Statistical Data Cleaning with Applications in R

Mark van der Loo
Statistics Netherlands
The Netherlands

Edwin de Jonge
Statistics Netherlands
The Netherlands

WILEY

Library of Congress Cataloging-in-Publication Data

Names: Loo, Mark van der, 1976- author.| Jonge, Edwin de, 1972- author.
Title: Statistical data cleaning with applications in R / by Mark van der Loo, Edwin de Jonge
Description: Hoboken, NJ : John Wiley & Sons, 2018. | Includes bibliographical references and index. |
Identifiers: LCCN 2017049091 (print) | LCCN 2017059014 (ebook) | ISBN 9781118897140 (pdf) | ISBN 9781118897133 (epub) | ISBN 9781118897157 (cloth)
Subjects: LCSH: Statistics–Data processing. | R (Computer program language)
Classification: LCC QA276.45.R3 (ebook) | LCC QA276.45.R3 J663 2018 (print) | DDC 519.50285/5133–dc23
LC record available at https://lccn.loc.gov/2017049091

Set in 10/12pt WarnockPro by SPi Global, Chennai, India

Contents

Foreword

Data cleaning is often the most time-consuming part of data analysis. Although it has been recognized as a separate topic for a long time in Official Statistics (where it is called 'data editing') and also has been studied in relation to databases, literature aimed at the larger statistical community is limited. This is why, when the publisher invited us to expand our tutorial "An introduction to data cleaning with R", which we developed for the *useR!*2013 conference, into a book, we grabbed the opportunity with both hands. On the one hand, we felt that some of the methods that have been developed in the Official Statistics community over the last five decades deserved a wider audience. Perhaps, this book can help with that. On the other hand, we hope that this book will help in bringing some (often pre-existing) techniques into the Official Statistics community, as we move from survey-based data sources to administrative and "big" data sources.

For us, it would also be a nice way to systematize our knowledge and the software we have written on this topic. Looking back, we ended up not only writing this book, but also redeveloping and generalizing much of the data cleaning R packages we had written before. One of the reasons for this is that we discovered nice ways to generalize and expand our software and methods, and another is that we wished to connect to the recently emerged "tidyverse" style of interfaces to the R functionality.

What You Will Find in this Book

This book contains a selection of topics that we found to be useful while developing data cleaning (data editing) systems. The range is very broad, ranging from topics related to computer science, numerical methods, technical standards, statistics and data modeling, and programming.

This book covers topics in "technical data cleaning", including conversion and interpretation of numerical, text, and date types. The technical standards related to these data types are also covered in some detail. On the data content side of things, topics include data validation (data checking), error localization, various methods for error correction, and missing value imputation.

Wherever possible, the theory discussed in this book is illustrated with an executable R code. We have also included exercises throughout the book which we hope will guide the reader in further understanding both the software and the methods.

The mix of topics reflects both the breadth of the subject and of course the interests and expertise of the authors. The list of missing topics is of course much larger than that what is treated, but perhaps the most important ones are cleaning of time series objects and outlier detection.

For Who Is this Book?

Readers of this book are expected to have basic knowledge of mathematics and statistics and also some programming experience. We assume concepts such as expectation values, variance, and basic calculus and linear algebra as previous knowledge. It is beneficial to have at least some knowledge of R, since this is the language used in this book, but for convenience and reference, a short chapter explaining the basics is included.

Acknowledgments

This book would have not been possible without the work of many others. We would like to thank our colleagues at Statistics Netherlands for fruitful discussions on data validation, imputation, and error localization. Some of the chapters in this book are based on papers and reports written with co-authors. We thank Jeroen Pannekoek, Sander Scholtus, and Jacco Daalmans for their pleasant and fruitful collaboration. We are greatly indebted by the R core team, package developers, and the very supportive R community for their relentless efforts.

Finally, we would like to thank our families for their love and support.

June 2017 *Mark and Edwin*

Website

Do not forget to visit the companion website for this book:

www.data-cleaning.org

There you will find valuable materials designed to enhance your learning, including:

- supplementary materials

1

Data Cleaning

1.1 The Statistical Value Chain

The purpose of data cleaning is to bring data up to a level of quality such that it can reliably be used for the production of statistical models or statements. The necessary level of quality needed to create some statistical output is determined by a simple cost-benefit question: when is statistical output fit for use, and how much effort will it cost to bring the data up to that level?

One useful way to get a hold on this question is to think of data analyses in terms of a value chain. A value chain, roughly, consists of a sequence of activities that increase the value of a product step by step. The idea of a *statistical value chain* has become a common term in the official statistics community over the past two decades or so, although a single common definition seems to be lacking.[1] Roughly then, a statistical value chain is constructed by defining a number of meaningful intermediate data products, for which a chosen set of quality attributes are well described (Renssen and Van Delden 2008). There are many ways to go about this, but for these authors, the picture shown in Figure 1.1 has proven to be fairly generic and useful to organize thoughts around a statistical production process.

A nice trait of the schema in Figure 1.1 is that it naturally introduces activities that are typically categorized as 'data cleaning' into the statistical production process. From the left, we start with raw data. This must be worked up to satisfy (enough) technical standards so it can serve as input for consistency checks, data correction, and imputation procedures. Once this has been achieved, the data may be considered valid (enough) for the production of statistical parameters. These must then still be formatted to be ready for deployment as output.

One should realize that although the schema nicely organizes data analysis activities, in practice, the process is hardly linear. It is more common to clean data, create some aggregates, notice that something is wrong, and go back. The purpose of the value chain is more to keep an overview of where activities take place (e.g., by putting them in separate scripts) than to prescribe a linear order of the actual workflow. In practice, a workflow cycles multiple times through the subsequent stages of the value chain, until the quality of its output is good enough. In the following sections we will discuss each stage in a little more detail.

[1] One of the earliest references seems to be by Willeboordse (2000).

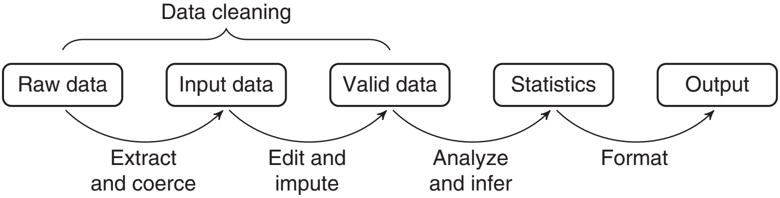


Figure 1.1 Part of a statistical value chain, showing five different levels of statistical value going from raw data to statistical product.

1.1.1 Raw Data

With *raw data*, we mean the data as it arrives at the desk of the analyst. The state of such data may of course vary enormously, depending on the data source. In any case, we take it as a given that the person responsible for analysis has little or no influence over how the data was gathered. The first step consists of making the data readily accessible and understandable. To be precise after the first processing steps, we demand that each value in the data be identified with the real-world object they represent (person, company, something else), for each value it is known what variable it represents (age, income, etc.), and the value is stored in the appropriate technical format (number and string).

Depending on the technical raw data format, the activities necessary to achieve the desired technical format typically include file conversion, string normalization (such as encoding conversion), and standardization and conversion of numerical values. Joining the data against a backbone population register of known statistical objects, possibly using procedures that can handle inexact matches of keys, is also considered a part of such a procedure. These procedures are treated in Chapters 3–5.

1.1.2 Input Data

Input data are data where each value is stored in the correct type and identified with the variable it represents and the statistical entity it pertains to. In many cases, such a dataset can be represented in tabular format, rows representing entities and columns representing variables. In the R community, this has come to be known as *tidy data* (Wickham, 2014b). Here, we leave the format open. Many data can be usefully represented in the form of a tree or graph (e.g., web pages and XML structures). As long as all the elements are readily identified and of the correct format, it can serve as Input data.

Once a dataset is at the level of input data, the treatment of missing values, implausible values, and implausible value combinations must take place. This process is commonly referred to as data editing and imputation. It differs from the previous steps in that it focuses on the consistency of data with respect to domain knowledge. Such domain knowledge can often be expressed as a set of rules, such as `age >= 0`, `mean(profit) > 0`, or `if ( age < 15 ) has_job = FALSE`. A substantial part of this book (Chapters 6–8) is devoted to defining, applying, and maintaining such rules so that data cleaning can be automated and hence be executed in a reproducible way. Moreover, in Chapter 7, we look into methodology that allows one to pick out a minimum number of fields in a record that may be altered or imputed such that all the rules can be satisfied. In Chapter 9, we will have a formal look on how data modification using knowledge rules can be safely automated, and Chapter 10 treats missing value imputation.

1.1.3 Valid Data

Data are valid once they are trusted to faithfully represent the variables and objects they represent. Making sure that data satisfies the domain knowledge expressed in the form of a set of validation rules is one reproducible way of doing so. Often this is complemented by some form of expert review, for example, based on various visualizations or reviewing of aggregate values by domain experts.

Once data is deemed valid, the statistics can be produced by known modeling and inference techniques. Depending on the preceding data cleaning process, these techniques may need those procedures into account, for example, when estimating variance after certain imputation procedures.

1.1.4 Statistics

Statistics are simply estimates of the output variables of interest. Often, these are simple aggregates (totals and means), but in principle they can consist of more complex parameters such as regression model coefficients or a trained machine learning model.

1.1.5 Output

Output is where the analysis stops. It is created by taking the statistics and preparing them for dissemination. This may involve technical formatting, for example, to make numbers available through a (web) API or layout formatting, for example, by preparing a report or visualization. In the case of technical formatting, a technical validation step may again be necessary, for example, by checking the output format against some (json or XML) schema. In general, the output of one analyst is raw data for another.

1.2 Notation and Conventions Used in this Book

The topics discussed in this book relate to a variety of subfields in mathematics, logic, statistics, computer science, and programming. This broad range of fields makes coming up with a consistent notation for different variable types and concepts a bit of a challenge, but we have attempted to use a consistent notation throughout.

General Mathematics and Logic

We follow the conventional notation and use $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ to denote the natural, integer, and real numbers. The symbols $\vee$, $\wedge$, $\neg$ stand for logical disjunction, conjunction, and negation, respectively. In the context of logic, $\oplus$ is used to denote 'exclusive or'. Sometimes, it is useful to distinguish between a definition and an equality. In these cases, a definition will be denoted using $\equiv$.

Linear Algebra

Vectors are denoted in lowercase bold, usually $\boldsymbol{x}, \boldsymbol{y}, \ldots$. Vectors are column vectors unless noted otherwise. The symbols $\mathbf{1}$ and $\mathbf{0}$ denote vectors of which the coefficients are all 1 or all 0, respectively. Matrices are denoted in uppercase bold, usually $\boldsymbol{A}, \boldsymbol{B}, \ldots$. The identity matrix is denoted by $\mathbb{I}$. Transposition is indicated with superscript T and matrix multiplication is implied by juxtaposition, for example, $\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}$. The standard Euclidean norm of a vector is denoted by $\|\boldsymbol{x}\|$, and if another L_k norm is used, this is indicated with a subscript. For example, $\|\boldsymbol{x}\|_1$ denotes the L_1 norm of $\boldsymbol{x}$. In the context of linear algebra, $\otimes$ and $\oplus$ denote the direct (tensor) product and the direct sum, respectively.

Probability and Statistics

Random variables are denoted in capitals, usually $X, Y, \ldots$. Probabilities and probability densities are both denoted by $P(X)$. Expected value, variance, and covariance are notated as $\mathrm{E}(X)$, $\mathrm{V}(X)$, and $\mathrm{cov}(X, Y)$, respectively. Estimates are accented with a hat, for example, $\hat{E}(X)$ denotes the estimated expected value for X.

Code and Variables

R code is presented in fixed width font sections. Output is prefixed with two hashes.

```
age <- sample(100,25,replace=TRUE)
mean(age)
## [1] 52.44
```

Sometimes, it is useful to distinguish between the variable in the code and the logical concept. In such cases, the code variable will be denoted `age`, and the concept will be denoted *age*.

2

A Brief Introduction to R

The following sections provide an overview of some of R's core features. Besides an installation of R, we recommend installing one of the available integrated development environments (IDEs) for R. A good IDE does not only offer a nice interface to R and its help system but also helps you to organize projects, code, and data.

To benefit the most of this tutorial, it is a good idea to try out the code examples for yourself, play around with them, and to explain the results.

2.1 R on the Command Line

After starting R, or an IDE that connects to R, you have access to an interactive console, or command-line interface. The first use of it is to replace a pocket calculator. You can type in a calculation, and R will return the answer (preceded by a `[1]`).

```
1 + 1
## [1] 2
```

To get started, experiment with the following statements. Make sure to play around a little. All common mathematical functions are implemented in R.

```
1 + 1
3^2
sin(pi/2)
(1 + 4) * 3
exp(1)
sqrt(16)
```

To reuse results or values, you can store them with the `<-` operator.

```
x <- 10
y <- 20
```

R has now remembered the values 10 and 20 and named them `x` and `y`. In fact, `x` and `y` are now officially *R objects*. R is very flexible, and there are several other ways to define an R object. We may replace `<-` with =, we may replace a statement `x <- 10` with `10 -> x`, or we can be extra verbose and use `assign("x",10)`. The = operator is the only one that is encountered with some frequency in practice. Since = is also used for named argument passing in function calls (see Section 2.6.1), we recommend using the `<-` for assignment.

The content of an R object can be printed simply by typing its name in the console.

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

R objects can be stored for further computation, the results of which may again be stored.

```
x + y
## [1] 30
z <- x * y
q <- x^2*z
q
## [1] 20000
```

Finally, we note that values and variables can be compared using standard comparison operators.

```
x <= y
## [1] TRUE
x == y
## [1] FALSE
x > y
## [1] FALSE
```

Observe that the operator testing for equality is written as the double equals symbol '=='. Make sure not to confuse this with the single equals symbol, which functions as assignment operator.

2.1.1 Getting Help and Learning R

R has a built-in help system where every possible function is described. If you know the name of the function, its help file can be requested with the `?` operator. For example, to show the help of the function `mean`, type the following:

```
?mean
```

If you are not sure of the function's name, the help files may be searched using the double question mark operator.

```
??average
```

IDEs for R have built-in search for the help files that may be more convenient.

There are a number of good online resources to get help from fellow users. Most notably, the Q&A site `stackoverflow.com` provides many R-related questions that have already been answered by users (and questions about many other topics as well). In fact, if you type an R-related question in a search engine, chances are that the first hit is a stackoverflow page. You may also want to subscribe to the R-help mailing list (see `https://www.r-project.org/mail.html`). Here, questions are often answered by the developers of the GNU R itself. Do observe the 'netiquette' and follow the posting guide before posting a question to the list. In particular, you should search the mailing list prior to posting a question to avoid double posts.

Besides resources where answers to questions can be found, there are many blogs discussing R and applications of R. A good way to become familiar with all the possibilities of R is to frequently visit r-bloggers.com, where many R-related blogs are collected and presented in a newspaper-like format. Browsing through the blogs allows you to stumble upon functions and ideas that you cannot get from just following a tutorial.

Learning R is not something you should do alone. Besides the online community from which you can benefit, many cities have R user groups that organize frequent meetings that you can join. If your organization is using R, it is a good idea to organize a local user group within the organization. All you need is a room, a projector, and a laptop to start organizing meetings. In our experience, user meetings are a very efficient (and fun!) way to share knowledge and experiences among colleagues, friends, or classmates. The point is that even in base R, there are thousands of functions and many ways to solve the same problem. Informal user meetings are a good way of bumping into solutions you otherwise might not have thought of.

2.2 Vectors

The most basic type of object in R is called a *vector*, a sequence of values of the same type. The object is so basic that you have already worked with them. When in the previous examples we computed x + y, R was in fact adding two numeric vectors of length 1 containing the numbers 10 and 20.

There are several ways to create a vector. One simple way is to use the function c() (for concatenate, or combine).

```
# a vector with numbers 1, 2, and 3
c(1,3,5)
# a vector with two text elements
c("hello world","hello universe")
```

Ordered number sequences can be generated with the colon operator (:) or with the seq function.

```
# a vector with numbers 1,2,...,10
1:10
# a sequence of numbers from 1 to 6 in 100 steps.
seq(1,6,length.out=100)
```

Sequences of random numbers from various distributions can be generated as well.

```
# 100 numbers drawn from the standard normal distribution
rnorm(100)
# 50 numbers drawn from the uniform distribution on [2,7]
runif(50,min=2,max=7)
```

You may try to combine values of a different type in a vector, but R will then convert the type when necessary.

```
c(1,"hello", 3.14)
## [1] "1"     "hello" "3.14"
```

When this vector is printed, there are quotes around the 'numbers' "1" and "3.14". That is because R decided to convert these numbers to text since one of the elements in the vector is text (you can always convert a number to text but not the other way around). By the way, in R such a conversion of type is usually referred to as *coercion*, which is just another word for the same thing.

