a data frame, we can use the `$` operator to access a single attribute:

```
V(n)[1]$country_name
[1] "Afghanistan"
```

We can also use the bracket operator to filter a subset of nodes, for example, those with democracy scores higher than 0.9:

```
V(n)[v2x_polyarchy > 0.9]
+ 7/174 vertices, named, from 5ad0bc2:
[1] 225 94 390 220 385 380 2
```

For the edges, the `E(n)` function works in the same way. For example, it allows us to find out which edge has the maximum total amount of trade in 2014 with:

```
E(n)[which.max(E(n)$smoothtotrade)]
+ 1/3456 edge from 5ad0bc2 (vertex names):
[1] 710--2
```

Not surprisingly, this is the edge between China (COW code 700) and the US (COW code 2). So far, we have only used `igraph` to retrieve information that we could have also extracted from the original tables. The real added value of the library, however, is its ability to perform network-specific calculations. One of these is the “centrality” of nodes in the network, which is a key concept in network analysis. More central nodes are those that are better connected to others in the network. Centrality can be computed in different ways. “Degree centrality” is one of the simplest centrality measures, and it is defined as the number of links (the “degree”) a node has to others. In `igraph`, we can calculate degree centrality with the `degree()` function, as in:

```
degree(n)[1:3]
700 540 339
13 29 13
```

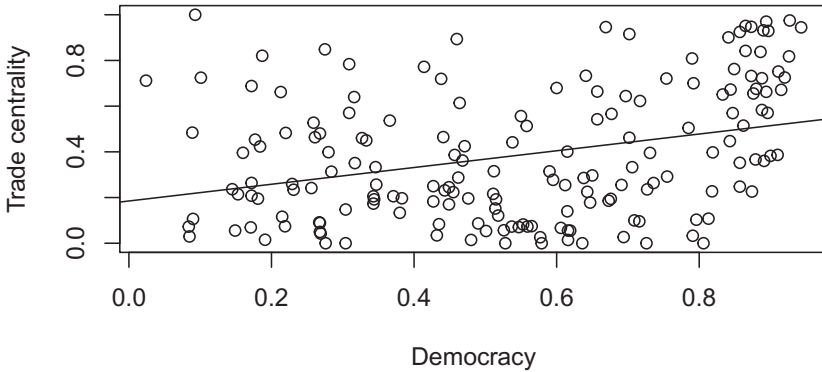



FIGURE 13.3. Level of democracy and centrality in the trade network.

Degree centrality, however, only considers the number of links that a country has, but not the other country it is connected to. Another centrality measure is eigenvector centrality, which gives higher centrality scores to those countries that are connected to other highly central countries. In other words, it measures centrality by identifying those countries that are connected to several other influential players in the trade network. Using the vector field, we can extract the centrality scores after running the corresponding function from *igraph*:

```
ec <- eigen centrality(n)$vector
```

We can use these centrality scores to carry out a first analysis of whether democracy is related to a country's position in the trade network. To do so, we create a bivariate plot of the democracy scores for the vertices in our network, and the centrality measures we have just computed (see Figure 13.3).

While the plot does not show a clear pattern, the linear fit is positive. Still, this relationship could be confounded, so we will conduct further analyses below. To conclude the discussion of *igraph*, let us examine the trade network graphically. The entire network is large and densely connected, which is why a plot of the entire network would not be useful. Therefore, we extract a subset of the network (a "subgraph") containing only those countries located in South America according to V-Dem's region coding (region 18):

```
sa <- induced_subgraph(n, V(n)[!is.na(e_regiongeo) & e_regiongeo == 18])
```

This results in a much smaller graph with only 12 nodes and 43 edges. Before we plot it, we define two properties of the network that are later

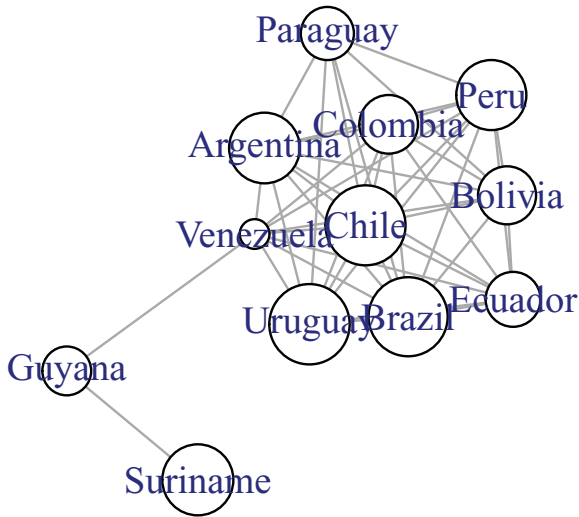


FIGURE 13.4. Trade network for South America.

used in the plot. First, we set a label for the nodes, which is simply the country name. Second, we define a weight for the edges, such that it is possible to distinguish trade relations with high volume from those with a low volume:

```
vertex_attr(sa, "label") <- V(sa)$country_name
edge_attr(sa, "weight") <- E(sa)$smoothtotrade
```

You can apply the generic `plot()` function to an `igraph` network and define a few network-specific parameters. In particular, we use a pre-defined layout function for the graph, which causes those nodes connected with edges of high weight (that is, with a high volume of trade between them) to be located close to each other. We also define the size of the nodes to be proportional to their democracy score, so that we can examine visually whether more democratic nodes are more central in the network:

```
plot(sa,
  layout = layout_with_gem,
  vertex.size = 40 * V(sa)$v2x_polyarchy,
  vertex.color = "white")
```

The plot in Figure 13.4 shows that some countries such as Chile are more central actors when it comes to trade in South America. At first

glance, there seems to be a weak relationship with the level of democracy, such that less democratic countries (Venezuela, Ecuador, Bolivia) are located more at the periphery of the network. Still, this relationship remains to be explored more systematically, which is what we do in the next section.

13.4 NETWORK DATA IN A RELATIONAL DATABASE

In the above example, we used a file-based workflow for processing and analyzing network data in R and the `igraph` package. For larger and more complex networks, it is often useful to store them in a database that can handle large amounts of data and make them available to different users and in different formats. The trade network that we study in this chapter is an example for a more complex network dataset, due to its longitudinal structure with annual observations. In our example above, we simply avoided this difficulty by using a snapshot of the network for the year 2014. Now that we are moving towards a database-backed setup, we want to be able to deal with the entire dataset, without taking shortcuts.

In the previous section, we have seen that network data can be stored as tabular data, and more precisely, as a combination of two tables: an adjacency (edge) list, and a vertex list. This makes it easy to transfer this setup to a relational database, similar to what we did in the previous chapters. Assuming that your PostgreSQL server is running and that you created the `networkdata` database for this chapter, we connect to the PostgreSQL server as before:

```
library(RPostgres)
db <- dbConnect(Postgres(),
  dbname = "networkdata",
  user = "postgres",
  password = "pgpasswd")
```

First, we import the trade network data, using the `dbWriteTable()` function in R. Note that we need to be careful with the coding of missing values (`-9`) in the data, to make sure they are correctly recognized as `NA` values during the import.

```
trade <- read.csv(file.path("ch13", "trade.csv.gz"), na.strings = "-9")
dbWriteTable(db, "trade", trade)
```

We repeat the same procedure for the V-Dem dataset, which contains additional attributes for the nodes in our dataset (the countries), again with annual values:

```
vdem <- read.csv(file.path("ch13", "vdem.csv"))
dbWriteTable(db, "vdem", vdem)
```

Is the trade network an undirected or a directed graph? In our exercises with `igraph` above, we treated it as an undirected network with symmetric links between countries, each link representing a trade relation with a given total volume. In reality, however, the dataset contains directed information: The `smoothflow1` and `smoothflow2` variables for two countries A and B tell us the amount of imports into A from B, and vice versa. The trade dataset lists each of these connections only once; for example, the edge between the US (COW code 2) and Canada (COW code 20) is present only once for each year, as an edge $2 \rightarrow 20$, while $20 \rightarrow 2$ is not listed as a separate entry in the data.