This automatic conversion has consequences for everyday use. For example, the function `read.csv` reads `csv` files into R's working memory. It automatically detects the value types of the columns assuming that the first row contains the column names. Now if you feed it a `csv` file, where one of the columns contains all numeric data, except in one field, say somewhere at the bottom, that whole column will be interpreted as a categorical variable by default. Of course this behavior can be controlled, but it is typical of R to perform coercion rather than throwing an error.

There are a few basic vector types with which R can work, listed in the following table:

`logical`	Boolean values `TRUE` or `FALSE`
`integer`	Whole numbers, $\mathbb{Z}$
`numeric`	Real numbers, $\mathbb{R}$
`complex`	Complex numbers, $\mathbb{C}$
`character`	Text
`raw`	Binary data.

There are also types for storing categorical and ordered data.

`factor`	Categorical data, unordered
`ordered`	Ordinal data

These types are really integer vectors combined with a table that describes which category (level) is stored as what integer.

You can ask any object of what type it is, using the `class` function.

```
x <- 1:3
y <- c("foo", "bar")
class(x)
## [1] "integer"
class(y)
## [1] "character"
```

There are two more types of metadata stored with a vector. The first is its number of elements, which can be retrieved with the `length` function.

```
length(y)
## [1] 2
```

Secondly, the elements of a vector can be given names. For example:

```
shoesize <- c(jan=43, pier=39, joris=45, korneel=42)
```

The names are printed when a vector is printed to screen, but they do not affect any computations based on the vector.

```
mean(shoesize)
## [1] 42.25
```

The names of a vector can be retrieved with the `names` function.

```
names(shoesize)
## [1] "jan"     "pier"    "joris"   "korneel"
```

2.2.1 Computing with Vectors

All arithmetic and comparison operators and mathematical functions can be used on numerical vectors as you would on single numbers. The convention is that such operators and functions work element-wise on vectors.

```
x <- c(2,3,5,7)
y <- c(1,2,4,8)
x + y
## [1]  3  5  9 15
x < y
## [1] FALSE FALSE FALSE  TRUE
exp(-x) + sin(y)
## [1]  0.9768063  0.9590845 -0.7500645  0.9902701
```

The result of adding or comparing two vectors is again a vector, which may be stored and used in further computation.

It is possible to combine two vectors of different length. To compute the result, the shortest vector is repeated over the longer one.

```
3 + x
## [1]  5  6  8 10
z <- c(1,2)
z + y
## [1]  2  4  5 10
```

Here, R adds 3 to each element of x. In the second line, it adds 1 to the first element of y and 2 to the second element of y. It then notices that it got to the end of the vector z, so it starts back at the beginning adding 1 to the third element of y and 2 to the second element of y. The formal term for this is *recycling*; it is a behavior that is deeply embedded in R. A natural question is what happens when one tries to add two vectors where the shorted vector does not 'fit' a whole number of times on the longer vector. The reader is invited to test this by executing the following statement:

```
1:3 + 5:8
```

Besides vectorized operations where vectors are combined to new vectors of similar size, the content of vectors can be summarized in various ways.

```
x <- rnorm(100)
# compute the mean
mean(x)
## [1] 0.01797222
# compute the sample variance
var(x)
## [1] 1.355544
# standard deviation
sd(x)
## [1] 1.164278
# Tukey's five-number summary
fivenum(x)
## [1] -2.3084766 -0.7386481 -0.2128319  0.8240454  2.9226912
```

Especially useful is the function summary, which can be used to summarize just about any type of R object, including vectors.

```
summary(x)
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -2.30848 -0.73186 -0.21283  0.01797  0.81193  2.92269
```

It is also possible to visualize the data in vectors. Common plotting functionality includes the following:

```
x <- rnorm(100)
y <- x + rnorm(100)
# scatterplot
plot(x,y)
# boxplot
boxplot(x)
# histogram
hist(x)
```

2.2.2 Arrays and Matrices

An *array* is just a vector, endowed with a bit of metadata that states its dimensions. Arrays can be created with the array function.

```
A <- array(1:12, dim=c(2,3,2))
```

Here, we created a $2 \times 3 \times 2$ array containing the numbers 1–12. We advise you to execute the above statement and observe the order in which R has filled this multidimensional array. Many data structures can usefully be represented as (multidimensional) arrays, with high-dimensional tables being the obvious example. Since arrays are all but equal to vectors, we can perform computations with them just like we can with vectors.

```
2 * A
c(1,2) * A
```

A matrix is an array that has precisely two dimensions. The purpose of a matrix in R is to represent the objects familiar from linear algebra. Matrices can be created with the matrix function.

```
A <- matrix(1:6,ncol=2)
b <- matrix(c(-1,1),nrow=2)
```

Since matrices are vectors under the hood, the multiplication A*A is executed elementwise. This means that linear operations on matrices (addition and multiplication by a constant) work as expected out of the box, thanks to recycling. To perform matrix operations, some special operators and functions are available in R. Below is an overview of the most important operations.

`A %*% b`	Matrix multiplication $\boldsymbol{Ab}$
`t(A)`	Matrix transposition $\boldsymbol{A}^T$
`solve(t(A)%*%A,b)`	Solve the linear system $\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$
`solve(t(A)%*%A)`	Compute the inverse $(\boldsymbol{A}^T\boldsymbol{A})^{-1}$
`svd(A)`	Singular value decomposition of $\boldsymbol{A}$.

Exercises for Section 2.2

Exercise 2.2.1 *On the command line, do the following:*

a) *Compute* $1^2 + 2^2 + \cdots + 50^2$.
b) *The mean of the sequence* $6, 7, \ldots, 75$
c) *Can you generate the sequence* $(1, 2, 4, 8)$ *in a single statement (not using* `c()`*)? Hint: think of recycling.*

Exercise 2.2.2 *If you create a vector with numbers in them, R by default stores them as* `numeric`*, or real values. You can force R to store integers by adding* `L` *after a number.*

```
x <- c(1L, 2L, 7L)
```

Now, execute the following code:

```
y <- 2 * x
```

Inspect the `class` *of* `x` *and* `y` *and explain what happened.*

2.3 Data Frames

A *data frame* is R's way to represent a rectangular data structure, where every row represents an observation, and every column represents a variable. An R-data frame is basically a sequence of vectors that may carry values of a different type, but they must all be of the same length.

R has a number of built-in datasets that can be used for examples and exercises.

```
data(iris)
head(iris,3)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

Here, we use the function `head` to print the first three lines of the dataset. The `iris` dataset contains sepal and petal length and width for three kinds of species of iris (see `?iris` for references). Like vectors, data frames can be summarized or plotted.

```
summary(iris)
plot(iris)
```

The `summary` command summarizes each column, and the plot command produces a matrix plot with scatter plots of each variable against one another. Other useful metadata can be retrieved with the following functions:

```
# the number of rows
nrow(iris)
# the number of columns
ncol(iris)
# both nr of rows and columns
```

```
dim(iris)
# names of the columns
names(iris)
# a shorter summary of a data.frame
str(iris)
```

The function `str` (short for structure) gives a technical overview of the contents of a `data.frame`, whereas `summary` gives a statistical summary.

Columns can be retrieved, added, or removed using the dollar operator.

```
# compute the mean sepal width
mean(iris$Sepal.Width)
## [1] 3.057333
# add a 'ratio' column
iris$ratio <- iris$Sepal.Width/iris$Sepal.Length
head(iris, 2)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species     ratio
## 1          5.1         3.5          1.4         0.2  setosa 0.6862745
## 2          4.9         3.0          1.4         0.2  setosa 0.6122449
# remove the 'ratio' column again
iris$ratio <- NULL
```

2.3.1 The Formula-Data Interface

Many functions in R support the so-called *formula-data interface*. This interface is aimed at specifying a relation between variables, separately from where the data can be found. A *formula* is an R-expression of the form

```
[dependent variable] ~ [independent variables]
```

where independent variables, or sometimes functions thereof, are stated on the right side of the tilde (~). Try the following examples to get a feel for this concept:

```
# specify a linear model
m <- lm(Sepal.Length ~ Sepal.Width + Species, data=iris)
summary(m)

# create a boxplot for each species
boxplot(Sepal.Length ~ Species, data = iris)

# create a scatterplot
plot(Sepal.Length ~ Sepal.Width, data = iris)
```

2.3.2 Selecting Rows and Columns; Boolean Operators

Being able to select subsets of data is a fundamental skill to data processing. In R, there are many ways to do so, including methods provided by packages. Here, we limit ourselves to methods provided by base R.

Perhaps, the most convenient command to select a subset of records using base R is the `subset` function. For example, to select a subset of records from the built-in `women` dataset whose height exceeds 70, we can do the following:

```
subset(women, height > 70)
##    height weight
## 14     71    159
## 15     72    164
```

The function accepts a data.frame and a logical statement that expresses the condition(s) for records in the subset. Note that we do not need to reference the women dataset in the logical statement. The subset function understands that height must be a variable stored in women. To build up logical statements, R supports the following basic Boolean operations and quantifiers:

&	Logical AND
|	Logical OR
!	Logical NOT
xor()	Logical EXCLUSIVE OR.
all()	Only true if every value in the input is TRUE.
any()	Only true if at least one value in the input is TRUE.

2.3.3 Selection with Indices

The second way to make selections from a dataset is to use vectors of indices. An index vector may be a logical vector of the same size as the object selected from or an integer or numeric vector stating the desired positions. To make a selection from an R object (vector, data frame), one uses the square bracket operators.

```
x <- c(1,7,10,13,19)
# select the 2nd element
x[2]
## [1] 7
# select the 2nd and 5th element
x[c(2,5)]
## [1]  7 19
# set the first element to 0
x[1] <- 0
# select all elements equal to zero
x[x==0]
## [1] 0
# set all elements equal to zero equal to 1
x[x==0] <- 1
```

To understand the last two statements, recall that a logical expression such as x==0 returns a logical vector, which may then be used as an index. Mastering indices is one of the most important R skills to acquire since it allows you to very flexibly and quickly find and alter data.

It is possible to separate the computation of an index from its application to data by storing a computed index prior to usage.

```
# find x < 15; I is a logical vector
I <- x < 15
# find x > 1; J is a logical vector
J <- x > 2
# select elements in x satisfying both I and J
x[I&J]
## [1]  7 10 13
```

There are a number of functions that can help you find specific values in a vector.

`min`, `max`	The smallest or largest value
`which.min`, `which.max`	The position of the smallest or largest value
`which`	Position of `TRUE`s in a `logical` vector

Here are some examples.

```
max(x)
## [1] 19
which.max(x)
## [1] 5
which(x == 7)
## [1] 2
```

Since data frames have two dimensions, you need two indices to select from them, one for the rows and one for the columns. For example, to select rows 3–7 and columns 2–4 from the `iris` dataset, do the following:

```
iris[3:7,2:4]
##   Sepal.Width Petal.Length Petal.Width
## 3         3.2          1.3         0.2
## 4         3.1          1.5         0.2
## 5         3.6          1.4         0.2
## 6         3.9          1.7         0.4
## 7         3.4          1.4         0.3
```

Indices before the comma select rows, and indices after the comma select columns. Leaving out an index means 'make no selection', that is, everything is returned. Here, we select all columns for the first row.

```
iris[1, ]
```

Similarly, we can select all rows for columns 2–4.

```
iris[ ,2:4]
```

There is one caveat when selecting columns in the above procedure. If only a single column is selected, for example, `iris[, 1]`, R will return a vector, rather than a single-column data frame. There are two ways to prevent this behavior. The first is by providing the extra argument `drop=FALSE` (meaning, dimensions will not be dropped).

```
iris[,1,drop=FALSE]
```

The second way is to not provide a comma and use only a single index when selecting columns.

```
iris[1]
```

Finally, we note that it is also possible to select columns with (vectors of) column names.

```
iris[ iris$Sepal.Length < 6, 'Species', drop=FALSE]
```

2.3.4 Data Frame Manipulation: The `dplyr` Package

The `dplyr` package of Wickham and Francois (2014) offers a set of functions that facilitate a very consistent way of working with data frames. The package will be discussed briefly in Section 4.2.3, but it also comes with an excellent tutorial, which can be found at the package's CRAN page.

Exercises for Section 2.3

Exercise 2.3.3 *The built-in* `women` *dataset contains two columns of data representing the average height (inches) and weight (pounds) of American women aged 30–39 (for some year prior to 1975).*

a) Add a column called `heightM` *representing* `height` *in meters. One inch equals 2.54 cm.*
b) Add a column called `weightKg` *representing* `weight` *in kilograms. One kilogram equals 2.2046 lb.*
c) The Quetelet index QI, (also body-mass index) is computed as

$$QI = \frac{\textit{weight in kilogram}}{(\textit{height in meter})^2}.$$

Add a column called `QI` *with the Quetelet index for each row in the* `women` *dataset.*
d) Compute the mean and median Quetelet index for this dataset.
e) How many indices are above 23? (Hint: you can count the number of `TRUE`*s in a vector by using computing the* `sum` *over them).*

Exercise 2.3.4 *Use (computed) indices to answer the following questions:*

a) How many rows in the `iris` *dataset have* `Petal.Length` *larger than 5 and* `Sepal.Width` *smaller than 3. Hint you can* `sum` *over* `logical` *vectors to count the number of* `TRUE`*s.*
b) There are two specimens of `iris` *that have* `Sepal.Length` *equal to 5.7 and* `Sepal.Width` *equal to 2.8. Of what species are they? Use a single R statement to select the species from the dataset.*
c) Of what species is the iris with the largest `Sepal.Length`*?*

Exercise 2.3.5 *The* winsorized mean *is a robust way to estimate the mean that works by replacing every value above (or below) a certain threshold with the threshold. Compute the winsorized mean of the column* `Sepal.Length`*, with threshold 7. Do this by first replacing every value > 7 with 7 and then computing the mean (the correct answer is 5.80533).*

2.4 Special Values

Similar to most programming languages, R has provisions to represent exceptional values such as ±`Inf`, `NaN` (not a number), `NA` (missing value), and `NULL`.

The value `Inf` represents the result of a numerical calculation such as `1/0`, which, when written as $\lim_{x\to 0} 1/x$, would yield ∞. Not a number, or `NaN`, results from computations such as `0/0` or `Inf-Inf`, where even taking a limit is undefined. We will be more precise about these concepts in Section 3.1.3. For now, we only mention that conceptually, one can think of the number system with which R computes is the augmented real line

$$\mathbb{R} \cup \{\pm\infty\} \cup \mathtt{NaN} \cup \mathtt{NA}, \tag{2.1}$$

where NA represents a missing value. All of R's arithmetic operations and mathematical functions are closed on this set. That is, if you feed any operation element(s) from (2.1), the result will be in the same set.

```
6/0
## [1] Inf
6/0  - Inf
## [1] NaN
```

In general, you can a reason about Inf as if it is an ordinary number, with a few special rules you would expect, such as Inf + x == Inf for any finite number x. This also means that it can be detected using standard boolean operators.

```
x <- c(1,2,Inf)
x == Inf
## [1] FALSE FALSE  TRUE
```

The NaN value is conceptually a bit different. Informally, it can be understood as an 'undefined' value. Since we cannot be definite about the outcome when comparing an undefined value with another value, defined or undefined, any comparison with NaN will result in NA.

```
x <- c(1,2,NaN)
x == NaN
## [1] NA NA NA
```

In fact, any calculation involving NaN will yield something undefined; either a missing value or NaN but never a number. The way to detect NaNs is with a special detector function called is.nan.

```
is.nan(x)
## [1] FALSE FALSE  TRUE
```

For numeric data, there is a convenient function is.finite that establishes whether a value from the set of Eq. (2.1) represents a finite number.

```
is.finite(c(2,1,3,NA,7,Inf,NaN))
## [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
```

The value NULL can be thought of as the empty set. Unlike all the other special values, NULL has no type (its class is NULL) and length zero.

```
length( c(1,NULL,2))
## [1] 2
length( c(1,NA,2))
## [1] 3
```

Since NULL has no length and is not a vector (try is.vector(NULL)), comparing a vector with NULL needs to be defined separately. In R, such comparisons yield a logical vector of length zero.

```
c(1,2,3) == NULL
## logical(0)
```

A place where one encounters NULL is when requesting a column from a data.frame or an element from a list that does not exist.

```
iris$somecolumn
## NULL
```

To test for NULL occurrence, use function is.null.

```
is.null(iris$somecolumn)
## [1] TRUE
```

2.4.1 Missing Values

R is one of the few languages where computing with missing values is embedded deep within the language. Since missing data is a fact of life, this is one of the features making R particularly useful for working with data. One can think of NA as a placeholder in a vector.

```
x <- c(1.4,3.2,NA,7.1)
```

Here, the vector x has four known values at positions 1, 2, and 3, and at position 4 there should be a number, but its value is currently unknown. As a consequence, not every computation with x can be completed. For example,

```
x + 1
## [1] 2.4 4.2  NA 8.1
```

yields one NA since the result of adding 1 to the third position is unknown. Moreover,

```
mean(x)
## [1] NA
```

also yields NA. Since one of the values necessary to compute the mean is missing, the mean cannot be computed. R is so consistent with treating missing values that one may adopt the following slogan: "if one of the input values is missing, the result shall be missing". There are a few exceptions however; an example is given in Exercise 2.4.6.

In particular, this means that any comparison of NA with a value will yield NA, including comparing NA with itself.

```
NA == 27
## [1] NA
NA == NA
## [1] NA
NA == "foo"
## [1] NA
```

In order to detect missing values, use is.na.

```
x <- c(1,1,2,NA,5)
is.na(x)
## [1] FALSE FALSE FALSE  TRUE FALSE
```

Many of R's statistical functions, such as mean, var, sd, quantile, and so on, have an optional argument allowing you to ignore missing values.

```
mean(x, na.rm=TRUE)
## [1] 2.25
```

Here, the `rm` in `na.rm` stands for 'remove'. This means that the missing value is completely ignored, and the mean is computed in this case as $(1.4 + 3.2 + 7.1)/3$. A special case is the function `table`, which cross-tabulates occurrences of text or categorical values in one or more vectors. This function has an argument `useNA` with the options `"no"` (ignore positions where `NA` occurs), `"ifany"` (add `NA` as category if one or more occur), and `"always"` (always add `NA` as extra category).

Although missing values are always printed as `NA`, they do come in different types. R has several missing-value-representing constants:

`NA_integer_`

`NA_real_`

`NA_character_`

`NA_complex_`

These constants can be used whenever you need to force a missing value to be of a specific type. Missing `logical` values are represented by `NA` itself, which therefore has the role of representing a 'pure' `logical` missing value as well as the general missing-value representation. Since there is no `NA_raw_`, it is not possible to represent missing `raw` values in a way that is standardized by R.

Exercises for Section 2.4

Exercise 2.4.6 *Predict the outcome of the following R statements and explain the reasoning behind your answer:*

a) `1 + NA`
b) `mean(c(1,NULL,3))`
c) `Inf/Inf`
d) `sin(Inf)`
e) `NULL == NA`
f) `NA | TRUE`

2.5 Getting Data into and out of R

One of the features that makes R such an open ecosystem is its capability for reading from and exporting to many different data formats. This is to no small extent, thanks to the packaging system, which enables authors other than the R core team to add functionality, including capabilities to read new data formats.

Base R comes with the capability to read or write data from or to two sorts of files: text files (e.g., `csv`) and R's native `.RData` format. To store a dataset in `.RData` format, simply use the `save` function.

```
myiris <- iris
save(myiris, file="myiris.RData")
```

At a later time, you can reload the data with the `load` command.

```
load("myiris.RData")
# check that the data frame is indeed loaded.
ls()
```

The `.RData` format is not limited to rectangular dataset. Any data stored in R's memory can be exported to an `.RData` file.

```
# create a linear model
m <- lm(Sepal.Length ~ Sepal.Width + Species, data=iris)
# store the model and the dataset
save(m, iris, file="mymodel.RData")
```

Upon `load("mymodel.RData")`, the variables `m` and `iris` will be made available in R's workspace.

Tabular data can be exported to a delimited text file using one of the `write.-` functions. For example, the statement

```
write.csv(iris, file="myiris.csv", row.names=FALSE)
```

exports the iris dataset to a comma-separated file. We provide the option `row.names=FALSE` to prevent a spurious extra column with row numbers to be exported. Data can be read using the `read.csv` function.

```
my_iris <- read.csv(file="myiris.csv")
```

A `csv` file is a text file that has column headers on the first line and a single data record on each consecutive line. There are two common types of `csv` files. In the first type, commas are used to separate columns and a point as decimal mark. This is the type that was exported and imported in the previous example. The second type of `csv` uses a semicolon (`;`) to separate columns and a comma as decimal mark. The functions `read.csv2` and `write.csv2` follow the latter convention.

2.5.1 File Paths in R

A file path is a string that points you to the location on a file system. Different operating systems use different conventions for marking directories in a file path. Unix-alike systems[1] use file paths that resemble web addresses and use a forward slash (/) to separate directories, while Windows uses the backslash (\). R basically follows the unix/web convention where directories are separated by forward slashes. If so desired backslashes can be used, but each backslash has to be written twice since it also has a special meaning as escape character. This means that the following file paths are equivalent:

```
"C:/My Documents/myfile.csv"
"C:\\My Documents\\myfile.csv"
```

and the following will not be recognized as a valid file path by R:

```
"C:\My Documents\myfile.csv"  # not a valid file path.
```

[1] To be precise, operating systems that follow the POSIX standard.

There are a number of utilities that facilitate working with file paths; some of the most important are shown in the following table:

`file.path`	Combine several strings to a file path
`dir`	Show the files in a directory
`file.exists`	Check whether a file or directory exists
`file.info`	Retrieve basic information from a file
`tempfile`	Create a random temporary filename
`tempdir`	Get the name of the current temporary directory, used by R.

To get a feeling for the possibilities, try the following code:

```
fl <- tempfile(fileext=".csv")
write.csv2(iris, file=fl)
file.exists(fl)
file.info(fl)
```

2.5.2 Formats Provided by Packages

There are a number of packages allowing users to read data from a wide variety of (proprietary) data formats. Providing tutorials on these packages is beyond the scope of this book, but we will point out a few packages and the formats they support.

Package `foreign` provides functions for several formats including SAS XPORT, SPSS, Stata, minitab, and Octave. The `readr` package allows for faster reading of tabular text files than base R's functionality, which is interesting when you need to read either large files and/or a great many number of files. The `haven` package reads and writes SAS files in Stata and SPSS formats. The `memisc` package also allows for importing (subsets of) SPSS files. Package `readxl` can be used to read excel files.

An overview of data import and export possibilities is also given at the following website, hosted by the R project:

```
https://cran.r-project.org/doc/manuals/r-release/R-data.html
```

2.5.3 Reading Data from a Database

Retrieving data from a database always works in three steps. First, create a `connection` to the database. Second, SQL statements can be fed to the database through the connection, and results may be retrieved. Finally, the connection can be closed again. Package `DBI` offers a generic way to connect to databases, while packages such as `RODBC` offer connection to any database complying to the Open Database Connectivity standard.

As a small illustration, in the following example, we create a simple SQLite database, perform a few operations, and close it again. The `RSQLite` is used to create and connect to the database.

```
library(RSQLite)
# set up an in-memory SQLite database.
db <- dbConnect(SQLite())
# create a table in the database named "IRIS", and load the iris dataset.
dbWriteTable(db, name="IRIS", value=iris)
## [1] TRUE
# execute a query and retrieve its result
```

```
dbGetQuery(db, "select * from IRIS where [Sepal.Width] = 4.2")
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.5         4.2          1.4         0.2  setosa
# close the connection
dbDisconnect(db)
```

2.5.4 Working with Data External to R

Methods where one works with data external to R, that is, data which is not, or only partially, loaded into R's memory become beneficial when the dataset to be processed becomes too large to load into memory. One can distinguish two basic approaches: either the data gets processed chunkwise, loading a chunk of data into R memory, updating a calculation, and unloading the chunk, or one can translate R statements to a language understood by the data management system.

Packages that allow for chunkwise processing of data stored locally on file include LaF and ff, possibly combined with ffbase. Both packages export an object that looks and feels a lot like a data frame, except that in the background it is really a pointer to a file and that operations on it are processed chunk by chunk. The LaF package allows you to connect to a large ASCII file on disk, while ff requires the user to translate a data file to ff format before it becomes available in R.

There are several systems that support translating R to statements native to a data management system. For example, the dplyr package allows processing of tabular data stored in a database by translating R statements to the relevant SQL dialect. Of course there is a limit to what R statements can be translated, since not every statistical model in R has a SQL counterpart. Besides the dplyr package, which is available from the CRAN archive, there are several commercial database vendors that offer partial R interpretation.

Finally, we note that popular big data systems such as Hadoop and Spark can be used from a local R system, for example, through the rhadoop, sparklyr, and sparkR packages. The latter is not published on CRAN, but it comes with the Apache Spark distribution.

2.6 Functions

The purpose of a function is to abstract away details of a computation. For example, when a user types sqrt(2) at the R command line, (s)he is only interested in the value of the output: a reasonable approximation of the square root of 2 and not in the idiosyncrasies that come with the numerical algorithm that is used to approximate it. The numerical algorithm was developed by someone specialized in such matters and offered to users through a convenient interface: all one needs to know is the name of the function (here, sqrt), and what arguments it accepts (here, just one argument, and it is called x).

Functions are at the heart of the R language. In fact, each and every computation we performed in the previous section has been performed by functions, even when it did not look like a function. For example, even the statement

```
1 + 1
```

is just a pretty looking shorthand for the following function call:

```
`+`(1,1)
```

Since functions are such an important part of the language, it is important to know how to work with them and to have at least enough understanding of them to write simple functions.

In the following sections, some basic properties of R functions will be discussed. For advanced topics, the reader is referred to, for example, Wickham (2014a).

2.6.1 Using Functions

Many functions in R accept more than one argument. In many cases, there is a single main argument and several arguments that control the behavior of the function. These controlling parameters often have default values assigned to them, so you do not have to pass them every time. Here is an example:

```
data <- c(1,7,NA,10)
# default behavior of 'mean': return NA
mean(data)
## [1] NA
```

The default behavior can be changed by passing the optional argument `na.rm`, causing `mean` to ignore missing values.

```
mean(data, na.rm=TRUE)
## [1] 6
```

The help file for each function shows what the default values are in the 'usage' section. Try `?mean` to see what arguments are available, and what the default values are.

You can assign values to arguments in any order if their names are mentioned explicitly.

```
mean(x=data, na.rm=TRUE)
## [1] 6
mean(na.rm=TRUE, x=data)
## [1] 6
```

Using named arguments is recommended since it makes code easier to read. If names are not used, R matches arguments by order. The order is the order mentioned in the help file of the function.

2.6.2 Writing Functions

The structure of an R function is not all that different from a function as you once learned in math class. Recall that a mathematical function is defined by a *name*, a set of *arguments* from some domain, and a prescription that tells you how to combine the arguments into the result. For example, consider the mathematical function definition,

$$f(x) = x^2 - x - 1.$$

The name of this function is f, its argument is called x, and the prescription is given by $x^2 - x - 1$. After this definition, anyone with some understanding of mathematical notation can compute $f(3)$, for example.

In R, the above function is defined as follows:

```
f <- function(x){
  x^2 - x - 1
}
```

Here, the part `f <-` tells R that `f` is the name of something new. The keyword `function` is used to tell R that `f` is a function and has an argument called `x`. The function prescription is the part between { and }curly brackets.

```
x^2 - x - 1
```

Now, everybody who defined the function as above can compute $f(3)$ by typing the following at the command line:

```
f(3)
## [1] 5
```

This way, one can reuse the piece of code in the body `x^2 - x - 1` as often as desired without retyping it.

The Function Body and Return Values

A function body does not need to consist of a single line but can contain arbitrarily many R expressions. For example, we may split the computation of `f` in two lines:

```
g <- function(x){
  out <- x^2 - x - 1
  out
}
```

Here, function `g` first computes the value of `x^2 -x - 1` and stores it in a variable called `out`. This variable is hidden using `g` and can only be 'seen' by a code that is written in `g`'s function body, after `out` has been defined. In the next and final line, the computed value is returned. This is done by stating the name of the variable that stores the value to be returned.

2.7 Packages Used in this Book

There are a number of R extension packages used throughout the book. Below, these packages are listed with proper citations and a short description. Every package mentioned here is available through CRAN unless mentioned otherwise.

Amelia (Honaker *et al.*, 2011). Multivariate imputation for numerical values based on the multivariate normal distribution.

DBI (R Special Interest Group on Databases, 2014). Provides a high-level interface for database management from R. The package defines a set of standard classes that can be inherited from packages defining an explicit interface (such as `RODBC` and `RSQLite`).

daff (Fitzpatrick *et al.*, 2017). Diff, patch, and merge for data.frames.

dcmodify (van der Loo and de Jonge, 2015a). Provides methods for rule-based data modification. Rules are treated as object of computation and may be stored, analyzed, and processed.

deductive (van der Loo and de Jonge, 2017). Deductive correction, deductive imputation, and deterministic correction of numerical data.

docopt (de Jonge, 2016). Specify and parse command-line options for `Rscript`.

dplyr (Wickham *et al.*, 2014). Provides convenient interfaces for operations on tabular data, where the result is tabular as well. It integrates very well with the pipe operator of the `magrittr` package.

errorlocate (de Jonge and van der Loo, 2016). Provides error localization capabilities based on the principle of Fellegi and Holt.

ff (Adler *et al.*, 2014). Makes it possible to compute with large data files stored on disk. Such data needs to be translated to `ff` format first.

ffbase (de Jonge *et al.*, 2015). Provides basic statistical functionality for objects stored in `ff` format.

ggplot2 (Wickham, 2009). Provides an R-based implementation of the Grammar of Graphics of Wilkinson (2005).

haven (Wickham and Miller, 2015). Facilitates reading from and writing to several popular statistical data formats.

igraph (Csardi and Nepusz, 2006). Implements functionality for manipulation, analyzing, and plotting of graphs.

jsonlite (Ooms, 2014). Facilitates importing data stored in `json` format to R and to export R objects in `json` objects.

kernlab (Karatzoglou and Feinerer, 2007). Implements a number of kernel functions popular in machine learning.

LaF (van der Laan, 2015). Provides methods for accessing and processing tabular ASCII data stored on disk.

lubridate (Grolemund and Wickham, 2011). Provides utility functions for parsing date formats and for computing with dates.

lumberjack (van der Loo, 2017a). A function composition (pipe) operator and extendable framework for tracking changes in data.

magrittr (Bache and Wickham, 2014). Provides chaining operators that allows one to easily call a sequence of functions where each next function accepts the output of the previous function.

memisc (Elff, 2015). Provides a suite of tools for managing survey data, including import functionality from several statistical data formats.

mice (van Buuren and Groothuis-Oudshoorn, 2011). Multivariate imputation by chained equations.

missForest (Stekhoven and Bühlmann, 2012). Nonparametric missing value imputation for mixed-type data based on an iterative Random Forest algorithm.

microbenchmark (Mersmann, 2014). Provides functionality for accurately measuring and comparing the performance of R statements, down to nanoseconds.

plyr (Wickham, 2011). Provides convenient ways for group-based aggregation of data stored as `data.frame`, `array`, or `list`.

qdapRegex (Rinker, 2014). Provides a set of convenience functions for text processing based on common regular expressions.

randomForest (Liaw and Wiener, 2002). Interface to the famous random forest algorithm of Breiman (2001).

readr (Wickham and Francois, 2015). Provides fast functions for reading tabular text data.

rex (Ushey and Hester, 2014). Facilitates definition of possibly complex regular expressions in a more human-readable format.

reshape2 (Wickham, 2007). Provides simple interfaces for reshaping (pivoting) rectangular datasets.

rhadoop (Revolution Analytics, 2015). A collection of five R packages that allow for Hadoop connectivity from R. It is not available on `CRAN` but can be downloaded from the following website: `https://github.com/RevolutionAnalytics/RHadoop/wiki`.

RODBC (Ripley and Lapsley, 2015). Provides connectivity to databases that support the Open DataBase Connectivity standard.

rspa (van der Loo, 2017b). Adjust numerical data to fit equality and/or inequality constraints.

RSQLite (Wickham and Francois, 2014). Provides connectivity with SQLite databases.

rvest (Wickham *et al.*, 2014). Provides web scraping functionality.

sparkR (Venkataraman, 2013). Provides a front-end for the Spark distributed computing system. This package is not available on CRAN but comes with the Apache Spark installation.

simputation (van der Loo, 2017c). Provides a uniform interface to many commonly used imputation methods. Fits in the `magrittr` pipeline.

stringdist (van der Loo, 2014). Provides approximate text matching and string distance functionality based on a number of popular string metric algorithms.

stringi (Gagolewski and Tartanus, 2014). Provides an interface to the ICU library for character string processing.

stringr (Wickham, 2010). Provides a number of convenience functions for string processing based on the `stringi` backend.

textcat (Hornik *et al.*, 2013). Provides text categorization methods based on *n*-gram analyzes.

tm (Meyer *et al.*, 2008). Provides basic text mining infrastructure for R.

validate (van der Loo and de Jonge, 2015b). Provides infrastructure for managing and applying data validation rules.

validatetools (de Jonge, van der Loo, 2017). Provides checking and simplifying data validation rule sets.

VIM (Templ *et al.*, 2016). Visualization and imputation of missing values.

yaml (Stephens, 2014). Provides functionality to read and write data in yaml format.

Zelig (Imai *et al.*, 2008). A framework for statistical analysis, including for working with multiply imputed datasets.

3

Technical Representation of Data

Ideally, data analysts should not have to worry too much about how exactly data is technically stored in computer memory. Indeed, much effort in computer engineering has gone into trying to abstract away technical details, such as how a real number is represented as a sequence of bits.

In practice, however, consequences of technical choices eventually pop up. For example, there is probably hardly any computer user who has not at some point seen symbols similar to □ or � appearing in their text editor. Such symbols indicate that the program displaying the text was not able to translate the string of bytes it read into readable characters. As a second example, consider the output of the following calculation in R.