This data structure may be useful if we want to minimize the size of a data file, but it is not convenient when working with the data. For that reason, it is useful to turn the trade dataset into a proper directed network, where each edge has a start and an end point and only one attribute, trade volume. How can we do this? If we assume that `smoothflow1` is our main edge attribute, we could keep the existing entries, but would have to add all of them again, but with reversed direction and `smoothflow2` as the edge attribute. This is exactly what the following line does:

```
dbExecute(db,
  "INSERT INTO trade (ccode1, ccode2, year, smoothflow1)
  SELECT ccode2, ccode1, year, smoothflow2 FROM trade")
```

Let us go through the different parts of this statement. Overall, it is an `INSERT` statement, so it takes some data and adds it to the `trade` table. Importantly, it specifies four columns of the table that the new data should be inserted in: `ccode1`, `ccode2`, `year` and `smoothflow1`. These are the variables we would like to retain for our directed network with annual observations. But what is the data we want to insert? It is simply the entire trade table, but with reversed column order: `ccode2` should be inserted as `ccode1`, `ccode1` as `ccode2`, and `smoothflow2` as `smoothflow1`, while `year` remains the same. This is what we do in the second part of the `INSERT` statement, where we define the data to be inserted with a simple `SELECT` statement *on the same table the data should be inserted in*, but with the columns reordered such that they match the ones in the existing table.

Before we can proceed, however, let us clean up the table by removing the unnecessary columns `smoothflow2` and `smoothtotrade`:

```
dbExecute(db,
  "ALTER TABLE trade
  DROP COLUMN smoothflow2, DROP COLUMN smoothtotrade")
```

Also, the trade data contains entries with missing values in the trade volume column, or where the trade volume is zero. The latter is equivalent to there being *no* trade link between the two countries, so we remove these entries:

```
dbExecute(db,
  "DELETE FROM trade WHERE smoothflow1 IS NULL OR smoothflow1 = 0")
```

Finally, we strongly recommend that you create indexes on those columns that are frequently used to join tables, or to retrieve and aggregate data:

```
dbExecute(db, "CREATE INDEX ON vdem (cowcode)")
dbExecute(db, "CREATE INDEX ON vdem (year)")
dbExecute(db, "CREATE INDEX ON trade (ccode1)")
dbExecute(db, "CREATE INDEX ON trade (ccode2)")
dbExecute(db, "CREATE INDEX ON trade (year)")
```

Now, let us compute some simple network statistics directly in the database. Keep in mind that a relational database such as PostgreSQL has no notion of a network, so we cannot simply use existing functions to derive network statistics such as the length of shortest paths or centrality measures. However, we can use aggregate functions to compute the degree centrality of the different nodes. Recall that degree centrality is the number of incoming or outgoing links in a network. The following statement calculates this measure for the year 2014 and for all trade links with a volume of at least 100 million dollars, in decreasing order:

```
deg <- dbGetQuery(db,
  "SELECT ccode1, count(*) AS indegree
  FROM trade
  WHERE year = 2014 AND smoothflow1 >= 100
  GROUP BY ccode1
  ORDER BY indegree DESC")
deg[1:3,]
  ccode1 indegree
1      710      128
2         2      115
3      750      102
```

The statement selects trade links for 2014, and for each country (ccode1) counts the number of links that exist. Recall that the links in our trade network denote imports, which means that the three countries above are those that have the largest number of trade partners they import from. China, for example, imported from 128 other countries, with a volume of at least 100 million dollars from each. In a similar way, we can compute the centrality in terms of total import volume, simply by replacing the count() function with sum():

```
deg <- dbGetQuery(db,
  "SELECT ccode1, sum(smoothflow1) AS totalimports
  FROM trade
  WHERE year = 2014
  GROUP BY ccode1
  ORDER BY totalimports DESC")
deg[1:3,]
  ccode1 totalimports
1      2      2344005
2     710     2075854
3     255     1201135
```

Note that unlike working with *igraph*, we do not hard-wire the network such that it consists, for example, only of trade links with a volume of 100 million dollars. Rather, we can dynamically extract different parts of the network, depending on what we need. This is a convenient way to deal with more complex network data, where the complete dataset resides in a relational database, and snapshots are dynamically extracted to be analyzed in *igraph* or another network analysis software. Our database also allows us to deal with the time series nature of our data. For example, the code below computes the degree centrality of the US over time (again restricted to those links with at least 100 million dollars):

```
deg <- dbGetQuery(db,
  "SELECT year, count(*) AS indegree
  FROM trade
  WHERE ccode1 = 2 AND smoothflow1 >= 100
  GROUP BY year
  ORDER BY year DESC")
deg[1:7,]
  year indegree
1 2014      115
2 2013      116
3 2012      117
4 2011      117
5 2010      118
6 2009      116
7 2008      122
```

As a final step in our analysis, we test the link between democracy and trade more systematically. To this end, we extract our network data in a way that makes it possible for regression analysis to be applied. Here, we use a simple *dyadic* setup, where we treat each trade link as a single observation. In this analysis, we model the trade volume from one state to another as a function of the economic performance of the two states as well as their level of democracy. This is one simple way to analyze network data; one problem is that it ignores all of the network structure beyond the individual dyads. For example, in this dyadic analysis we treat the imports from State A to State B as independent of other trade flows (e.g., from State C to State B). More complex network models can accommodate these higher-order dependencies, although this makes the estimation much more difficult (Ward et al., 2013).

For the purpose of our simple dyadic model, we have to export the data as a single data frame such that R can fit a regression model. This data frame contains data about edges as well as vertices – essentially, it is a combination of the vertex and edge lists we used in this chapter. In the following statement, we merge the `trade` and `vdem` tables in our database into a single data frame by means of a `SELECT` statement. The join operation we use here is based on the entries in the `trade` table, and appends V-Dem variables both for the first and the second country in each dyad. Note that we are joining the V-Dem table *twice* to the trade links: for the first country `cocode1` in the dyad, and then again for the second country `cocode2`. This is why we have to use the two alias names for the V-Dem table: `vdem1` and `vdem2`. Also note that in order to make the estimation of the regression model faster, we restrict the data again to one year with `trade.year = 2014`. However, you can remove this part of the statement to obtain the data for the entire time period:

```
tradedyads <- dbGetQuery(db,
  "SELECT
    cocode1, cocode2, trade.year, smoothflow1,
    vdem1.v2x_polyarchy AS polyarchy1,
    vdem1.e_migdppc AS gdppc1,
    vdem2.v2x_polyarchy AS polyarchy2,
    vdem2.e_migdppc AS gdppc2
  FROM
    trade,
    vdem vdem1,
    vdem vdem2
  WHERE
    cocode1 = vdem1.cowcode AND
    trade.year = vdem1.year AND
    cocode2 = vdem2.cowcode AND
    trade.year = vdem2.year AND
    trade.year = 2014")
```

This design omits those pairs of countries that do not trade with each other. Does democracy affect not just the volume of trade between pairs of trading countries, but also whether a trade link exists between them? To test this, we need to construct our dataset such that it contains all possible dyads, in other words, all pairs of countries *regardless of whether they trade or not*. For all these possible dyads, we add data from V-Dem about economic performance and level of democracy, and then use these data in a regression model to explain whether they had a trade link or not.