```
if ( 1 - 0.9 == 0.1 ) print("ok") else print("oh no!")
## [1] "oh no!"
```

Although it seems reasonable to expect `"ok"`, apparently $1 - 0.9$ is not precisely equal to 0.1 for a computer although the difference is admittedly small.

```
(1-0.9) - 0.1
## [1] -2.775558e-17
```

These examples are forms of what is commonly termed *abstraction leakage*: issues that have to do with the underlying technical representation of data exposed to the user. In the case of the misrepresentation of text, the problem is that the text editor assumes text is stored in one encoding, while it is stored in another. In the case of the ‘failed’ calculation, the small difference between the expected and obtained result is the result of a trade-off: the lack of precision is small enough to be well compensated by the vast gain in speed for nearly all applications that involve statistical data processing. We will encounter numerical precision issues further on in this book, when we discuss algorithms for error localization and value adaption.

This chapter discusses the technical representation of the most important data types: integers, real numbers, and text while exposing caveats one may encounter while programming with data. Special attention is paid to representation of these data types in R.

3.1 Numeric Data

3.1.1 Integers

Mathematically, the set of natural numbers, denoted $\mathbb{N}$, consists of the counting numbers $\{0, 1, 2, \ldots\}$. The set of integers consists of all positive and negative natural numbers: $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$.

Technically, both natural numbers and integers are represented as a sequence of N bits. The number of bits used to represent a whole number depends on the software one uses. In low-level languages like `C`, `C++`, or `Java` as well as in some higher level languages, like `octave` (but not R), the value of N can be chosen. Common values are 8-, 16-, 32-, or 64-bit representations because these correspond to integer multiples of the size of computing units on the most common hardware (bytes). Once N is fixed, the maximum number of either natural numbers or integers that is representable by the bit sequence is fixed. Most languages also provide a way to choose between a representation of unsigned integers ($\mathbb{N}$) or signed integers ($\mathbb{Z}$).

The main consequence of this representation is that contrary to the definitions of $\mathbb{N}$ and $\mathbb{Z}$, the number of representable integers is not infinite. The actual *range* of integers that can be represented depends on how integers are translated to bit sequences.

Table 3.1 illustrates the conventions based on a three-bit representation. Unsigned integers $n \in \mathbb{N}$ are represented in standard binary representation. That is, a positive integer is represented as a sequence $b_0 b_1 \cdots b_n$ where the bits b_t are uniquely determined by

$$n = \sum_{t=0}^{N-1} b_t 2^{N-t-1}. \tag{3.1}$$

The range of unsigned integers is given by

$$0 \leq n \leq 2^N - 1. \tag{3.2}$$

For integers $z \in \mathbb{Z}$, there are three conventions in use. The first and most simple one is the *signed integer* convention, where the first bit encodes the sign and the next $N - 1$ bits the magnitude. This implies that 0 has two representations, which may be denoted $\{+0, -0\}$. The range of numbers that can be represented in this convention is given by

$$1 - 2^{N-1} \leq z \leq 2^{N-1} - 1. \tag{3.3}$$

The second convention is called *one's complement*. Here, the representable range is the same as in the *signed integer* convention, but the order of the negative numbers is reversed. Observe that this implies that changing the sign of a number implies taking the complement of each bit. For example, in the three-bit representation, +2 is represented

Table 3.1 Conventions for integer representation in three-bit representation.

Bit sequence	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
Unsigned integer	0	1	2	3	4	5	6	7
Signed integer	+0	+1	+2	+3	−0	−1	−2	−3
One's complement	+0	+1	+2	+3	−3	−2	−1	−0
Two's complement	0	+1	+2	+3	−4	−3	−2	−1

by 010 and −2 by 101. Zero is represented by both 000 and 111. The third convention is called *two's complement*. In this representation, the negative numbers are shifted one, avoiding the double zero. The representable range is now asymmetric about 0:

$$-2^{N-1} \leq z \leq 2^{N-1} - 1. \tag{3.4}$$

Starting from a negative integer, its positive counterpart can be found by taking the complement of each bit and subtracting one from the result. The two's complement is the most popular since it allows for simpler algorithms for addition and subtraction.

The fact that there are several ways to represent integers means that one cannot simply check the number of bits and compute the represented range. Instead, computer languages usually have built-in constants that indicate the limits of integer representations. In `C`, for example, the constants `INT_MIN` and `INT_MAX` hold the lower and upper limits for the range-signed 16-bit integers. The following exercises point out a few cases where things can go wrong if one disregards the finiteness of integer representations.

Exercises for Section 3.1.1

In R, integer data is stored as a 32-bit signed integer on any platform (including 64-bit platforms), usually in the two's complement representation.

Exercise 3.1.1 *Explain which of the following numeric codes can or cannot be stored as integer data in R.*

a) United States Social security numbers, consisting of nine figures.
b) Universal Product Code-A. This is a barcode convention that labels products with 12 decimal digits.
c) Phone numbers consisting of 10 digits, all starting with zero.

Exercise 3.1.2 *Explain why the following R command yields an error.*

```
read.table(textConnection("123451234512345")
  , colClasses="integer")
```

'Integer overflow' is what happens when a computation on two integers yields a number that is out of the representable range, for example, `2 * MAX_INT`. The result of such an instruction depends on the programming language; or if the language does not define that behavior, it depends on the specific implementation (compiler). In `C`, for example, behavior under integer overflow is not defined. The following exercise is aimed to find out how R handles integer overflow.

Exercise 3.1.3 *Execute the following statement and determine the* `class` *of* `max_int`*.*

```
max_int <-.Machine$integer.max
```

Now execute the following statements and explain what happens (recall that the postfix `L` *behind a numeric literal indicates integer).*

```
 x1 <- 2L * max_int
 x2 <- 2 * max_int
```

Hint: determine the `class` *of* `x1` *and* `x2`.

3.1.2 Integers in R

In R, there are two sorts of integers, whose behavior depends on their use. First, there is *integer data*, as discussed in the previous section. This refers to the values that are stored in a vector of class `integer`. These values are represented by a 32-bit signed integer representation, regardless of whether you are using a 32- or 64-bit platform. The largest representable integer is stored in the hidden global list of constants called `.Machine` and can be obtained as follows.

```
 .Machine$integer.max
## [1] 2147483647
```

which, in our case, corresponds to $2^{31} - 1$. Assuming the two's complement representation, one expects that the largest representable negative integer is equal to -2^{31}. However, trying to store such a number as integer yields a warning.

```
 int_min <- as.integer(-2^31)
 ## Warning: NAs introduced by coercion to integer range
 one_above_min <- as.integer(-2^{31}+1)
```

The reason is that in R, the largest negative number that can be represented by a 32-bit signed integer representation is used to encode NA, decreasing the actual range of integers with one. Overflow situations are handled by coercing an integer to `numeric`, which is what happened in Exercise 3.1.3 of the previous section. In fact, R will quietly coerce a number to `numeric` upon any calculation of which the result is not integer, so executing `sqrt(2L)` does not give any errors.

Users who need to do calculations on integers larger than `.Machine$integer.max` have two options. First, one may simply be satisfied that larger numbers are represented as `numeric`. If the type is important, there are several packages that offer support for large integer data. Package `bit64` (Oehlschlägel, 2014) offers a class called `integer64`, which can store large integers at the expense of eight bytes each. The package is based on `C` code that is optimized with a lot of attention to performance, sometimes outperforming R's native 32-bit integer routines. One example where the use of 64-bit integers may be beneficial is when one needs to store columns with long numeric codes (e.g., product codes) and performance is very important.

The second use of 'integers' in R is not as data but as an *index* into data vectors. In versions of R $\geq$ 3.0.0, one can create index vectors of lengths that go beyond `.Machine$integer.max`, if you run R on a 64-bit platform (R Core Team, 2013). Since integers cannot go beyond `.Machine$integer.max`, vectors of class `numeric` with large values may be used as integer.

For reasons to become clear in the next section, R internally uses up to 53 bits for storing a signed integer index (recall that R allows for negative indices to leave out elements). This means that the maximum vector length for integers stored in the two's complement convention is $2^{52} - 1 \approx 0.45 \times 10^{16}$.

Most importantly, this means that matrices or arrays, whose individual dimensions do not exceed `.Machine$integer.max` but have more elements in total, can now be represented. There are some limitations on the use of linear algebra on such large matrices: the underlying libraries used by R are not always prepared for matrices of large sizes. Statistical operations like `colSums`, and so on, may be expected to work. The fifth volume of the R Journal (issue 1) lists some subtle exceptions when using long vectors. We deem those sufficiently idiosyncratic not to treat them here.

Exercises for Section 3.1.2

The default data reading functions in R such as `read.table` will automatically detect the appropriate storage format for each column that it reads, unless the formats are specified explicitly by the user. It will try to store numbers as an `integer` unless numbers are either too large or explicitly denoted as a real number.

Exercise 3.1.4 *Predict the column types of* `d1` *and* `d2` *after executing the following statements.*