Before you take a closer look at the following statement, let us first think about how we generate such a data structure. Our strategy consists of two steps: First, we create a list of all possible dyads, irrespective of whether trade occurs between them. Second, we add the data on trade links, such that we can identify those cases in our complete list of dyads that actually have a trade link. Process for the first step: Our `vdem` table (the node list) contains all the countries in our sample, observed once per year. To create a list with all possible dyads, we simply join the table with itself. As you can see in the next statement, the `vdem` table is used twice, once as `vdem1`, and once as `vdem2`. Since we have time series data with annual observations, we need to make sure, however, that we only join observations from the same year (`vdem1.year = vdem2.year`) and also exclude links from one country to itself (`vdem1.cowcode != vdem2.cowcode`), since we cannot have dyads linking a country to itself. This is what the complete statement looks like:

```
allldyads <- dbGetQuery(db,
  "SELECT
    vdem1.cowcode AS ccode1,
    vdem2.cowcode AS ccode2,
    vdem1.year,
    vdem1.v2x_polyarchy AS polyarchy1,
    vdem1.e_migdppc AS gdppc1,
    vdem2.v2x_polyarchy AS polyarchy2,
    vdem2.e_migdppc AS gdppc2
  FROM
    vdem vdem1,
    vdem vdem2
  WHERE
    vdem1.year = vdem2.year AND
    vdem1.cowcode != vdem2.cowcode AND
    vdem1.year = 2014")
```

For performance reasons, we again restrict this example to observations from the year 2014. As you can see, this gives us a large dataset with 30,102 observations, much more than those in our original trade dataset.

This is not surprising, since the latter contains only pairs of countries where some trade has been registered, whereas our dataset lists all possible pairs of countries.

We can now continue to the second step: Join the information in the trade table to the complete list of dyads. The following statement takes our code above to generate a *temporary, virtual* table as part of a SELECT statement. This is done using the WITH keyword in SQL. Our virtual table is called dyads and can be used in the main statement as if it were a real table:

```
alldyads <- dbGetQuery(db,
  "WITH dyads AS
  (SELECT
    vdem1.cowcode AS ccode1,
    vdem2.cowcode AS ccode2,
    vdem1.year,
    vdem1.v2x_polyarchy AS polyarchy1,
    vdem1.e_migdppc AS gdppc1,
    vdem2.v2x_polyarchy AS polyarchy2,
    vdem2.e_migdppc AS gdppc2
  FROM vdem vdem1, vdem vdem2
  WHERE
    vdem1.year = vdem2.year AND
    vdem1.cowcode != vdem2.cowcode)
  SELECT
    dyads.ccode1,
    dyads.ccode2,
    dyads.year,
    polyarchy1,
    gdppc1,
    polyarchy2,
    gdppc2,
    smoothflow1
  FROM dyads LEFT JOIN trade ON
    dyads.ccode1 = trade.ccode1 AND
    dyads.ccode2 = trade.ccode2 AND
    dyads.year = trade.year
  WHERE dyads.year = 2014")
dbDisconnect(db)
```

The most important part of the statement is the LEFT JOIN of dyads to trade – as you may recall, a left join preserves all data from the first table, and joins those entries from the second table where the join condition (on ccode1, ccode2, and year) is met. Fields from the second table (such as smoothflow1) will be filled with NULL values for those rows from the first table without a match in the second table. This example again restricts the data to the year 2014 – you can get the entire dataset by removing the LIMIT clause of the statement.

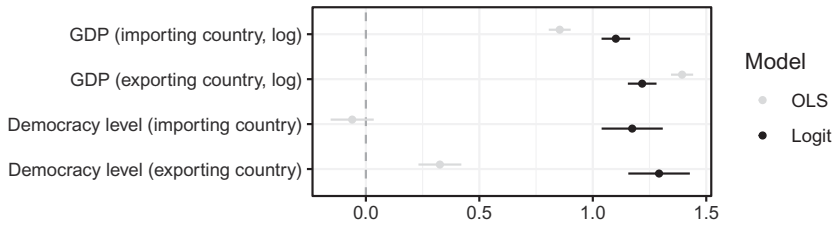


FIGURE 13.5. Coefficient plots for the regression models on trade and democracy.

The dataset we generate here can be used in an analysis where the dependent variable is the existence (0/1) of a trade link between a pair of countries. We can dynamically generate this binary dependent variable by testing whether the volume of imports between two countries is NA (which corresponds to the NULL values generated by the SQL left join, and indicates that the trade dataset does not contain a link between them).

13.5 RESULTS: TRADE AND DEMOCRACY

We have now extracted two datasets from our relational database. The first one (*tradedyads*) contains only those pairs of countries that trade with each other, and it allows us to study how the level of democracy affects the volume of trade between them. The *alldyads* dataset is a list of all possible dyads, and we will use it to analyze the impact of democracy on whether two countries trade at all. For the first analysis with volume of trade as the dependent variable, we simply fit a linear regression model, using the *log10-smoothflow1* variable as the dependent variable. For the second analysis, we use a logit model with a binary dependent variable, which takes the value 1 if *smoothflow1* is not NA, and 0 otherwise. Each model includes the democracy levels of the importing country (*polyarchy1*) and the exporting country (*polyarchy2*), as well as their GDP per capita values (log-transformed).

Rather than showing the regression tables, I present the coefficients from the models graphically in Figure 13.5. Not surprisingly, richer countries import and export more, as the positive coefficients for GDP variables show. Beyond that, democracy affects trade: The more democratic both countries are in a given dyad, the more likely it is that they trade with each other (see the coefficients for the logit model). The effect of democracy on the volume of trade is not as clear-cut, as the results from the OLS model show: While more democratic countries export more, there is no evidence that democracy affects the amount of imports into a country.

13.6 SUMMARY AND OUTLOOK

Much research in the social sciences is about relationships between different kinds of entities, and network data are designed to capture this. Networks consist of nodes and the links between them, and are usually stored in adjacency matrices or adjacency lists. We saw that the latter format is much more versatile, and corresponds to well-designed tabular data. More complex network data, where nodes and edges have additional attributes attached to them, can be stored with separate tables for the nodes and edges, which are linked by unique node identifiers.

In this chapter, we used the `igraph` package for R, which is designed to process and analyze network data, as well as create flexible visualizations. Due to its special focus, it is able to generate network-specific measures, such as different types of centrality for the network nodes. We also discussed how network data can be processed in a relational database. A strength of this approach is its ability to process large network datasets using the built-in performance improvements such as indexes. Our examples above demonstrate how you can generate different types of network datasets for your analysis, while keeping the data in a relational database. PostgreSQL does not have graph-specific functions for network data, but it can be extremely helpful in managing your network data and shaping it in different ways. For your work with network data, here is a set of recommendations:

- *Always prefer the adjacency list format:* As the above examples showed, it is much easier to work with network data that come in the form of adjacency *lists* rather than matrices. Adjacency lists conform to the “long” table format, which has a number of advantages. Most importantly, simple networks queries such as the number of neighbors per node become simple data aggregation operations, which can easily be done in R or PostgreSQL.
- *Use tabular data formats for network data:* Unlike for spatial data, there are few established data formats specifically for network data. In most cases, other, more generic ones are used to store information about graphs, for example, XML, JSON, or the CSV format. These formats are usually a good choice, even though they do not incorporate network-specific features. For example, as in our example above, with a tabular data format we need to distribute the dataset in different files.
- *Keep the different datasets (nodes and edges) consistent:* Since network data is often spread out across different files (nodes and edges), there can be potential inconsistencies between them. For example, a node

referenced in the edge list can be missing in the node list. Tools such as `igraph` can detect these issues, and in PostgreSQL you can use the referential integrity checks with primary and foreign keys for this.

- *For large networks, graph databases can be useful:* It is possible to leverage a relational database such as PostgreSQL for large network datasets, allowing us to use referential integrity checks and indexing. However, PostgreSQL is restricted to tabular data, and does not have any functionality to deal with graph operations – for example, it is difficult to find the neighbors of the neighbors of a given node. For this purpose, it is possible to use a specialized type of database designed for graphs, such as Neo4j.