```
d1 <- read.table(textConnection("1324665248"))
d2 <- read.table(textConnection("4827647632"))
```

Check your result and explain what happened. If you were asked to read in a file containing 10-digit product codes, what storage format would you choose?

3.1.3 Real Numbers

The mathematical set of real numbers (ℝ, points on a line) is generally approximated with the standard *double precision* floating point representation. It is an approximation in two ways: first of all, the number of elements that can be represented is finite, and, secondly, since each number is represented as a finite sequence of bits, there is usually a roundoff error.

The double precision floating point representation is fixed in an international standard, which is proposed and maintained by the International Association of Electrical and Electronics Engineers IEEE (2008). The version at the time of writing was adopted in 2008 and it is widely supported by hardware and compilers. The standard defines not only the storage format of floating point numbers but also a number of common operations on those numbers including arithmetic operations and a number of mathematical functions like `exp` and `log`. The broad acceptance and adoption of the standard implies that such basic calculations are for a very large part reproducible across different architectures and softwares without effort on behalf of the user—a situation which is not entirely comparable with the state of affairs in text representation (discussed subsequently). The IEEE standard actually defines several numeric representations, but the double precision format is of most use in statistical applications. A second format one may encounter is the *single precision* format, which is currently widely used as the format for graphics processing unit (GPU) processing.

3.1.4 Double Precision Numbers

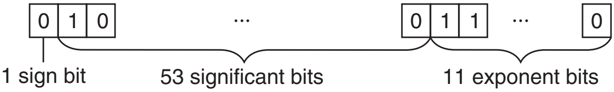
To discuss the merits and demerits of floating point arithmetic, we will start at the bottom, that is, the bit-wise representation of double precision numbers and work our way up to the real numbers they represent. Along the way we will discuss the consequences of this standard for working with numerical data.

Double precision represents numbers in scientific notation, in base 2. That is, each number is written as

$$(-1)^s \times m \times 2^e, \tag{3.5}$$

where the numbers s (sign), m (significant), and e (exponent) are denoted in binary format. There is some freedom in this specification, since, for example, the binary numbers 0.1×2^1 and 1.0×2^0 are equal. To remove ambiguity and guarantee a level of precision, the IEEE standard prescribes that *normal numbers* are represented such that the first bit is always 1. This means that the value of e is minimized (and thus the significant maximized), except when the represented number is exactly zero.

Double precision numbers are represented by a sequence of 64 bits, which are interpreted as shown in the following diagram.



Here, the first bit represents the number's sign where 0 represents +. The next 53 bits represent the nonnegative significant, which should be interpreted as a binary number with the radix ('decimal point') immediately following the first digit: $m \equiv m_0 \cdot m_1 m_2 \cdots m_{52}$. Its decimal representation can be computed as

$$\sum_{j=0}^{52} m_j 2^{-j}. \tag{3.6}$$

The final 11 bits represent a signed integer that is used to encode the value of the exponent as well as the special cases, NaN and ±Inf. Here is how double precision numbers are interpreted.

- Numbers where $s = \pm 1$, $m \geq 1$, $-1022 \leq e \leq 1023$, and $e \neq 0$ are called *normal numbers*. Normal numbers have $m_0 = 1$ and are specified up to 52 bits behind the binary point.
- Numbers where $s = \pm 1$, $0 \leq m < 1$, and $e = 0$ are called *subnormal numbers*. These are to be interpreted as the tiny values given by $(-1)^s \times m \times 2^{-1022}$. Subnormal numbers have $m_0 = 0$ and therefore are specified up to less than 52 bits behind the binary point.
- Numbers where $s = \pm 1$, $m = 0$, and $e = 0$ represent ± 0.
- Numbers where $s = \pm 1$, $m = 0$, and $e = 1024$ represent $\pm$ Inf.
- Numbers where $s = \pm 1$, $m > 0$, and $e = 1024$ represent NaN.

From this definition we see that rather than the set of real numbers, the set of double precision numbers conceptually represents an extended set given by

$$[-\infty, -0] \cup [+0, +\infty] \cup \{\mathsf{NaN}\}. \tag{3.7}$$

Indeed, the concepts of a signed zero and not-a-number are not normally part of real calculus but may nevertheless result from computations on double precision numbers. The same is true for $\pm\infty$; recall that although there are an infinite number of arbitrarily large numbers in $\mathbb{R}$, infinity itself is not an element of $\mathbb{R}$. The correspondence between infinity in real calculus and Inf in the IEEE standard is as follows: given a function $f : \mathbb{R} \to \mathbb{R}$ and its double precision counterpart f, then

$$\text{if } \lim_{x\to a} f(x) = \infty \text{ then } \mathsf{f}(\mathsf{fl}(a)) = \mathsf{Inf}, \tag{3.8}$$

where $\mathsf{fl}(a)$ denotes the double precision representation of the real number a. In particular, in floating point arithmetic we have $1/\pm 0 = \pm\mathsf{Inf}$, corresponding to the limit $\lim_{x\to\pm 0} 1/x$, where x approaches zero from above (+) or below (−). Furthermore, any calculation that results in a number greater than the largest double precision number results in Inf as well. The value NaN is reserved for results of calculations whose outcome cannot be an element of $\mathbb{R}$, such as Inf − Inf or $0/0$. More generally, any computation where one of the operands is NaN will result in NaN.

A few properties can easily be derived from the interpretation of the rules listed. First, using Eqs (3.5) and (3.6), setting all $m_i = 1$ $(0 \leq i \leq 52)$ and $e = 1023$ we see that the largest representable positive number is given by

$$\sum_{j=0}^{52} 2^{1023-j} \approx 1.8 \times 10^{308}. \tag{3.9}$$

The smallest positive number that can be represented with the full 52 bits of precision behind the binary point is computed by setting $m_0 = 1$, $m_1 \ldots m_{52}$ equal to zero, and $e = -1022$. Using the same equations we get

$$\varepsilon(0) = 2^{-1022} \approx 2.2 \times 10^{-308}. \tag{3.10}$$

The significance of this number is that it is the smallest number that differs from zero without additional roundoff error, which is why we call it $\varepsilon(0)$ here. We can generalize this notion a little bit and define $\varepsilon(x)$ as the difference between the floating point number x and its successor. The case for $x = 1$ is easily computed. Since there are 52 bits behind the decimal point, we have $\varepsilon(1) = (1 + 2^{-52}) - 1 \approx 2.2 \times 10^{-16}$.

3.1.5 The Concept of Machine Precision

As discussed earlier, the value of the rounding error made in the representation of real numbers depends on the represented number: the density of double precision numbers as a subset of real numbers is highest around zero and decreases exponentially at larger (positive or negative) numbers. It is therefore customary to work with the *machine precision U*, which is the smallest positive number such that

$$\left|\frac{x - \mathsf{fl}(x)}{x}\right| \leq U$$

for all floating point numbers $\mathsf{fl}(x)$. Although the precise value of $\mathsf{fl}(x) - x$ depends on the rounding scheme that is used, it is clear that $|\mathsf{fl}(x) - x| \leq \varepsilon(x)/2$, so we get

$$\left|\frac{\varepsilon(x)}{2x}\right| \leq U.$$

We now write $|\mathsf{fl}(x)| = m(x)2^e(x)$, where, m and e represent the sign, mantissa, and exponent of the floating point representation of x. We have, for the double precision format:

$$|\varepsilon(x)| = (m(x) + 2^{-52})2^{e(x)} - m(x)2^{e(x)} = 2^{-52}2^{e(x)}.$$

Similarly, the value $|x|$ may be written as $\overline{m}(x)2^{e(x)}$, where $\overline{m}(x)$ is the mantissa with an infinite number of digits, so we have

$$\left|\frac{2^{-52}}{2\overline{m}(x)}\right| \leq U.$$

The left-hand side of this equation is maximal when $\overline{m}(x)$ is minimal. Since we defined the mantissa such that $1 \leq \overline{m}(x) < 2$, we get

$$U = 2^{-53} = \frac{\varepsilon(1)}{2} \approx 1.1 \times 10^{-16}. \tag{3.11}$$

Although we have done the calculation explicitly for 64-bit double precision numbers here, the conclusion $U = \varepsilon(1)/2$ only depends on the part of the demand that the mantissa is larger than or equal to 1.

3.1.6 Consequences of Working with Floating Point Numbers

Working with double precision numbers means working with rounded numbers. This implies not only that a rounding error is made when representing numbers but also that a rounding error is introduced at each calculation. It is beyond the scope of this book to extensively discuss how rounding errors propagate through calculations. Instead, we point out a few classical results that give some idea of the severity of problems that may arise because of roundoff errors. We focus our attention on solving linear systems of equations, as we will reencounter such systems several times in this book.

The problem of solving systems of the form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3.12}$$

with $\boldsymbol{A}$ a not necessarily square matrix is encountered in many statistical applications. However, since we need to represent $\boldsymbol{A}$ as a matrix of floating point numbers, computer programs are really trying to solve

$$(\boldsymbol{A} + \Delta\boldsymbol{A})\boldsymbol{x} = \boldsymbol{b} + \Delta\boldsymbol{b},$$

where $\Delta\boldsymbol{A}$ and $\Delta\boldsymbol{b}$ are roundoff errors. If Gaussian elimination is used to solve Eq. (3.12), it can be shown that a bound on the relative error in the solution due to rounding errors in $\boldsymbol{b}$ is given by [Stoer and Bulirsch (2002) Section 4.4 or Golub and van Loan (1996) Section 3.5.1]

$$\frac{\|\Delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \leq \|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\|\frac{\|\Delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|,} \tag{3.13}$$

where $\|\cdot\|$ denotes a (matrix) norm and $\Delta\boldsymbol{x}$ is the difference between the actual solution $\boldsymbol{x}$ and the solution generated by the algorithm. The quantity

$$\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\|,$$

is called the condition number of matrix $\boldsymbol{A}$. It can be interpreted as a measure of the ratio between the 'size' of matrix $\boldsymbol{A}$ and its inverse. Its value depends on what norm is chosen, but the following statements are always true: $\kappa(\boldsymbol{A}) \geq 1$ and $\kappa(\boldsymbol{A}) = 1$ for any matrix that has the property $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A} = \boldsymbol{I}$. In particular, $\kappa(\boldsymbol{I}) = 1$.

An interesting special case occurs if we replace $\boldsymbol{A}$ with a positive definite $n \times n$ matrix $\boldsymbol{H}$, for example $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$. Choosing the 2-norm, the vector norms are just to the Euclidian norms and it can be shown that the condition number is then equal to

$$\kappa_2(\boldsymbol{H}) = \frac{\lambda^{\max}}{\lambda^{\min}}, \tag{3.14}$$

where $\lambda^{\max}$ and $\lambda^{\min}$ are the largest and smallest positive eigenvalues of $\boldsymbol{A}$. This immediately shows that the relative error in the solution of Eq. (3.12) increases sharply when $\lambda^{\min}$ approaches 0, that is, when $\boldsymbol{A}$ is nearly singular. In fact, the smallest eigenvalue $\lambda^{\min}$ may also be interpreted as a distance between $\boldsymbol{A}$ and the nearest singular matrix [Stoer and Bulirsch (2002), Theorem 6.4.9].

A second interesting case is when the ∞-norm is chosen. Since we consider $\boldsymbol{b}$ fixed, we have $\|\Delta\boldsymbol{b}\|_\infty/\|\boldsymbol{b}\|_\infty \approx U$, so we obtain the heuristic

$$\frac{\|\Delta\boldsymbol{x}\|_\infty}{\|\boldsymbol{x}\|_\infty} \approx \kappa_\infty(\boldsymbol{A})U,$$

where U is as defined in the previous section. Using Eq. (3.11) and taking the logarithm to base 10, we get approximately

$$\log\frac{\|\Delta\boldsymbol{x}\|_\infty}{\|\boldsymbol{x}\|_\infty} \approx \log\kappa_\infty(\boldsymbol{A}) - 16. \tag{3.15}$$

This equation is interpreted as follows: if $\kappa_\infty(\boldsymbol{A}) = 10^k$, then the solution vector computed with Gaussian elimination has approximately $16 - k$ correct digits. Even though this result is partially heuristic, and strictly only applies to a single algorithm, it does provide valuable insight into the effect of numerical instabilities on a solution.

3.1.7 Dealing with the Consequences

Since exact analysis of propagation of rounding errors through a calculation is very often unfeasible, scientific programmers typically use a few rules of thumb and standard procedures to avoid numerical difficulties.

First of all, whenever two floating point numbers have to be compared for equality, one checks their similarity to within a certain tolerance.

```
# bad practice (x and y must be scalar):
if ( x == y ) {...}

# good practice (x and y must be scalar):
if ( abs(x-y) < tol ){...}

# or, shorter (x and y may be vectors):
if ( all.equal(x,y) ) {...}
```

As tolerance, the square root of $\varepsilon(1)$ is often used, as this is provided as a constant in many programming languages. This means that precision is guaranteed up to about half the number of digits behind the comma supported by the floating point representation. For double precision numbers, which means about eight figures behind the comma, R actually has a built-in function named `all.equal` which does exactly that: it compares elements of `x` and `y` to within a tolerance whose default is indeed the square root of R's built-in constant `.Machine$double.eps`.

Secondly, it is good (and often unavoidable) practice to scale the data you are working with such that the numbers appearing in a calculation do not differ too much in size. As a rule of thumb one should avoid computations involving double precision numbers that differ more than eight orders of magnitude in a single calculation since the $\sqrt{\varepsilon(1)}$ rule is used as a default in many (publicly available) numerical software. Well-known examples include linear system solvers (including R's `solve` function) and LP solvers which usually have several tolerance settings pertaining to various parts of the numerical algorithm used. The easiest way of doing so is to start by choosing the units of measurement so that data values are on the order of, say, 1–10. In a similar discussion on scaling in the context of nonlinear parameter optimization, Nash (2014) recommends to scale values to so they are between 1 and 10. For example, if your data contains values on the order of millions of kilogram, express them in megatons, if data contains values of around 10^{-9} g/l express them in nanogram per liter.

Finally, linear systems may be rescaled by multiplication with diagonal matrices as follows. We define

$$\boldsymbol{A}^* = \boldsymbol{D}_1^{-1}\boldsymbol{A}\boldsymbol{D}_2,\ \boldsymbol{x}^* = \boldsymbol{D}_2^{-1}\boldsymbol{x}, \text{ and } \boldsymbol{b}^* = \boldsymbol{D}_1^{-1}\boldsymbol{b},$$

where $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ are diagonal column and row scaling matrices, respectively. The idea is then to first solve the more numerically stable system given by

$$\boldsymbol{A}^*\boldsymbol{x}^* = \boldsymbol{b}^*,$$

and then produce the actual result as $\boldsymbol{x} = \boldsymbol{D}_2\boldsymbol{x}^*$. Unfortunately, there is no default scaling procedure that is guaranteed to work in all cases and, in fact, scaling methods may render things worse. A fair amount of research and literature has gone into finding, without too much computational effort, scaling methods that minimize the condition number of the scaled matrix. A common heuristic used to scale matrices is to choose $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ so that

$$\sum_k |A^*_{ik}| = \sum_l |A^*_{jl}|, \text{ for all } i, j.$$

A matrix satisfying such a condition is called *equilibrated*. A simple row scaling is given by choosing $\boldsymbol{D}_1 = \boldsymbol{I}$ and scaling the rows with $D_{2,ii} = 1/\sum_j |A_{ij}|$. This ensures that all rows sum up to one. A combined row-column scaling using also $D_{1,jj} = \sum_i |A_{ij}|$ can be used as a first approximation to obtain an equilibrated matrix.

Although software for solving numerical problems often have various scaling built-in methods available, it remains a task of the user to choose or apply an appropriate scaling method for the problem at hand. Once again, even having a (near) perfect equilibrated matrix does not ensure a small condition number, so the effects of scaling should be investigated.

Exercises for Section 3.1.3

Exercise 3.1.5 *Determine the machine precision U for numbers in single precision format (hint: the mantissa consists of 23 bits).*

Exercise 3.1.6 *Compute the smallest representable positive subnormal number in double precision format.*

Exercise 3.1.7 *The fact that roundoff errors are made at each arithmetic operation also implies that computer addition is not associative. That is, in double precision arithmetic it is, in general, not true that* $(x + y) + z = x + (y + z)$. *Test this using R or any other software you use to do calculations by computing* $(0.4 + 0.3) + 0.1$ *and* $0.4 + (0.3 + 0.1)$ *and comparing the results. How big is the difference?*

Recall that the Mahalanobis distance of a vector $\boldsymbol{x}$ of random variables to a given reference vector $\boldsymbol{\mu}$ is given by

$$[(\boldsymbol{x}^T - \boldsymbol{\mu})\boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})]^{\frac{1}{2}}.$$

Its computation involves multiplication of an inverse positive symmetric matrix with a fixed vector which is equivalent to solving a linear system such as Eq. (3.12). In the following exercise we investigate the numerical stability of this distance.

Exercise 3.1.8

a) *Using the* `women` *dataset from R, compute the covariance matrix* $\boldsymbol{\Sigma}$ *between variables'* height *and* weight.
b) *Compute the condition number of the covariance matrix under the 2-norm. (Hint: see the help pages for the* `kappa` *function).*
c) *If we were to use the covariance matrix to solve linear systems (e.g., to compute the Mahalanobis distance) to how many figures behind the comma can we trust the answer?*
d) *Now, rescale the variables'* height *and* weight *with the z-transformation using R's* `scale` *function. Recompute the covariance matrix and the condition numbers. Discuss the numerical stability of the Mahalanobis distance for the scaled with respect to the unscaled variables.*

3.1.8 Numeric Data in R

The only floating point format supported by R is the double precision format, which is stored in vectors of class `numeric`. The various numerical constants related to double-precision representation can be retrieved from the global variable `.Machine$double.<variable>`. The most important is `.Machine$double.eps`, which holds the value of $\varepsilon(1)$.

Missing values, represented by NA, are stored as a specially marked NaN. Recall that a NaN is encoded by any double-precision number where the exponent $e = 1024$. This means that programmers are free to set the 53 bits in the mantissa to encode extra information. The IEEE standard explicitly mentions such a so-called payload as an option to encode user-defined special values. It also specifies that arithmetic operations involving NaN will result in a NaN. It is however not defined what happens when the payload of two NaNs differ. The result is that otherwise commutative operations may get different results when two different NaNs are multiplied.