PART V

CONCLUSION

Best Practices in Data Management

At the end of our journey through different tools and techniques for data management, what have we learned? How can we use these skills to make our research better? As I emphasized throughout the book, good data management means thinking about two different aspects: the *structure* of your data, and the *manipulation* of the data content.

14.1 TWO GENERAL RECOMMENDATIONS

Thinking about data structure is usually straightforward in the social sciences, since the vast majority of our data is stored in tables or can be fit into a tabular format. A tabular data format is one where we have information on separate cases (rows), each of which consists of the same set of variables (columns). Since this tabular data structure is so common in the social sciences, we hardly reflect on it. Still, there is considerable variation in how different tools deal with tables. In spreadsheets, for example, a table is simply a two-dimensional container of information, without necessarily imposing a standardized structure of variables and cases. Rows can have different lengths, cells can be merged, and the type of data stored in a column can vary between cases. This means that spreadsheet software offers little support to ensure the correct format of our data and to reduce errors, which (among many other problems) can lead to trouble when processing spreadsheet data further in statistical software. Hence, it is up to the user to check for errors and inconsistencies in the data.

Other tools take data structure much more seriously. R, for example, explicitly uses a rectangular tables as its standard data structures, such as data frames or tibbles. While the columns in a data frame always have types, these types can change silently when you re-assign values to them. This is called “dynamic typing,” and it is a standard feature of the R programming language. Therefore, while R at least maintains typed columns, it also does not prevent you from certain errors, such as setting a string value in what is supposed to be a numeric variable. Relational databases follow a stricter approach here. As you recall, in a relational database, data definition is a separate step in your workflow, where you set up the structure of tables before adding content. This involves specifying the columns and their types, and the database system later makes sure that only valid data is entered into the respective columns. Changing column types is an explicit step and needs to be done using the appropriate commands in SQL.

Data structure, however, also relates to the question of long vs. wide tables, or the splitting of data across several tables. Recall that “wide” tables are those where we have a dataset with two dimensions (e.g., countries and years), and where the columns are used to represent one of these dimensions. In a “long” table, the two dimensions are simply stored in different columns. We discussed that wide tables are usually not a good choice, as the addition of more data requires changes to the table structure (the addition of more columns). Also, as a general rule of thumb, tables should contain data on exactly one type of entity only. In order to avoid redundant data, we distributed our data across several tables, and linked them to each other by means of unique identifiers.

In sum, when working on an empirical research project, it is a good idea to reflect on the structure of your data. In some cases, this will be easy, in particular if your research project involves a single table only. For other projects, it may be worth spending some time to figure out a suitable structure underlying the data, as this will make your work easier. In the following, I list a number of questions that can help you do this.

RECOMMENDATION 1: THINK ABOUT THE STRUCTURE OF YOUR DATA.

- What variables does your dataset contain, and what are their types?
- Can you avoid redundant data? For example, can one variable simply be generated by transforming one or more other variable(s)? If so, there is no need to keep the former in the main dataset.

- Does your table grow down when adding data? Do you keep your data in a “long” format, such that the addition of new data and new variables remains easy?
- Do you only have data on one type of entity in the dataset? If not, you may consider dividing your data into several tables, making sure that unique identifiers exist to link the records.

The second main aspect to consider for an empirical research project is the workflow in which you process your data. As discussed in Chapter 1, data management involves the creation of datasets for analysis, starting from one or more raw input datasets. One of the most important requirements for us is the transparency and replicability of the data processing workflow. In other words, we need to make sure to document every step we take in getting from the raw input datasets to the analysis dataset. The way to do this is to save this workflow as a script, for example, using the R programming language. Manual “point-and-click” operations such as editing and reformatting data in a spreadsheet should be avoided if at all possible (but can of course be useful when creating a new, manually coded dataset from scratch). If you follow the approach taken throughout the book, all your data management work will be done in R and possibly SQL, which is why it is transparent and can be replicated.

Another important question is whether you need a file-based workflow, or whether you can benefit from using a dedicated database for your data. A purely file-based approach is technically easier to implement, since you do not need an additional database server and the separate steps to import/export the data. However, databases can be useful if your datasets are large and/or involve many interlinked tables. In these cases, specialized functionality in a relational DBMS can help you keep these tables consistent, while being able to speed up operations involving large numbers of records. Also, databases are designed for a multi-user environment with different levels of access. For example, this makes it possible to keep a database available such that some users can update it, while others have read-only access.

Of course, your workflow and the tools you choose for it are also strongly determined by the type of data you deal with. Simple tables with text and numbers can be processed in almost any type of statistical toolkit or database system. If you need to process more specialized types such as spatially referenced or textual data, this will affect your choice

of software. I presented various extension libraries for R, but we also discussed how PostgreSQL can be extended to manage data beyond the traditional tabular model.

Overall, the choice of a particular processing pipeline is closely related to the above questions around data structure. Once you know what type of data you deal with and what its structure should be, you can select the suitable software tools for processing it. Again, here is a list of possible questions you should consider when doing so.

**RECOMMENDATION 2: THINK ABOUT THE WORKFLOW
TO PROCESS YOUR DATA.**

- What is the amount of data you have? R can deal with small to medium-sized datasets well, but it may find large ones difficult to process. In these cases, databases provide the necessary features such as indexing, which allow you to deal with large amounts of data efficiently.
- Do you require the software to ensure the correctness and consistency of your data? If so, it may be useful to opt for a relational database with explicitly defined table structures and support for interlinked tables.
- Do you need to deal with specialized types of data? R has many extension libraries that can handle spatial data, text data, or networks. Some of this functionality is also available through PostgreSQL's extensions, but this is much more limited.
- Does the complexity of the technical setup matter? Purely file-based data storage with data processing in R is easier to set up, and requires less technical overhead for others replicating this work.
- Do several collaborators work on the data at the same time? If there is concurrent access to data by several users, file-based data storage is often not ideal. For these cases, a distributed setup with data stored on a separate database server should be preferred.

To conclude the book, I give some recommendations on how to handle two challenges that often arise in data management and coding: the collaborative work on research projects, and the public dissemination of research data.

14.2 COLLABORATIVE DATA MANAGEMENT

More and more work in the social sciences is conducted collaboratively, which means that several scholars work together to produce an article or a book. Oftentimes, this also means that empirical work on a project is

carried out by different researchers, and data and code must be shared between them. What is the best way to organize this process?

A simple and very common solution is to use a shared drive (such as Dropbox) for the exchange of data and code. In practice, however, this can lead to several problems. First, with multiple users accessing the same shared files, there is a huge risk of someone overwriting somebody else's changes. Imagine User A and User B editing an R script on the shared folder at the same time. If User A saves their changes first, User B will overwrite these changes when saving their edits. The same can happen to data stored on a shared drive. The second (and related) problem is that most cloud storage services do not keep histories of files, unless you explicitly enable (and pay for) this functionality. That is, once you save a file, only the latest version remains available, and you no longer have access to earlier ones. For these reasons, I recommend not to use simple shared drives for collaboration; they only work if there is a clear division of labor such that at any given point in time, there is *only one user writing to the shared drive*. This may be difficult to implement in practice and can still result in data loss, which is why it is preferable to use a version control system (VCS).

VCS have been developed for computer programming, so they can also be used for data management as long as all your operations are documented in code (which after reading this book, they should be). A VCS can be helpful for your work in two ways. First, you can save *different versions* of your source code as you continue to work on it. These versions represent different stages of your project, and you typically add a short summary to each version to describe what it does. Overall, this transparent approach to code development is also very useful when working on a project alone (without collaborators), since it allows you to go back to particular versions and track the changes you made since then – for example, to check for errors in your code. Second, for collaborative work, the VCS allows you to combine and merge changes made by different users into a single code base. Figure 14.1 illustrates this.

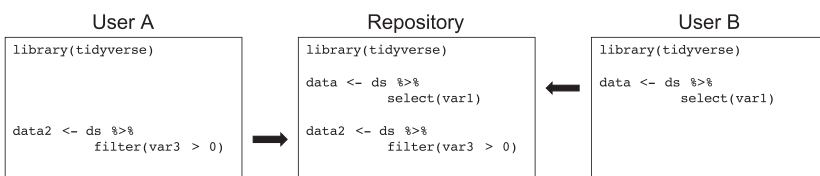


FIGURE 14.1. A version control system for collaborative coding.

If two researchers (User A and User B) each work on the same script, each of them can push their changes into a main repository maintained by the VCS. This repository serves as the storage facility for the different revisions that the users make to the code. It allows each user to pull changes made by the others and merge them into their local copy of the script, without overwriting their own edits. There exist different types of VCS. By now, *distributed* VCS such as Git are the most popular, and there exist numerous introductions and quick-start guides online (see, e.g., Blischak et al., 2016).

What type of content should be managed by a VCS? Systems of this kind are tailored to the management of source code. That is, they are designed to track changes in text files, but not in binary files (such as Stata's `.dta` files). For the purpose of data management, this usually means that only your data management *scripts* should be placed under version control, but not the data files themselves, if you can avoid it. For collaborative projects, I recommend that your code include download commands to fetch the input datasets from their original location, rather than storing these datasets in your repository. This way, each collaborator can generate the output datasets simply by running the project code, without the need to manually obtain copies of the input datasets. This, however, is only possible if the original input datasets do not change and can always be obtained from a given location. If this is not the case, adding a copy to your repository is usually the best option. Of course, if your data resides in a relational database and needs to be accessed by the different collaborators in the project, this is even easier, since you simply connect to the same database and perform much of your data processing there. The code to do this should of course be located in a file under version control.

A detailed introduction to version control is beyond the scope of this book. My goal in this section was to make you aware of these systems and provide some intuition on what they do. If you frequently work on projects with several collaborators, it is definitely worth learning more about these systems, even if the initial learning curve may be steep. However, as mentioned above, VCS are designed to manage code, not data. If you need to distribute data files, we take a brief look at data dissemination in the following section.

14.3 DISSEMINATING RESEARCH DATA AND CODE

The data management and processing that we described in this book is done by you as a researcher, and by the other members of your team.

However, at some point during the project, you usually need to make your procedures public, to ensure transparency of your scientific approach and to give others the possibility to replicate your work. At the latest, this should happen when a research article is published, and many journals by now require the publication of replication data and code along with the article. Typically, this includes material for the analysis only; in the terminology introduced in Chapter 1, this is the analysis dataset(s) and the analysis code.

However, at a time when empirical datasets in the social sciences are becoming increasingly complex and large, there are good reasons to also share data and code that were used to generate the analysis data in the first place. In particular, if several input datasets are involved, mistakes can happen when merging them. While sharing *code* is straightforward since the files are small and there is usually no sensitive content involved, this may be different for research *data*. Datasets can contain sensitive information and attributes that identify individuals, which is why they often cannot be shared in public. Other data are prevented from public dissemination by legal constraints, for example, if they were purchased from a commercial provider. While sharing data is preferred from a scientific point of view, it is essential that this happens within the given ethical and legal limits. To learn about new approaches for preserving privacy in research data while still making analysis possible, see, for example, Evans and King (2022). Also, all your data and code should be properly documented.

When all legal and ethical requirements for data sharing are met, you can use one of several ways to disseminate your research data and code. The easiest one is again to post these materials on a publicly available website, as many researchers and journals do for their replication materials. However, I recommend using one of the freely available, specialized portals for this purpose. Many institutions use the *Dataverse* software to create their own research dissemination infrastructure, but you can use Harvard's Dataverse at <https://dataverse.harvard.edu> to set up a free account and post your code and data. An even more comprehensive research infrastructure is provided by the *Open Science Foundation* at <https://osf.io>. While it contains numerous other useful tools to facilitate collaboration among researchers, it also allows you to share your data and code publicly.

These two portals have several features that make them particularly suitable for data dissemination. First, they maintain different versions of files. We already saw that this is useful for code files, but is also necessary when distributing research data publicly. Using the file history offered

by Dataverse or OSF, you can release new versions of your data that fix mistakes or extend coverage of your data. Since these versions can be tagged with a short description, you can document the evolution of a file or a dataset for users outside your project. Second, the data portals allow you to make your data files accessible under a fixed link, regardless of the current version of the file. That is, when you distribute a link to your file, this link remains the same even if you update the file. Older versions remain accessible in the data repository, but you need to explicitly request the respective version. This makes the development of datasets, but also the access to them, flexible and transparent, without the need to use your own website or a simple cloud storage facility.

14.4 SUMMARY AND OUTLOOK

At the beginning of this book, I asked whether it is useful to spend an entire book on data management. I gave a number of reasons for why this would be helpful to quantitative researchers working with increasingly complex and large amounts of data. A key learning I emphasized in the book is the need to *document* every data processing step in getting from the raw input data to the dataset(s) used for analysis. This makes data management as *convenient* as possible for you as a researcher, but also ensures that your research is *replicable and transparent* for others. All this is possible when you perform data management in code, such that each operation is included in your script. It becomes much more difficult when using manual operations, for example, in a spreadsheet software. The tools and techniques we covered in this book also allow you to make your data management process *scalable* and *versatile*, such that the same approaches can be used regardless of the size of your dataset or the types of data that you intend to use for your research project. Here, in particular, I introduced relational databases as a dedicated data management tool, which can handle large amounts of data, but can also be extended to cover more specialized types such as spatial or textual data.

The tabular data model has been of central importance throughout this book. Tables come naturally to social scientists – all the applied examples we teach in introductory methods classes involve data formatted as tables. In the book, I showed how tables are useful for many applications, including those that go beyond the conventional format of a single, rectangular table. Data can be distributed across several tables, but the traditional structure can also be extended to include more specialized columns such as those with spatial coordinates. Despite the continued

importance of structured, tabular data, unstructured data will become increasingly relevant also in the social sciences. In this book, we only scratched the surface of this topic when dealing with text data. Therefore, if you want to continue to expand your data analysis skills beyond the topics covered in this book, learning more about text analysis and related topics would be a promising way to go. I hope that this book can serve as a useful and comprehensive preparation as you continue this journey.

Bibliography

- Acemoglu, Daron, and Robinson, James A. 2005. *Economic Origins of Dictatorship and Democracy*. Cambridge University Press.
- Afrobarometer. 2021. *Afrobarometer Surveys*. Available at <https://afrobarometer.org>.
- Albert, Réka, and Barabási, Albert-László. 2002. Statistical Mechanics of Complex Networks. *Reviews of Modern Physics*, 74(1), 47.
- Barberá, Pablo. 2015. Birds of the Same Feather Tweet Together: Bayesian Ideal Point Estimation Using Twitter Data. *Political Analysis*, 23(1), 76–91.
- Barbieri, Katherine, and Keshk, Omar M.G. 2017. *Correlates of War Project Trade Data Set Codebook, Version 4.0*. Available at <http://correlatesofwar.org>.
- Barbieri, Katherine, Keshk, Omar M. G., and Pollins, Brian M. 2009. Trading Data: Evaluating Our Assumptions and Coding Rules. *Conflict Management and Peace Science*, 26(5), 471–491.
- Baturo, Alexander, Dasandi, Niheer, and Mikhaylov, Slava J. 2017. Understanding State Preferences with Text as Data: Introducing the UN General Debate Corpus. *Research & Politics*, 4(2).
- Blischak, John D., Davenport, Emily R., and Wilson, Greg. 2016. A Quick Introduction to Version Control with Git and GitHub. *PLoS Computational Biology*, 12(1), e1004668.
- Bliss, Harry, and Russett, Bruce. 1998. Democratic Trading Partners: The Liberal Connection, 1962–1989. *Journal of Politics*, 60(4), 1126–1147.
- Bryan, Jennifer. 2015. *How to Name Files*. Speaker Deck Online Video. Available at <https://speakerdeck.com/jennybc/how-to-name-files>.
- Chen, Xi, and Nordhaus, William D. 2011. Using Luminosity Data as a Proxy for Economic Statistics. *Proceedings of the National Academy of Sciences*, 108(21), 8589–8594.
- Coppedge, Michael, Gerring, John, Knutsen, Carl Henrik, Lindberg, Staffan I., Teorell, Jan, Altman, David, Bernhard, Michael, Fish, M. Steven, Glynn, Adam, Hicken, Allen, Lührmann, Anna, Marquardt, Kyle L., McMann,