```
NA_real_  * NaN
## [1] NaN
NaN * NA_real_
## [1] NA
```

This result depends on the compiler/hardware combination used to build R and may differ across systems (this result was obtained on a 64-bit Linux machine with an Intel chipset running GNU R compiled with the GNU Compiler Collection). This means that in R, information on prior 'missingness' of a value may be lost when it is combined with a regular NaN in a calculation. In practice, this will not often be a problem, and the function `is.na` recognizes NA as well as NaN. We encourage the readers to familiarize themselves with the behavior of `is.na` and `is.nan` as well as to try out various combinations of adding and multiplying NA variants (`NA_integer_`, `NA_real_`, `NA`) with NaN in different orders.

There are a few functions in base R pertaining to numerical stability and scaling. The function `kappa` computes (an estimate of) the two-condition number, given in Eq. (3.14); function `rcond` can be used to compute the reciprocal condition number under the one-norm (default) or the infinity norm by passing `norm="I"`. The function `scale`, by default, scales each column of a matrix-like object (`matrix`, `array`, `data.frame`) with the z-transformation. However, it accepts extra arguments to define custom scale and/or shift values to facilitate any scaling without the need for a matrix multiplication.

3.2 Text Data

There are many ways in which text can be stored in computer memory, and over time more than a thousand of such text encoding schemes have been developed for various purposes, languages, and alphabets. Along with the development of these technical standards, a rather confused mix of terminology has emerged. Although a technical report of the Unicode Consortium (Whistler *et al.*, 2008) introduces generalized terminology that should cover all systems, less precise terms such as 'character set' and 'character encoding' are still widely in use.

In the next sections we introduce the terminology used in this book and point out common pitfalls and tools related to text representation. A few encoding systems are discussed briefly before we focus on text representation in R.

3.2.1 Terminology and Encodings

We shall use the term *alphabet* to indicate a nonempty, finite set of abstract symbols. Examples of alphabets are the numbers zero to nine, the lowercase Latin alphabet, the binary alphabet $\{0, 1\}$. The fact that the alphabet is abstract means to say that it is independent of how it is represented in computer memory or in print. For example, both 'a' and '*a*' are representations of a symbol that can be described as 'the first letter of the Latin alphabet'. Similarly, one may choose various ways to represent the letter 'a' as a sequence of bytes. An alphabet is denoted with the symbol Σ. A *string* is a finite concatenation (sequence) of zero or more elements from Σ. The set of all possible strings is denoted Σ^*.

The term *character encoding* will be used to indicate a map that assigns to each character of an alphabet a unique byte or sequence of bytes. This means that given a string we can assign to it a byte sequence by concatenating the byte sequences assigned symbol by symbol. This definition covers many commonly used encoding schemes such as ASCII or UTF-8, but not all. More generally, we define an *encoding* as a reversible map that assigns to a string of symbols from an alphabet a sequence of bytes.

We will illustrate the difference between the two definitions with an example. Consider the two-symbol alphabet $\Sigma = \{a, b\}$ and we define two encodings denoted ϕ_1 and ϕ_2. The first one is a character encoding and it is defined as

$$\phi_1(a) = 0,\ \phi_1(b) = 1,$$

where we use 0 and 1 instead of byte sequences to keep the notation simple. We can easily extend ϕ_1 to the set of strings. For example, if $s = s_1 s_2 \cdots s_k$ is a string, then we define

$$\phi_1(s) = \phi_1(s_1)\phi_1(s_2) \cdots \phi_1(s_k).$$

For example, $\phi_1(abb) = 011$. The definition of ϕ_2 is a little more complicated. Given a string s, $\phi_2(s)$ is computed by

1) compute $\phi_1(s)$.
2) replace every occurrence of 11 with 2.

So, for example, $\phi_2(abb) = 02$. This example shows that ϕ_2, in general, cannot be written as a concatenation of symbol-by-symbol translations. The trade-off is that while ϕ_2 allows for a more compact storage, ϕ_1 is easier to compute.

Many encodings encountered in practice are actually character encodings. That is, they can be (de)coded on a symbol-by-symbol basis. Examples include the widely used ASCII, UTF-8, and Latin-1 character encoding standards. Encodings that cannot allow for symbol-by-symbol (de)-coding exist as well. For example, 'punycode' (Costello, 2003) is an encoding system that translates strings made up of symbols from the Unicode alphabet (see subsequent text) to a possibly longer sequence of ASCII symbols. A second example is the 'BOCU-1' encoding standard (Scherer and Davis, 2006) that compresses strings from the same alphabet while keeping, among other things, the binary sorting order unaltered.

3.2.2 Unicode

Historically, hundreds of encodings have been developed and many of them are widely in use. One practical problem is that different encoding schemes can assign different symbols to the same byte sequence. Or, a byte sequence valid in one encoding is invalid in another. The development of all these conflicting encodings has led to the current situation where, in general, one cannot determine by looking at the byte sequence alone, which encoding is used with certainty. This is why, for example, the XML standard requires the encoding of XML documents to be declared explicitly if it is not in one of the encodings predefined in the XML standard (W3C, 2008).

In an attempt to resolve this situation, the Unicode Consortium developed a set of standards that aim to facilitate the exchange of textual data between systems using different encodings. Two of these standards are of particular interest: first of all, the Unicode Standard aims to assign a number to every possible symbol, used in any language, ever. The unicode standard can be seen as a large table where each row contains a description of a symbol and a number. The column with descriptions can be interpreted as the Unicode alphabet: it is the list of abstract symbols contained in the standard. The numbers are called *code points* in Unicode terminology. The standard is still updated frequently. For example, in version 7.0 of the standard (Allen *et al.*, 2014), the alphabet has been extended with more than 2500 symbols, including the set of 'Sinhala Archaic Numbers'

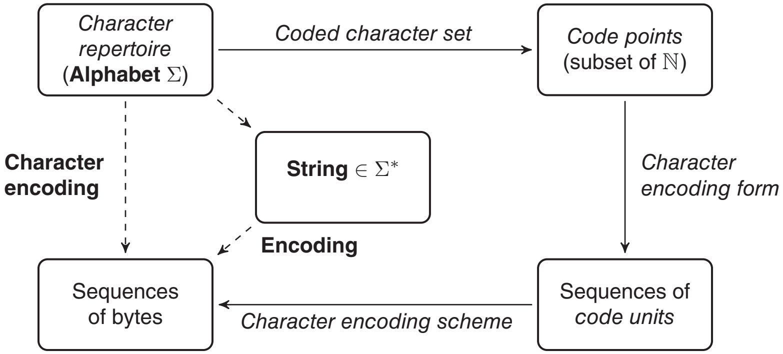


Figure 3.1 Terminology of the Unicode Character Encoding Model (UCEM, Whistler *et al.* (2008)) and terms used in this book. The fine-grained mappings defined in the Unicode standard are typeset in italics. Terminology used in this book is printed in bold. In Unicode, a character encoding system is defined in three steps, where the first step (definition of character repertoire connected to code points) is fixed in the standard. The ensuing translation steps may be defined by programmers conforming to the standard or one can use one of the mappings developed by the Unicode consortium, such as UTF-8.

and an extensive set of emojis. In total, the Unicode standard currently has room to define 1 114 112 code points of which 252 603 are defined in version 7.0

Secondly, the Unicode Character Encoding Model Whistler *et al.* (2008) offers methods and standard terminology in which an encoding can be described. An overview is shown in Figure 3.1 along with the terminology used in this book. The first step in this model is to assign to each symbol in an alphabet a number (in Unicode terms, this is called a *character repertoire*). Since the Unicode standard aims to provide such a mapping for each symbol ever used, an encoding will commonly start with choosing a subset of the mappings provided by unicode. Such a subset is called a *Coded character set*. In the next step, a *character encoding form* maps each code point to an integer or sequence of integers with a specified binary width; this is called a *sequence of code units*. Sequences of code units can be explicitly stored in computer memory and are already a form of encoding. In the final step, a *character encoding scheme* defines how sequences of code units are represented as sequences of bytes, allowing, for example, for compression to take place.

Since the terminology used in the Unicode standard is fairly detailed, we will continue to use the terms encoding or character encoding as a shorthand.

3.2.3 Some Popular Encodings

According to one survey (w3tech, 2014), at the time of writing, about 82% of all websites are encoded in UTF-8, with Latin-1 (ISO-8859-1) as runner-up at about 10%. ASCII is used as the basis for many programming languages and can, in several respects, be regarded as the progenitor of encodings. Although the most common operating systems (Linux-based, Mac OSX, Windows) of interest to us support the Unicode standard, they do not all use the same encoding scheme. Microsoft Windows used to use the Windows Code Pages system but it has switched to UTF-16 for the internal representation of text since Windows 2000. Linux-based systems commonly use UTF-8 for internal text representation (and scripts, for example) and the Cocoa development environments for Mac OSX use UTF-16 as a basis.

It is noteworthy that for many encodings, there are excellent and detailed resources online. Wikipedia, for instance, offers detailed overviews of many encoding standards. For encodings from the Unicode standard, the ultimate resource is freely available from `unicode.org`. The website `fileformat.info` offers an excellent searchable overview of all symbols in the unicode standard, including their representation in various encodings. Here, we give a short overview of some commonly used character encodings and point out some commonly encountered technical issues.

American Standard Code for Information Interchange (ASCII)
ASCII encodes an alphabet consisting of 128 symbols using 7 bits for each symbol. When stored as a byte, an ASCII symbol always has the first (most significant) bit set to 0. The 128 symbols are subdivided in 32 control characters such as 'shift', 'tab', and 'bell' and 96 printable characters such as 'a–z', '0–9', and 'space'. The ASCII alphabet is based on the alphabet used in the English language and therefore contains no accented characters.

Universal Character Set Transformation Format—8 Bit (UTF-8)
This character encoding, defined by the Unicode Consortium, is capable of representing every symbol in the Unicode alphabet. Each symbol is stored as a sequence of one, two, three, or maximally four bytes. UTF-8 extends the ASCII character set in the sense that the first 127 symbols of UTF-8 are equal to the ASCII alphabet. There are two technicalities to keep in mind when working with UTF-8, UTF-16, or UTF-32 encoded data sets: the byte order mark (BOM) and *unicode equivalence*.

In UTF-8, the BOM is a sequence of three bytes at the beginning of a file that has no other purpose but signal to the interpreter that the text is encoded in UTF-8 (for other purposes, see UTF-16). Inclusion of a BOM in a file is optional; but since it may be present, unicode-aware programs must take it into account.

Unicode equivalence is the term used to indicate that some characters may be represented as multiple code points. This is the case for accented characters which may either be represented as a single code unit or as a code unit for the character followed by one or more modifying code units that add accents to the preceding character. For example, the Latin small letter 'o' with diaereses (umlaut) can be represented as the two-byte UTF-8 code unit `0xc3 0xb6` (ö), or by two consecutive code units given by (`0xcf`, `0xcc 0x88`) where `0xcf` represents the 'o' and `0xcc 0x88` adds the diaereses. For many text processing applications, it is desirable to transform a text so the same representation is chosen throughout. This process is called *unicode normalization* and is discussed in Chapter 5.

Universal Character Set Transformation Format—16 bit (UTF-16)
Like UTF-8, this character encoding is defined in the Unicode Standard and is capable of representing every symbol in the unicode alphabet. A UTF-16 code unit consists of either two bytes (for the first $2^{16} = 65\ 536$ code points) or pairs of two-byte units (for code points above 2^{16}). The idea is that the most used code points are represented in the range $[0, 65\ 536]$.

The storage format of the first 2^{16} code points in UTF-16 are just the integers $0, 1, \ldots, 2^{16}$ in binary format. It is a legacy of early hardware design that some computer systems store bytes with the least significant bit last (this is called little-endian) and other systems with the most significant bit last (this is called big-endian). Therefore,

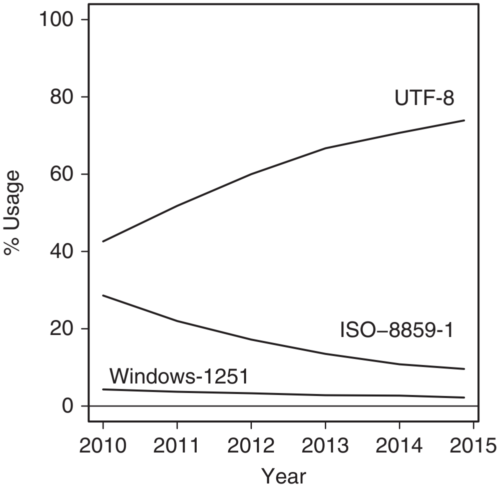


Figure 3.2 Percentages of websites using encodings as a function of time. Shown here are three encodings most popular at the time of writing. *Source*: Courtesy of w3tech (2014).

a sequence of UTF-16-encoded symbols may be preceded with a BOM, allowing a program to find out in what endianness the sequence was stored. Like in UTF-8, the issue of unicode equivalence may be resolved using unicode normalization tools.

Universal Character Set Transformation Format—32 Bit (UTF-32)

This character encoding is arguably the simplest of the encodings defined by the unicode standard: each symbol from the unicode alphabet is represented by a 32-bit integer. Unlike the UTF-16 and UTF-8 encodings, in UTF-32, each code point is represented by a single code unit. Like UTF-16, the BOM indicates the endianness of the storage format; and like UTF-16 and UTF-8, unicode equivalence is something to take into account.

Latin-1 (ISO-8859-1) and Windows Code Page 1252

Latin-1 is an 8-bit character encoding. It includes ASCII and uses the room left by that encoding (one bit) to represent 127 extra symbols. The extra symbols are intended mostly to represent a number of European languages. The 256 symbols of Latin-1 coincide with the first 256 code points of the Unicode alphabet. Latin-1 is the first in a series of ASCII-extensions defined in ISO/IEC standards 8859-1 to 8859-10 (Figure 3.2). All of them include the ASCII set as a basis but add various extensions, including, for example, Cyrillic (8859-5), Greek (8859-7), and Hebrew (8859-8). Latin-1 was frequently for websites until it was surpassed by the use of UTF-8 around 2008.

Windows Code Page 1252 (often abbreviated to CP1252) is a character encoding that, in principle, became obsolete when the Windows operating system internally switched to UTF-16. However, it is still widely used, for example, in office applications and it is also the default encoding used by R on Windows. Windows Code Page 1252 is identical to Latin-1 except for a range of 31 characters. The range concerns little-used control characters which are replaced with symbols representing characters in CP1252.

Exercises for Section 3.2

Exercise 3.2.9 *Given an English text with a length of n symbols. How many bytes are necessary to store it in UTF-8, UTF-16, or UTF-32? You may neglect the contribution of the BOM.*

3.2.4 Textual Data in R: Objects of Class Character

In R, text is stored in vectors of class `character`. A character variable can be defined on the command line by enclosing a string with single or double quotes. When double quotes are used, single quotes can be enclosed in the string; when single quotes enclose the string, double quotes can be included.

```
"Just say 'hello'"
## [1] "Just say 'hello'"
'Just say "hello"'
## [1] "Just say \"hello\""
```

Character vectors are always printed with double quotes enclosing them and internal double quotes are preceded with a backslash.

The backslash is called an *escape character* and it signals that a special string constant or command is coming after it. In the previous example, it signals that the next character is the *string constant* `\"`, which stands for a text double quote that is not to be interpreted as the end of the string. So we could have defined the second example as follows.

```
"Just say \"hello\""
## [1] "Just say \"hello\""
```

R accepts a number of predefined string constants, which are summarized in Table 3.2. Three of them (`\'`,`\"`, and `\\`) are simply shorthands for characters that otherwise have a special meaning. The other constants signal a command. For example, the `\n` signals a new line. The most interesting commands are those that allow users to define a character by their representation as a byte sequence (`\`, `\x`) or by referring to their unicode code point (`\u`, `\U`).

Recall that a string is represented as a sequence of bytes whose value depends on the used encoding. With `\` or `\x` one can define single-byte characters, referring directly to their number in octal or hexadecimal notation. As a simple example, consider the capital letter 'A'. It is represented as code point 65 (in either UTF-8 or Latin-1), which equals 101_8 in octal and `0x41` in hexadecimal notation. Therefore, the following three character definitions are equivalent:

Table 3.2 Text constants in R.

Constant	Description	Constant	Description
`\'`	Single quote	`\t`	Tab character
`\"`	Double quote	`\b`	Backspace
`\\`	The backslash	`\f`	Form feed
`\n`	New line	`\v`	Vertical tab
`\r`	Carriage return	`\a`	Bell
`\`*ooo*	Character in octal code in current encoding (max three)		
`\x`*hh*	Character in hexadecimal code in current encoding (max two)		
`\u`*hhhh*	Unicode code point in hexadecimal code (max four)[a)]		
`\U`*hhhhhhhh*	Unicode code point in hexadecimal code (max eight)[a)]		

Allowed octal codes are of the form *a*, 0 … , 7, hexadecimal numbers are of the form 0, … , 9, `A`, … , `F` or `a`, … , `f`.

a) Only in locales where multi-byte encoding is supported.

```
c("A", "\101", "\x41")
## [1] "A" "A" "A"
```

The main drawback of the commands '\' and '\x' is that for characters beyond the ASCII range, their interpretation is undefined. This limitation is overcome by the unicode commands \u and \U. These allow one to directly refer to the unicode code point in hexadecimal notation.

```
"Latin small letter 'o' with diaereses: \uf6"
## [1] "Latin small letter 'o' with diaereses: ö"
```

Here, we use that the unicode code point of 'ö' is 246, or 0xf6 hexadecimal.

Under the hood, each element of a character vector is just a sequence of bytes (more precisely, C's char type) with a maximum length of $2^{31}-1$ bytes. Since for non-ASCII symbols, the number of bytes used to represent a symbol depends on the encoding scheme, the actual memory usage for storing a string varies accordingly.

Many of R's built-in functions dealing with strings are capable of interpreting it as a sequence of symbols or as a sequence of bytes. For example, nchar by default counts the number of symbols, but may be told to count the number of bytes.

```
nchar("Motörhead")
## [1] 9
nchar("Motörhead", type="bytes")
## [1] 10
```

The symbols in an alphabet do not always occupy space when printed. For example, the character for zero-width non-braking space (0xfeff) is invisible. Therefore, the option type="width" computes the number of columns used when printing a string to the screen (using cat) in a fixed-width font.

```
nchar("FO\ufeffO",type="width")
## [1] 3
```

The amount of memory used for storage of character data is minimized in R. Invisible to the user, each unique string is stored once in a central pool of memory shared by all string-carrying objects within the R session. When a new string is created, R checks if it was defined before; and if so, the previously stored version will be used. Copies are only made on an as-needed basis, for example, when a string is altered.

3.2.5 Encoding in R

By default R assumes that strings are stored in the encoding defined by current locale settings. The local encoding settings that R finds on your system can be found with Sys.getlocale. The name of the setting to look for is LC_CTYPE. Typical output on a Mac OSX or Linux machine would be something like

```
Sys.getlocale("LC_CTYPE")
## [1] "en_US.UTF-8"
```

while on a Windows machine one gets something looking like

```
Sys.getlocale("LC_CTYPE")
## [1] "Dutch_Netherlands.1252"
```

Notice the structure of the output; it should be read as follows:

```
[language]_[variant].[encoding]
```

When text data is entered from the command line, read from file, or retrieved from some other connection, R will assume that data is offered in the encoding specified in the locale settings unless explicitly specified otherwise. If input is not in the locale encoding, one may specify the encoding of the data at read-time and R will label the data accordingly.

Before discussing how to specify encoding at read-time, it is good to have a look at how R recognizes encoding of strings it has already stored. Since it is not generally possible to detect with certainty from a byte sequence alone what encoding was used to store it, it is possible to mark each element of a character vector with an encoding label so it can be handled accordingly. R supports the following labels:

- `"UTF-8"` for UTF-8 encoded strings;
- `"latin1"` for ISO-8859-1 encoded strings;
- `"bytes"` for strings that should be treated as byte-sequences; and
- `""` or unknown encoding. In this case, it is assumed that a string is stored in the encoding specified by the locale settings of the system running R.

Strings for which all bytes have a value ≤ 127 are assumed to be ASCII and will never have an encoding label, since ASCII is the same across the three supported encoding labels. The reason that R supports this limited set of encoding labels is that `latin1` and `UTF-8` are the only encodings that are sure to be supported across all systems on which R is supported. That is to say, the actual overlap is probably larger; but one problem is that the same encoding may have different names across systems. This problem is addressed further in Chapter 5 when we discuss encoding conversion and transliteration.

Different elements of a character vector may have different encoding labels. The encoding labels can be retrieved with the function `Encoding`.