- Kelly, Paxton, Pamela, Pemstein, Daniel, Seim, Brigitte, Sigman, Rachel, Skaaning, Svend-Erik, Staton, Jeffrey, Wilson, Steven, Cornell, Agnes, Gastaldi, Lisa, Gjerløw, Haakon, Ilchenko, Nina, Krusell, Joshua, Maxwell, Laura, Mechkova, Valeriya, Medzihorsky, Juraj, Pernes, Josefine, von Römer, Johannes, Stepanova, Natalia, Sundström, Aksel, Tzelgov, Eitan, Wang, Yi-ting, Wig, Tore, and Ziblatt, Daniel. 2019. *V-Dem Country-Year Dataset v9*. Varieties of Democracy (V-Dem) Project. Available at <https://www.v-dem.net>.
- Correlates of War Project. 2008. *State System Membership List, v2008.1*. Available online at <http://correlatesofwar.org>.
- DA-RT Initiative. 2015. *Data Access & Research Transparency*. Available at <https://www.dartstatement.org/>.
- Data Carpentry. 2017. *Data Organization in Spreadsheets for Social Scientists*. Online Course. Available at <https://datacarpentry.org/spreadsheets-socialsci/>.
- Döring, Holger, and Manow, Philip. 2018. *Parliaments and Governments Database (ParlGov): Information on Parties, Elections and Cabinets in Modern Democracies*. Development version. Available at <http://www.parlgov.org>.
- Dreher, Axel, Sturm, Jan-Egbert, and Vreeland, James Raymond. 2009. Development Aid and International Politics: Does Membership on the UN Security Council Influence World Bank Decisions? *Journal of Development Economics*, 88(1), 1–18.
- Evans, Georgina, and King, Gary. 2022. Statistically Valid Inferences from Differentially Private Data Releases, with Application to the Facebook URLs Dataset. *Political Analysis*, FirstView. DOI: 10.1017/pan.2022.1.
- Faundeen, John, Burley, Thomas E., Carlino, Jennifer A., Govoni, David L., Henkel, Heather S., Holl, Sally L., Hutchison, Vivian B., Martin, Elizabeth, Montgomery, Ellyn T., Ladino, Cassandra, Tessler, Steven, and Zolly, Lisa S. 2014. *The United States Geological Survey Science Data Lifecycle Model*. Technical report, available at <https://doi.org/10.3133/ofr20131265>.
- Gleditsch, Kristian Skrede. 2020. *Distance between Capital Cities*. Available at <http://ksgleditsch.com/data-5.html>.
- Grimmer, Justin, Roberts, Margaret E., and Stewart, Brandon M. 2022. *Text as Data: A New Framework for Machine Learning and the Social Sciences*. Princeton University Press.
- Grofman, Bernard, and Lijphart, Arend. 1986. *Electoral Laws and Their Political Consequences*. Vol. 1. Algora Publishing.
- Henderson, Margaret E. 2017. *Data Management: A Practical Guide for Librarians*. Practical guides for librarians, no. no. 28. Lanham, MD: Rowman & Littlefield.
- Högbladh, Stina. 2019. *UCDP GED Codebook Version 19.1*. Department of Peace and Conflict Research, Uppsala University.
- Houle, Christian. 2009. Inequality and Democracy: Why Inequality Harms Consolidation but Does Not Affect Democratization. *World Politics*, 61(4), 589–622.
- Izzo, Phil. 2012. *Is Data Is, or Is Data Ain't, a Plural?* The Wall Street Journal Real Time Economics Blog. Available at <https://blogs.wsj.com/economics/2012/07/05/is-data-is-or-is-data-aint-a-plural/>.

- Lee, Hoon, and Mitchell, Sara McLaughlin. 2012. Foreign Direct Investment and Territorial Disputes. *Journal of Conflict Resolution*, 56(4), 675–703.
- Lovelace, Robin, Nowosad, Jakub, and Muenchow, Jannes. 2019. *Geocomputation with R*. CRC Press. Available at <https://geocompr.robinlovelace.net>.
- Marshall, Monty G., Gurr, Ted Robert, and Jaggers, Keith. 2015. *Polity IV Project: Political Regime Characteristics and Transitions, 1800–2015*. Available at <http://www.systemicpeace.org/polity/polity4.htm>.
- Mudde, Cas. 2004. The Populist Zeitgeist. *Government and Opposition*, 39(4), 541–563.
- Newman, Mark E. J. 2004. Coauthorship Networks and Patterns of Scientific Collaboration. *Proceedings of the National Academy of Sciences*, 101(1), 5200–5205.
- Nordhaus, William D. 2006. Geography and Macroeconomics: New Data and New Findings. *Proceedings of the National Academy of Sciences*, 103(10), 3510–3517.
- Piketty, Thomas. 2014. *Capital in the 21st Century*. Cambridge, MA: Harvard University Press.
- Polo, Sara M. T. 2020. The Quality of Terrorist Violence: Explaining the Logic of Terrorist Target Choice. *Journal of Peace Research*, 57(2), 235–250.
- Richardson, Lewis Fry. 1960. *Statistics of Deadly Quarrels*. Pittsburgh, PA: Boxwood.
- Rooduijn, Matthijs, Van Kessel, Stijn, Froio, Caterina, Pirro, Andrea, De Lange, Sarah, Halikiopoulou, Daphne, Lewis, Paul, Mudde, Cas, and Taggart, Paul. 2019. *The PopuList: An Overview of Populist, Far Right, Far Left and Eurosceptic Parties in Europe*. Available at <http://www.popu-list.org>.
- Shafanovich, Yakov. 2005. *Common Format and MIME Type for CSV Files*. IETF Request for Comments. Available at <https://www.ietf.org/rfc/rfc4180.txt>.
- Strong, Robert A. 2021. *Jimmy Carter: The American Franchise*. Miller Center of Public Affairs, University of Virginia. Available at <https://millercenter.org/president/carter/the-american-franchise>.
- Sundberg, Ralph, and Melander, Erik. 2013. Introducing the UCDP Georeferenced Event Dataset. *Journal of Peace Research*, 50(4), 523–532.
- The Economist. 2016. *Excel Errors and Science Papers: Spreadsheets Are Playing Havoc with Scientists*. Graphic Detail Blog. Available at www.economist.com/graphic-detail/2016/09/07/excel-errors-and-science-papers.
- The White House. 2013. *Remarks by the President on Economic Mobility*. Press Release. Available at <https://obamawhitehouse.archives.gov/the-press-office/2013/12/04/remarks-president-economic-mobility>.
- The WID Team. 2020. *World Inequality Database*. Online Database. Available at <https://wid.world>.
- Tollefsen, Andreas Forø, Strand, Håvard, and Buhaug, Halvard. 2012. PRIO-GRID: A Unified Spatial Data Structure. *Journal of Peace Research*, 49(2), 363–374.
- United Nations Department of Economic and Social Affairs. 2019. *World Population Dynamics*. Available at <https://population.un.org/wpp/Download/Standard/Population/>.

- US Agency for International Development. 2021. *Demographic and Health Surveys*. Available at <https://dhsprogram.com>.
- US Federal Reserve Bank St. Louis. 2020. *FRED Data Portal*. Online Database. Available at <https://fred.stlouisfed.org>.
- US Library of Congress. 2019. *Sustainability of Digital Formats: Planning for Library of Congress Collections*. Online Resource. Available at <https://www.loc.gov/preservation/digital/formats/fdd/descriptions.shtml>.
2020. *Chronological List of Presidents, First Ladies, and Vice Presidents of the United States*. Online Resource. Available at https://www.loc.gov/rr/print/list/057_chron.html.
- Ward, Michael D., Ahlquist, John S., and Rozenas, Arturas. 2013. Gravity's Rainbow: A Dynamic Latent Space Model for the World Trade Network. *Network Science*, 1(1), 95–118.
- Warren, T. Camber. 2015. Explosive Connections? Mass Media, Social Media, and the Geography of Collective Violence in African States. *Journal of Peace Research*, 52(3), 297–311.
- Wasser, Leah. 2020. *Coordinate Reference System and Spatial Projection*. Earth Data Analytics Online Certificate, Lesson 3. Available at <https://www.earthdatascience.org/courses/earth-analytics/spatial-data-r/intro-to-coordinate-reference-systems/>.
- Weidmann, Nils B. 2011. Violence ‘from above’ or ‘from below’? The Role of Ethnicity in Bosnia’s Civil War. *Journal of Politics*, 73(4), 1178–1190.
- Wickham, Hadley. 2021. *The tidyverse Style Guide*. Available at <https://style.tidyverse.org>.
- Wickham, Hadley, and Golemund, Garrett. 2016. *R for Data Science*. Sebastopol, CA: O’Reilly.
- World Bank. 2021. *World Development Indicators*. Available at <https://databank.worldbank.org/source/world-development-indicators>.
- Worlds of Journalism Study. 2019. *Data and Key Tables: WJS2 (2012–2016)*. Online Resource. Available at <https://worldsofjournalism.org/data-d79/data-and-key-tables-2012-2016/>.

Index

- @@ SQL operator, 182
- [] operator, 26, 91, 194
- \$ operator, 27, 91
- %>% operator, 89

- Acemoglu, Daron, 88
- adist() function, 186
- adjacency list, 189, 205
- adjacency matrix, 189
- Afrobarometer, 4
- aggregation function, 32, 69, 83, 94, 95, 116
- agrep() function, 186
- agrpl() function, 186
- Ahlquist, John, 201
- Albert, Réka, 188
- ALTER TABLE SQL statement, 115, 130, 159, 162, 199
- Altman, David, 191
- American Standard Code for Information Interchange, 41
- Apple Numbers, 59
- ArcGIS, 149
- arrange() function, 98
- as.data.frame() function, 91, 157, 179
- as.Date() function, 77
- as.numeric() function, 79
- attribute table, 147
- attributes (graph), 188
- avg() SQL function, 141

- Barabási, Albert-László, 188
- Barberá, Pablo, 53
- Barbieri, Katherine, 188, 190

- Baturo, Alexander, 169
- Bernhard, Michael, 191
- Bliss, Harry, 190
- bounding box, 153
- Bryan, Jennifer, 55, 56

- Cartesian product, 81
- centrality, 194
- character variable, 27
- Chen, Xi, 60
- class() function, 53
- cloud storage, 213
- coding (measurement), 24
- colnames() function, 77
- coordinate system, 149, 153
 - geographic (unprojected), 149
 - projected, 149
- Coppedge, Michael, 191
- Cornell, Agnes, 191
- corpus (text data), 167
- Correlates of War, 92, 190
- countrycode package, 92
- countrycode() function, 92
- CREATE INDEX SQL statement, 139, 182, 199
- CREATE TABLE SQL statement, 111, 138
- CREATE USER SQL statement, 140

- DA-RT initiative, 8
- Dasandi, Niheer, 169
- data
 - redundancy, 33, 104, 121
 - representation, 25
 - sensitive, 215

- data (Cont.)
 - sharing, 215
 - structure, 210
 - structured, 168
 - type, 27, 79
 - unstructured, 168, 184
 - versioning, 216
- Data Carpentry, 72
- data definition, 36, 108
- data extraction, 36, 109
- data frame, 25
- data manipulation, 36, 109
- data storage
 - persistent, 39
 - volatile, 39
- data, scientific, 23
- `data.frame()` function, 25, 30, 89
- database client, 105
- database management system, 10, 20, 103
 - access control, 140
- database server, 105
- dataset, scientific, 24
- Dataverse, 215
- `dbAppendTable()` function, 114
- `dbConnect()` function, 20, 107
- `dbExecute()` function, 110
- `dbGetQuery()` function, 112
- DBI interface, 107
- `dbListTables()` function, 111
- `dbWriteTable()` function, 114, 159, 179
- De Lange, Sarah, 123
- declarative programming, 119
- degree centrality, 194, 199
- `degree()` function, 194
- DELETE SQL statement, 113, 131, 199
- Demographic and Health Surveys, 4, 9
- `dfm()` function, 177
- `dfm_remove()` function, 177
- `dfm_select()` function, 178
- doBy package, 32, 82
- document (text data), 167
- document variable, 171
- document-feature matrix, 177
- Döring, Holger, 110
- double variable, 27
- dplyr package, 88
- Dreher, Axel, 49
- DROP TABLE SQL statement, 114
- Dropbox, 213
- dynamic typing, 210
- E() function, 194
- edge (graph), 187
- `edge_attr()` function, 196
- `eigen_centrality()` function, 195
- eigenvector centrality, 195
- electoral disproportionality, 109, 118
- entity-relationship-model, 133
- EPSG list of spatial reference systems, 153
- escaping (characters), 173
- Evans, Georgina, 215
- event dataset, 150
- `expand.grid()` function, 138
- `extract()` SQL function, 115
- `featnames()` function, 178
- feature, spatial, 151
- field separator, 46
- file
 - binary, 40
 - compression, 48, 185
 - encoding, 41, 168
 - extension, 43
 - name, 55
 - text, 40
 - type, 39
- file format
 - CSV, 45, 70
 - Excel, 49, 64, 92
 - guide, 57
 - MS Word, 171
 - PDF, 171
 - R data, 52
 - serialized R data, 52
 - shapefile, 154
 - SPSS, 51
 - Stata, 50
- `file.path()` function, 18
- `filter()` function, 98
- Fish, Steven, 191
- foreign key, 122, 129
- `format()` function, 77, 79
- FRED data portal, 75
- Froio, Caterina, 123
- fuzzy string matching, 186
- G-Econ, 60
- Gallagher index, 109, 118
- Gastaldi, Lisa, 191
- `generate_series()` SQL function, 138
- Geo-referenced Event Dataset, 150
- Geographic Information Systems, 147

- geometry column, 159
 geometry, spatial, 148, 159
 Gerring, John, 191
 Git, 214
 Gjerløw, Haakon, 191
 Gleditsch, Kristian Skrede, 45
 Glynn, Adam, 191
 GRANT SQL statement, 141
 graph, 187
 directed, 188
 undirected, 187
 graph database, 206
 graph_from_data_frame() function, 193
 grep() function, 173
 grepl() function, 173
 Grimmer, Justin, 167
 Grofman, Bernard, 109
 group() function, 97
 group_by() function, 94, 157
 grouping (tibble), 94
 guess_encoding() function, 42
 gzfile() function, 48
- Halikiopoulou, Daphne, 123
 haven package, 50, 51
 Hicken, Allen, 191
 Houle, Christian, 88
 Höglbladh, Stina, 150
- identical() function, 54
 if_else() function, 93
 igraph package, 192
 Ilchenko, Nina, 191
 index (search), 136
 induced_subgraph() function, 195
 inner_join() function, 93
 INSERT SQL statement, 112, 130, 198
 install.packages() function, 18
 integer variable, 27
 invisible characters, 40, 172
 is.na() function, 195
- join, 93, 125
 inner, 93, 126
 left, 203
 spatial, 151, 156, 161
- Keshk, Omar, 188, 190
 King, Gary, 215
 Knutsen, Carl Henrik, 191
 Krusell, Joshua, 191
- kwic() function, 176
- labelled package, 51
 lag() function, 97
 lagged variable, 97
 layer, spatial, 156
 Lee, Hoon, 188
 left_join() function, 93, 158
 length() function, 83
 Levenshtein distance, 186
 levenshtein() SQL function, 186
 Lewis, Paul, 123
 LibreOffice, 59
 Lijphart, Arend, 109
 LIKE SQL operator, 180
 Lindberg, Staffan, 191
 load() function, 53
 logical variable, 27
 long table, 32, 97, 210
 Lovelace, Robin, 165
 ls() function, 53
 Lührmann, Anna, 191
- Manow, Philip, 110
 map projection, 149
 Marquardt, Kyle, 191
 Maxwell, Laura, 191
 McMann, Kelly, 191
 mean() function, 83
 Mechkova, Valeriya, 191
 Medzihorsky, Juraj, 191
 Melander, Erik, 150
 merge() function, 35, 79, 81, 89
 metadata (text data), 167
 Microsoft Excel, 49, 59
 cell formatting, 63
 filtering, 65
 freeze panes, 63
 Pivot table, 68
 sheets, 61
 Mikhaylov, Slava, 169
 Mitchell, Sara McLaughlin, 188
 Mudde, Cas, 122, 123
 Muenchow, Jannes, 165
 mutate() function, 92, 97
 MySQL, 106
- n() function, 94
 natural language processing, 175
 Neo4j, 206
 Newman, Mark, 187
 node (graph), 187

- Nordhaus, William, 60
 Nowosad, Jakub, 165
 nrow() function, 80
- Obama, Barack, 75
 one-to-many relationship, 125
 one-to-one relationship, 127
 Open Office, 59
 Open Science Foundation, 215
 openslx package, 50
 Oracle, 106
 overlay (spatial), 151, 156
- ParlGov, 109, 122
 Paxton, Pamela, 191
 Pemstein, Daniel, 191
 Pernes, Josefine, 191
 pgAdmin, 144
 Piketty, Thomas, 75
 pipe operator, 89
 Pirro, Andrea, 123
 pivot_longer() function, 98
 pivot_wider() function, 98
 plot.igraph() function, 196
 plot.sf() function, 153
 Polity IV, 88, 92
 Pollins, Brian, 190
 Polo, Sara, 50
 populism, 122
 PopuList, 123
 PostGIS, 150, 158, 159
 Postico, 144
 pre-registration, 6
 primary key, 122, 129, 134
 procedural programming, 119
- QGIS, 149, 165
 quanteda package, 171, 176
- R**
 code style, 21
 console, 15
 environment, 19
 help, 18
 packages, 18
 script, 74
 working directory, 17
 random() SQL function, 138
 raster data, 147
 rbind() function, 28
 read.csv() function, 43, 46, 48, 76, 78, 79, 192
 read_csv() function, 90
 read_delim() function, 90
 read_dta() function, 50
 read_excel() function, 49, 92
 read_sav() function, 51
 readr package, 42
 readRDS() function, 54
 readtext package, 170
 readtext() function, 171, 179
 readxl package, 49, 92
 referential integrity, 129
 regexpr_replace() SQL function, 180
 regular expressions, 172, 179
 relational integrity, 104, 206
 reliability, 24
 rename() function, 91
 renv package, 19
 replication, 8
 research workflow, 4, 5
 REVOKE SQL statement, 142
 Richardson, Lewis Fry, 3
 right_join() function, 93
 rm() function, 53
 Robert, Margaret, 167
 Robinson, James, 88
 Rooduijn, Matthijs, 123
 rowSums() function, 178
 Rozenas, Arturas, 201
 RPostgres package, 20, 107
 RStudio, 15, 16, 78
 RStudio project file, 17
 runif() function, 89, 138
 Russett, Bruce, 190
- sample() function, 138
 save() function, 53
 saveRDS() function, 54
 Seim, Brigitte, 191
 SELECT SQL statement, 112, 201, 202
 select() function, 90, 92
 sf package, 151
 Shafranovich, Yakov, 48
 Sigman, Rachel, 191
 Skaaning, Svend-Erik, 191
 slice() function, 90
 SPSS, 51
 SQL, 106
 aggregation, 116, 161
 data definition, 108
 data extraction, 109
 data manipulation, 109
 data types, 108, 111, 130

- SQL (Cont.)
 full text search, 181
 grouping, 116
 join, 125, 161, 201, 203
 NULL values, 113, 128, 203
 syntax, 108
 text search query, 182
 wildcard, 112
 SQL Server, 106
 sqrt() SQL function, 116
 st_as_sf() function, 153
 st_contains() SQL function, 161
 st_geometry() function, 153
 st_join() function, 156
 st_point() SQL function, 160
 st_read() function, 160, 162
 st_setSRID() SQL function, 160
 st_write() function, 160
 Stata, 50
 Staton, Jeffrey, 191
 stemming, 182
 Stepanova, Natalia, 191
 Stewart, Brandon, 167
 stopwords, 177, 182
 str() function, 78
 string quotation, 47
 string variable, 27
 Strong, Robert, 84
 structure (data), 25
 Sturm, Jan, 49
 subgraph, 195
 subset() function, 27, 29, 77, 78, 89, 192
 substr() function, 81, 172
 sum() SQL function, 117
 summarize() function, 94
 summaryBy() function, 32, 82, 83
 Sundberg, Ralph, 150
 Sundström, Aksel, 191
 Sys.time() function, 138
 Taggart, Paul, 123
 Teorell, Jan, 191
 tibble, 91
 tibble package, 91
 tidyr package, 88
 tidytext package, 185
 to_tsquery() SQL function, 182
 to_tsvector() SQL function, 181, 182
 token (text data), 176, 182
 tokens() function, 177
 tolower() function, 78
 topfeatures() function, 178
 data.frame\$typeof() function, 27
 Tzelgov, Eitan, 191
 UN General Debate, 169
 UN General Debate Speech Corpus, 169
 UN Sustainable Development Goals, 169, 183
 ungroup() function, 95, 97
 Unicode, 41, 70
 Unicode standard, 41
 UPDATE SQL statement, 160, 162
 USGS Data Lifecycle, 6
 V() function, 194
 validity, 24
 Van Kessel, Stijn, 123
 var_label() function, 51
 Varieties of Democracy, 5, 191
 vector data, 147
 version control system, 213
 vertex (graph), 187
 vertex_attr() function, 196
 von Römer, Johannes, 191
 Vreeland, James, 49
 Wang, Yi-ting, 191
 Ward, Michael, 201
 Wasser, Leah, 149
 which.max() function, 194
 Wickham, Hadley, 90
 wide table, 30, 97, 210
 Wig, Tore, 191
 Wilson, Steven, 191
 WITH SQL statement, 162, 203
 World Development Indicators, 4
 World Inequality Database, 75, 88
 Worlds of Journalism Study, 51
 write.csv() function, 47
 write_dta() function, 51
 write_sav() function, 52
 writeLines() function, 173
 Ziblatt, Daniel, 191