```
Encoding(c("Motörhead","On Parole"))
## [1] "UTF-8" "unknown"
```

Here, the first element contains a non-ASCII character and is therefore stored and labeled as UTF-8, as defined by the locale in the environment where R was run. The second element only contains ASCII characters and therefore receives no special encoding label. Users may set an encoding label as well, but this should be done with caution as a misspecification will cause erroneous interpretation.

```
s <- "Motörhead"
Encoding(s) <- "latin1"
s
## [1] "MotÃ¶rhead"
```

Given that strings can be labeled with different encodings, one needs to decide what to do when two strings are concatenated. In R, concatenation can be done with `paste`, which uses the following rules:

- If any of the concatenated strings is marked as `"bytes"`, the result is marked as `"bytes"`.

- If none of the concatenated strings are marked as "bytes", but any one of them is marked as UTF-8, all strings are converted to UTF-8 as necessary and the result is marked with UTF-8.
- Otherwise the result is in the encoding declared in the current locale. A mark is added if it is one of UTF-8 and latin1 and one of the inputs is marked (i.e., if all input is ASCII, all output is unmarked ASCII as well).

The abovementioned logic goes to show that working with text data in various encodings is rather complicated. It is therefore highly recommended to make sure, when you combine text data from different sources, that in-memory they are stored in the same encoding. This and other kinds of string normalization processes are discussed further in Chapter 5.

3.2.6 Reading and Writing of Data with Non-Local Encoding

For reading tabular data from text files, the high-level function read.table and its cousins (read.csv and so on) accept two encoding-related arguments. The argument encoding is used to add an encoding label to character vectors after the file is read. This can be used when the file encoding differs from the local encoding, but is one of latin1 or UTF-8, since these are supported on every R system (recall that ASCII is contained in both as the first 127 characters). For other encodings, the argument fileEncoding is used to indicate the encoding in which the file to be read is stored. R will attempt to translate the file to the local encoding while reading in that case. For example, to read a file encoded in CP1254 on an OS whose locale is not CP1254, one uses the following command.

```
read.table([filename], fileEncoding="CP1254")
```

A recipe for reading non-local encodings that works for most systems looks as follows.

- For any system, if the file is in ASCII, no encoding needs to be specified.
- For Linux/Mac, if the file is in UTF-8, no encoding needs to be specified. If the file is in latin1, use encoding="latin1"; for all other encodings use fileEncoding="[encoding]".
- For Windows, if the file is in latin1 or UTF-8, use encoding="latin1", respectively encoding="UTF=8". For all other encodings, use fileEncoding="[encoding]".

For OSs not running on a UTF-8 locale (e.g., Windows), this may not always work.
A list of accepted values for "fileEncoding" can be requested by typing

```
iconvlist()
```

on the command line. The number, naming, and type of encodings available to R depend on the system running it. The only encodings that are guaranteed to work on all systems supported by R are latin1, UTF-8, and "" (unknown, otherwise known as native encoding). In practice, most systems support at least a few hundred encodings, so the actual overlap is likely to be larger. The reason for this platform dependency is that R depends on a third-party library (iconv) of which different implementations exist for different platforms.[1] So if you need to specify input encoding, and your code

[1] iconv is part of the Single Unix Specification as designed by The Open Group (2013).

needs to run on different platforms, it is necessary to check whether the input encoding is supported on the platforms you are aiming at.

To read general text files, one can use the `readLines` function. This function reads (part of) a file line by line, converting the input to R's internal encoding scheme. Since `readLines` does not accept a `fileEncoding` argument, specification of input encoding must be done by first opening a connection to a file while specifying the input encoding, call `readLines` to read and convert from the connection and finally close the connection.

```
# Open a read-only connection to a file named [filename],
# stored in CP1254 ecoding.
con <- file([filename], open="r", encoding="CP1254")
# Read, converting from CP1254 to internal encoding
dat <- readLines(con)
# Close the connection
close(con)
```

Optionally, the `readLines` function accepts an `encoding` argument that allows one to supply the output with an encoding label. This procedure generalizes to several types of connections. For example, the `url` function opens a connection to a web page and `gzfile` opens a connection that allows for direct reading from files that have been compressed using `gzip`. An overview of connections offered in base R is given in Table 3.3.

The procedures for writing text in a specified encoding are similar to the file reading process. The functions `write.table` and its cousins have a `fileEncoding` argument allowing one to specify any output encoding listed by `iconvlist`. Writing text to a (file) connection with non-local encoding is illustrated with the following example.

```
# If necessary, create a new file [filename]
file.create([filename])
# Open a writable connection that converts text to CP1254
con <- file([filename], open="w", encoding="CP1254")
# Write text to the file
write("hällo world", file=con)
# Close the connection
close(con)
```

Again, this procedure using connections generalizes to any connection that can be opened for writing.

Table 3.3 Functions opening connections in base R.

Text/binary	Compressed	Other
`file`	`gzfile`	`pipe`
`url`	`bzfile`	`fifo`
	`xzfile`	`socketConnection`
	`unz`	

All of them have an `encoding` argument to set the input encoding.

3.2.7 Detecting Encoding

Ideally, a data source comes with a description of its encoding. In practice however, such descriptions (like in the first lines of an XML file) may either be wrong or missing. In such a case, one can try to guess the encoding by analyzing the byte sequences. There are no facilities for encoding detection in base R, but the `stringi` package of Gagolewski and Tartanus (2014) exports such functionality, based on an external library (ICU, 2014).

To detect the encoding of a file, one can read in the first, say, several hundred bytes, and then use `stri_enc_detect` to guess the encoding. Given here is an example. We created a test file containing the following text.[2]

> Considérant que la méconnaissance et le mépris des droits de l'homme ont conduit à des actes de barbarie qui révoltent la conscience de l'humanité et que l'avènement d'un monde où les êtres humains seront libres de parler et de croire, libérés de la terreur et de la misère, a été proclamé comme la plus haute aspiration de l'homme.

To detect the encoding, we read in the first 200 bytes using `readBin` and then estimate the encoding.

```
library(stringi)
bytes <- readBin("hr.txt", what="raw",n=200)
stri_enc_detect(bytes)
## [[1]]
## [[1]]$Encoding
##  [1] "UTF-8"       "ISO-8859-1" "ISO-8859-2" "ISO-8859-9" "UTF-16BE"
##  [6] "UTF-16LE"    "GB18030"    "Big5"       "IBM424_rtl" "IBM420_ltr"
##
## [[1]]$Language
##  [1] ""   "fr" "ro" "tr" ""   ""   "zh" "zh" "he" "ar"
##
## [[1]]$Confidence
##  [1] 1.00 0.98 0.42 0.20 0.10 0.10 0.10 0.10 0.01 0.01
```

The encoding detector returns a list of possible encodings and languages, together with a confidence level for each encoding. The confidence level ranges from 0 to 1 and depends on the number of bytes used to detect encoding and the number of non-ASCII characters encountered. In this case, UTF-8 encoding was detected with confidence 1 and ISO-8859-1 with French language with confidence 0.98. Encoding is detected by matching byte sequences against a number of known encoding standards and statistical checks on frequently used characters within an encoding. The `stri_enc_detect` function can detect 30 encodings used for text, including the UTF family, ISO-8859-1 – 9, Windows CP1250–6, and several encodings for Chinese, Japanese, Korean, Cyrillic, and Hebrew alphabets. An up-to-date list is given in the documentation of `stri_enc_detect`. It depends on the guessed encoding what languages can be detected. For the UTF family of encodings, no language detection is performed.

[2] From the Universal Declaration of Human Rights.

Besides the encoding detection function, stringi exports a number of functions that may be used to check if a string (or raw vector) contains data of a certain encoding. These include `stri_enc_isascii`, `stri_enc_isutf8`, and several others.

3.2.8 Collation and Sorting

Sorting is an important subtask in many data processing applications including merging, tabulation, or clustering. Sorting of strings depends on the order or collation defined on the alphabet from which the string is derived. Here, we first define ordering on a set of strings in an abstract sense before discussing how collation and sorting is applied in practice.

Collation and Lexicographical Ordering

A *collation*, denoted $\leq$, is a total order on an alphabet. This means that for all a and b in Σ the following properties hold.

- Totality. Either $a \leq b$ or $b \leq a$, or both.
- Transitivity. $a \leq b$ and $b \leq c$ implies $a \leq c$.
- Antisymmetry. $a \leq b$ and $b \leq a$ implies $a = b$.

Defining a collation on an alphabet allows one to uniquely order the symbols with a sorting algorithm of choice. As it turns out, defining a collation is enough to make the set of strings Σ^* sortable as well, by applying the same algorithm you would use to order, for example, a sequence of names: lexicographical ordering.

Given two strings s and t. We denote the length of the strings (the number of abstract characters of which they are composed) with $|s|$ and $|t|$. The empty string, denoted as ϵ, is the unique string with length zero. The empty string is also the identity under concatenation: $s\epsilon = \epsilon s = s$ for all $s \in \Sigma^*$. The *lexicographical ordering* on Σ^* is now defined as follows.

$$\epsilon \leq u \text{ for all } u \in \Sigma^*;$$

$$s \leq t \text{ if } s_1 \leq t_1 \text{or } s_1 = t_1 \text{and } s_{2\cdots|s|} \leq t_{2\cdots|t|},$$

where the subscripts indicate substrings.

Practical Issues: Collation Charts

Collation varies strongly across alphabets and languages and is therefore locale dependent. Collation tables are detailed and contain choices that must, to an extent, be arbitrary. Computer alphabets typically consist of capital and small letters, numbers, punctuation symbols, combining symbols (accent-adders) and even invisible symbols like command characters or the zero-width binding space. In some alphabets, accented characters are added to the alphabet. For example, the Finnish alphabet in collation order is given by

A B C D E F G H I J K L M N O P Q R S T U V X Y Z Å Ä Ö

so it has no W and Å, Ä and Ö sort after Z. In German, however, the Ä sorts right after A. Some languages have sorting rules depending on the combination of two common characters. In Czech, for example, the combination CH has its own pronunciation and sorts after H and before I. A large number of collation charts can be found on `collation-charts.org` (Barkov, 2014).

In R, collation and thus the behavior of `sort` and `order` depend on the current locale setting. The locale setting for the current locale can be found with the following command.

```
Sys.getlocale("LC_COLLATE")
## [1] "en_US.UTF-8"
```

If a `character` vector containing strings in different encodings is passed to `sort` or `order`, all contents are first translated to UTF-8. Just like for the default encoding and translation thereof, R depends on external resources for collation, which are likely to differ between platforms. The `stringi` package offers collation control that is platform independent. Here is an example of how strings containing numbers can be sorted as if they were actual numbers.

```
library(stringi)
# default sorting: 1 < 2
stri_sort(c("19","2"))
## [1] "19" "2"
# numeric sorting: 2 < 19
stri_sort(c("19","2"), opts_collator = stri_opts_collator(numeric=TRUE))
## [1] "2"  "19"
```

Using `stri_opts_collator(locale=[locale])` the collation order according to a certain language and region combination can be set. The list of available locales that come with the ICU library can be requested with `stri_locale_list()`.

Exercises for Section 3.2.8

Exercise 3.2.10 *Given the alphabet* $\Sigma = \{0, 1\}$ *with the collation* $0 \leq 1$*. Apply the definition of lexicographical ordering to order the following sets of strings:* $\{0011, 0101, 1100\}$, $\{0101, 0011, 010110\}$.

3.3 Times and Dates

There are two types of pitfalls that data analysts may encounter when processing time-like data. The first has to do with regional differences in notation of dates, such as the North American month-day-year notation versus the European day-month-year notation as well as translation issues for the names of weekdays and months. Secondly, there is the difference between the actual physical passing of time and the time denoted on our calendars and clocks. These differences are caused by leap days, time zones, daylight saving times, and occasional leap seconds. Here, we discuss how the passing of time is measured and how it gets translated to time the clocks and calendars in our software.

3.3.1 AIT, UTC, and POSIX Seconds Since the Epcoch

The SI unit of time is the second (s). The quantity of time that is associated with 1 s is defined as [Terrien (1968), but see BIPS (2006)].

> the duration of 9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom (at 0K, and at rest).

In practice, the amount of time passing is internationally agreed upon by employing a set of atomic clocks of which the values are combined to a weighted average called the International Atomic Time (TAI)[3] , t_{TAI}. For the purpose of computation, we interpret TAI as a measure for the actual, universal physical passing of time. It is accurate up to about one part in 10^{16}.

Clocks that are synchronized to TAI do not synchronize directly to the atomic clocks, but to a shifted variant called the Coordinated Universal Time (UTC)[4] , t_{UTC}. The difference between UTC and TAI has a historical reason. Before the modern definition, a second was defined as 1/864 000th of the mean solar time, measured at the prime meridian at Greenwich (United Kingdom). The mean solar time is computed as a year's average of the apparent length of the day as measured with a sun dial. As measurements became more accurate, it became obvious that the rotation of the earth around its axis varies slightly over the years. For this reason, a leap second is introduced at irregular intervals, such that the number of seconds in the mean solar time does not differ more than 0.9 from 864 000 s. Unlike leap days, leap seconds may be positive or negative, so the t_{UTC} is related to t_{TAI} as

$$t_{\mathrm{UTC}} = t_{\mathrm{TAI}} - n^{-}_{\mathrm{leap}} + n^{+}_{\mathrm{leap}}, \tag{3.16}$$

where $n^{\pm}_{\mathrm{leap}}$ are the number of positive (negative) leap seconds. At the time of writing (November, 2014), $t_{\mathrm{UTC}} = t_{\mathrm{TAI}} - 35$. Leap seconds are announced a few months in advance in Bulletin C of the International Earth Rotation and Reference Systems Service (IERS, 2014). Worldwide many clocks, including Internet time, are synchronized to UTC; and in most countries, the official time is offset by an integer number of hours from UTC. The actual offset may depend on local arrangements concerning daylight saving times.

Operating systems typically have a built-in service to report the POSIX elapsed seconds since the epoch, or POSIX time (t_p) in short (The Open Group, 2013). POSIX time is defined with respect to UTC, so it can be computed from time synchronized through a network. Denoting UTC as in the form (y, d, h, m, s), for year, day of the year, hour, minute, second, the POSIX time t_p is defined as

$$\begin{aligned} t_p = {} & s + 60m + 3600h + 86400d \\ & + 31536000(y-70) + 86400((y-69)\backslash 4) \\ & - 86400((y-1)\backslash 100) + 86400((y+299)\backslash 400). \end{aligned} \tag{3.17}$$

Here, we use integer division (division without remainder), so $a\backslash b = \lfloor a/b \rfloor$. In the first line, the number of seconds elapsed in the current year is computed. Recall that leap days occur every year that is divisible by 4 except at centuries, which must be devisable by 400. The second line in the equation offsets the base to 1900 and then corrects for leap days occurring since 1973. The last line subtracts a day for each century and adds one again for each century that is divisible by 400.

[3] After the French *temps atomique international*.

[4] A compromise between the English CUT and the French *Temps Universel Coordonné* (TUC).

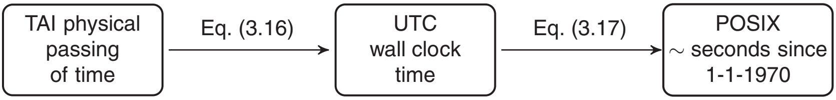


Figure 3.3 Relation between TAI, UTC, and POSIX time.

POSIX time should be interpreted as an approximation to the number of physical seconds passed since 1 January 1970, 00 : 00 : 00 h. It is an approximation because where UTC takes account of leap seconds when deriving clock time; POSIX time does not take leap seconds into account when going back from UTC to 'seconds since 1970'. The difference between the actual TAI number of seconds since 1970 and the number of POSIX seconds since 1970 depends on the number and sign of leap seconds that were introduced since 1970. At the time of writing, 25 positive leap seconds have been applied since 1970. The setback from this convention is that there will be inaccuracies when determining time differences over periods containing one or more leap seconds. This will only rarely be significant, however, since leap seconds are introduced about every one-and-a-half years. For most applications, accuracy to the second is not relevant. The main advantage of the abovementioned definition is that it ensures that each day has precisely $60 \times 60 \times 24$ s.

Figure 3.3 summarizes the relations between the three time standards: TAI represents the universal passing of time, leap second correction yields the time on our calendars and clocks (UTC). POSIX time approximates the number of seconds since 1-1-1970 (00 : 00 : 00), from UTC, disregarding leap seconds.

3.3.2 Time and Date Notation

Time and date notation varies from region to region. For example, while English speakers often denote a point in time as 'December 17 2015, 11:30 AM', others may use '17 December 2015, 11:30'. And while some people would call Sunday the first day of the week, others call it the seventh day. The ISO 8601 standard ISO (2004) offers standardized and unambiguous ways to denote calendar dates and time of day. It applies to dates expressed on the Gregorian calendar and times on the 24-h timescale.

Under the ISO 8601 standard, a complete date is written as a sequence consisting of year, month, day of month; a sequence of year and day of year; or year, week of year, day of week. Each element in a full date has a prescribed number of digits (e.g., week 6 would be written as 06) and each sequence can be represented in two ways: a basic format consisting of only digits and an extended format where date elements are separated by a hyphen. Optionally, a date may be preceded with a plus or minus sign to indicate dates prior to the year 0000. Table 3.4 gives an overview of the options with an example. When dates are represented as a sequence of year, week of the year, and day of the week, the first day of the week is Monday.

To represent dates with a reduced accuracy, one simply takes off the undesired part at the end of the notation. The only exception is the `YYYYMMDD` notation. If the `DD` part is taken off, a hyphen must be inserted between `YYYY` and `MM`. The standard does not make recommendations for representing dates without including the year.

Since a week can overlap 2 years, we need to agree upon when a week counts for the current year and when for the next. The standard prescribes that the first week of the

Table 3.4 ISO 8601 notational formats for calendar dates on the Gregorian calendar.

Style	Notation	Example
Basic	(±)YYYYMMDD	19760928
Extended[a)]	(±)YYYY-MM-DD	1976-09-28
Basic	(±)YYYYDDD	1976272
Extended	(±)YYYY-DDD	1976-272
Basic	(±)YYYYWwwD	1976W292
Extended	(±)YYYYWwwD	1976-W29-2

Bracketed expressions are optional. Days of the week are numbered with Monday as day 1.

a) Without the sign, this is the default format recognized by `as.POSIXct` in R. See Section 3.3.3.

Table 3.5 ISO 8601 notational formats for time of day in the 24- h system.

Style	Notation	Example
Basic	hhmmss(,ss)(Z)(±hh(:)mm)	062455
Extended	hh:mm:ss(,ss)(Z)(±hh(:)mm)	06:24:55

Bracketed expressions are optional.

year can have maximally 3 days in the previous year (Table 3.5). So if New Year's day is on a Friday, Saturday, or Sunday, the week is counted as belonging to the previous year; if New Year's day falls on a Monday, Tuesday, or Wednesday, the week is counted in the new year. Weeks are numbered from 01 to (maximally) 53.

Time of day is represented in hours, minutes, and seconds. Seconds may optionally be specified to an arbitrary number of digits after the decimal separator; but if a decimal separator is present, at least one digit should be put behind it. The most common notation is probably `hh:mm:ss`. If the time represents a UTC time, this can be indicated by concatenating a capital `Z`. The offset from UTC is denoted by appending `±hh:mm`, in which case the `Z` is unnecessary. In all cases, the colon is optional as well.

Date and time of day notation may be combined in the following way:

```
[date notation]T[time notation]
```

In practice, the time separator `T` is often replaced with a space. In fact, a separator may be left out if both sender and receiver agree upon it and there is no chance of confusion.

Exercises for Section 3.3.2

Exercise 3.3.11 *Determine which of the following notations conform or disconform to the ISO 8601 standard.*

a) `03-03-2012`

b) `03-25-1845`

c) `16322516`

d) `10-11-99`

e) `02-01-1977`

e) `12:51:33,`

f) `12:51:01,425`

g) `24:00Z`

h) `22:00+1:00`

i) `15:45:00-08:00`

For the ones that do not conform to the standard, propose a better notation when possible.

3.3.3 Time and Date Storage in R

There are two object classes in R that store a point in time. The first, and most important, is `POSIXct`, where `ct` means 'calendar time' (Hornik, 2014). It stores the number of seconds elapsed since 01-01-1970 00:00:00 in the UTC time zone. The underlying storage format is an double precision number (an R `numeric`), so it can store fractional seconds as well.

```
t0 <- Sys.time()
t0
## [1] "2017-06-30 16:34:35 CEST"
# the storage format is unveiled with 'unclass'
unclass(t0)
## [1] 1498833276
```

Given the number and signs of leap seconds elapsed, the time zone, and the status of daylight saving time (DST), the wall clock time at any place on earth can be derived from this object. When printed to the screen, as in the given example, R uses information from locale settings to determine to what time zone and DST setting it should be translated. In this conversion, leap seconds are not taken into account.

The second class storing times in R is called `POSIXlt`, where `lt` stands for 'local time' (Ripley and Hornik, 2001). It contains a list of integer vectors representing date and time in human-readable format.

```
t0_lt <- as.POSIXlt(t0)
names(unclass(t0_lt))
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"
##  [8] "yday"   "isdst"  "zone"   "gmtoff"
```

Most of these names speak for themselves. However, there are a few things to take into account. First of all, the year is represented as the number of years after 1900.

```
t0_lt$year
## [1] 117
```

Furthermore, `isdst` indicates whether DST is in effect. The `zone` (a character indicating time zone) and `gmtoff` (offset from Greenwich Mean Time) are only filled on a subset of the systems that are supported by R and we do not recommend to rely upon them. Since `POSIXct` is a simple format, which can be stored as a column in a `data.frame`, it is recommended to do all calculations and storage in the form of `POSIXct` and use `POSIXlt` perhaps only as a convenience to obtain numeric values for years, months, days and so on.

Finally, users who need an accurate conversion from POSIXct to TAI will need the date and time where leap seconds were introduced. These are stored in a global variable called `.leap.seconds`.

3.3.4 Time and Date Conversion in R

Since dates and times, but especially dates, are denoted in many different formats, extracting time data from character data can be a chore. Fortunately there are tools available that facilitate extraction from, or conversion to, text format, also in the case when times and dates are not denoted in ISO 8601–compliant format.

In particular, interpretation of (abbreviated) names of weekdays and months depend on language and is hence governed by locale settings. The localization used to interpret such names can be found as follows:

```
Sys.getlocale("LC_TIME")
```

Here, we first point out the most convenient way to extract date information from character vectors through the `lubridate` package. Next, we discuss the fine-grained control offered by base R functionality.

The `Lubridate` Package

The `lubridate` package of Grolemund and Wickham (2011) allows one to extract dates from character data, transforming them to `POSIXct` format. Here is an example.

```
text <- c("2/1/1977", "02-Jan-1977", "It all started on 2
January 1977")
lubridate::dmy(text, locale="en_US.utf8")
## [1] "1977-01-02" "1977-01-02" "1977-01-02"
```

The function `dmy` attempts to extract a single date from each element of a character vector, assuming that dates are notated in day-month-year order. By default, `dmy` assumes that names of months are to be interpreted in the language defined in the current locale (`LC_TIME`). Optionally, one may pass the locale argument to force interpretation in other locale settings, as in the example. The `lubridate` package offers six equivalent functions to convert text to `POSIXct`, each corresponding to a permutation of day, month, and year in the character string. The names of these functions are listed here:

```
dmy   myd   ydm
mdy   dym   ymd
```

Once points in time are available in POSIXct format, the package offers a number of utilities to extract data from it. For example, the functions `wday`, `mday`, and `yday` extract the day of week, month, or year. The function `isoweek` returns the week number as defined in the ISO 8601 standard, while `week` counts the number of fully elapsed weeks since the beginning of the year. Comparable functions exist to extract months, years, hours, minutes, and seconds. The package offers a wealth of date/time manipulation functions which is well worth looking into.

Fine-Grained Control with Base R Functionality

To extract date-time data from a text string, one needs to specify precisely where in the string and in what format the data can be found. The workhorse function from base R for this task is strptime and it accepts a character vector and a format specification. The strptime function returns an object of class POSIXlt rather than POSIXct and it is a good idea to immediately convert.

```
text <- "It happened on 02-01/1977, at 05:30 in the morning"
(t <- as.POSIXct(strptime(text,
  "It happened on %d-%m/%Y, at %H:%M in the morning")))
## [1] "1977-01-02 05:30:00 CET"
```

Contrary to the functions in the lubridate package, one needs to specify precisely where in the string, what part of the data occurs using *conversion codes*. Also, strptime allows one to specify the time of day. Conversion codes are short text strings, starting with a '%' that tell strptime how to interpret a sequence of bytes. R supports a few dozen conversion codes. A full list is available in strptime's help file, but a selection is given in Table 3.6.

Most conversion codes work in two directions: they can be used to both extract POSIXct data from text and to convert POSIXct to text. Conversion to text can be done with format.

```
format(t)
## [1] "1977-01-02 05:30:00"
format(t,"%Y-%m-%d",usetz=TRUE)
## [1] "1977-01-02 CET"
```

Finally, we mention a few other utilities from base R that may be useful for processing data. Functions ISOdate and ISOdatetime can be used to specify a date or time,

Table 3.6 A selection of time and date conversion codes accepted by R's time and date conversion function like strptime and format.

Code	Explanation
%a,%A	Abbreviated, full week day name[a).
%w	Week day as decimal number (0–6), Sunday is 0[b)
%V	Week number (00–53)[b)
%b, %B	Abbreviated, full month name[a).
%m	Month as decimal number (00–12)[b).
%d	Day of the month as decimal number (00–31)[b).
%y	Last two digits of the year.
%Y	Full, four-digit year[b).
%z	Singed time zone offset[b).
%Z	Time zone abbreviation (output only)[a).
%H	Hour of the day (00–24)[b).
%M	Minutes (00–60)[b).
%S	Seconds (00–61)[b).

The full list described in the help file of strptime.

a) Interpretation depends on locale.

b) Complies with ISO 8601.

element by element as numeric values and return a POSIXct object. Sys.time returns the current system time as a POSIXct object.

3.3.5 Leap Days, Time Zones, and Daylight Saving Times

It is possible to specify, in text, points in time or dates that are either ambiguous or that do not exist. For example, since 2011 is not a leap year, the 29th of February of that year is not a valid day.

```
lubridate::ymd("2011-02-29", quiet=TRUE)
## [1] NA
strptime("2011-02-29","%Y-%m-%d")
## [1] NA
```

Both functions properly detect invalid date specifications.

Both strptime and the ymd-like functions of lubridate allow for specification of the timezone.

```
lubridate::ymd("2014-10-25",tz="Europe/Amsterdam")
## [1] "2014-10-25 CEST"
as.POSIXct(strptime("2014-10-25","%Y-%m-%d",tz="Europe/Amsterdam"))
## [1] "2014-10-25 CEST"
```

Here, strptime also allows for time zone specification within the string, using the %z conversion code.

```
as.POSIXct(strptime("2014-10-25 +0100","%Y-%m-%d %z"))
## [1] "2014-10-25 01:00:00 CEST"
```

Note that we have encountered three time zone representations: a numerical offset from UTC (the ISO 8601 standard), a long textual form such as Europe/Amsterdam, and an abbreviated form such as CEST (Central European Summer Time). The time zone to which the system running R is set can be requested with Sys.timezone(). A list of all available long formats can be requested with OlsonNames(). The function is named after the table of time zone data initiated by A. D. Olson and now maintained and distributed by IANA (2014). Operating systems frequently update to the latest 'Olson tables' to reflect changes in policies regarding DSTs and time zone borders. Recall that in POSIXct, a point in time is stored as the number of seconds after 1970-01-01 00:00 at UTC, so one may confirm that the following two time specifications have the same POSIXct representation.

```
as.integer(as.POSIXct(strptime("2014-12-10 15:00","%Y-%m-%d %H:%M"
  , tz="Europe/London")))
## [1] 1418223600
as.integer(as.POSIXct(strptime("2014-12-10 16:00","%Y-%m-%d %H:%M"
  , tz="Europe/Amsterdam")))
## [1] 1418223600
```

Because of DST policies, it is possible to define, in a text string points in time that are either impossible or ambiguous. For example, in Spain, summer time 2014 started on 30 March, 02:00 when the clock was turned forward by 1 h. Consequently, the command

```
as.POSIXct(strptime("2014-03-30 02:30","%Y-%m-%d %H:%M"
  , tz="Europe/Madrid"))
```

cannot be expected to produce meaningful output. Unfortunately, the precise behavior of `strptime` is operating system dependent. Ambiguous situations occur when the clock is turned back an hour at the end of a DST period since the clock passes the same hour twice. Again, the precise behavior depends on the operating system running R. One way to resolve such cases is to pre-process text strings, filtering such invalid or ambiguous definitions using regular expressions (see Chapter 5).

Finally, we note that the help files for `as.POSIXct()`, `strptime()`, and `Sys.timezones()` contain detailed information on conversion, time zone data, and DSTs that are R-related. It is well worth studying these files if one is frequently encountered with (polluted) time data in text form.

3.4 Notes on Locale Settings

In the previous sections, we have frequently encountered references to locale settings. Locale settings are parameters and data present in an operating system that can be used by programmes to alter their behavior related to region, language, and encoding. Locale settings are standardized in the POSIX standard (The Open Group, 2013), where localization properties are subdivided in the following categories:

`LC_CTYPE`	Character classification and case conversion.
`LC_COLLATE`	Collation order.
`LC_MONETARY`	Monetary formatting.
`LC_NUMERIC`	Numeric, non-monetary formatting.
`LC_TIME`	Date and time formats.
`LC_MESSAGES`	Formats of informative and diagnostic messages and interactive responses.

An operating system conforming to the POSIX standard must at least support these categories. Many operating systems, including the ones on which R is supported, have additional locale categories. An overview of locale settings can be obtained from R in one of the two following ways:

```
Sys.getlocale("LC_ALL")
sessionInfo()
```

The former command returns a string of semicolon-separated locale settings, while the latter command returns an overview of locale settings as well as loaded packages, the R version, and more. To change a locale setting one uses `Sys.setlocale` in the following way:

```
Sys.setlocale("[locale_category]","[locale_identifier]")
```

In each locale category, a possibly large number of choices are made. For example, within the `LC_CTYPE` category alone, the set of valid upper- and lowercase characters is determined as well as how to translate between them. For this reason, one uses abbreviations called *locale identifiers* to identify such a set of choices. Locale identifiers are text strings in the following format:

```
[language]_[variant].[encoding]
```

Unfortunately, the notations for `[language]`, `[variant]`, and `[encoding]` are system dependent. The variant usually indicates a region where a variant of the language is spoken, for example, (on Linux) `en_US.utf8` and `en_UK.utf8`. POSIX-compliant systems use a standard two-letter notation for language that is defined by the Internet Engineering Task Force [IETF, Phillips and Davis (2009)]. However on Windows, language and variant are not abbreviated but spelled out in full.

To discover which locale identifiers are available on your system, under most Unix/Linux-like or Mac OSX, you can open a terminal (not the R terminal) and type the following command:

```
locale -a
```

Finding the available settings under Windows is a bit more involved. In a powershell window, you can type

```
[System.Globalization.Cultureinfo]::GetCultures(2)
```

to print the full list of available locale settings to screen. However, depending on your language settings, the output may or may not be directly usable for setting locales from R. There are several websites that list locale identifiers for Windows, for example, (Microsoft, 2014; Sheppard, 2014).

```
http://msdn.microsoft.com/en-us/goglobal/bb964664.aspx
http://ss64.com/locale.html
```

Note that both websites list the identifiers in `[language]-[variant]` format, while `[language]_[variant]` (with the underscore) must be used to set locales from R.

4

Data Structure

4.1 Introduction

An important step in data cleaning is getting raw data into a convenient structure for analysis. Data values must be coded in the right format, but the correct technical representation is not sufficient for making the data processable for statistical purposes. The collection of values must also be structured in a convenient manner. Raw data can have different structures, depending on the collection process or experimental setup. In this chapter, we will describe the commonly encountered structures. Most statistical analyses are done on tabular and matrix data, so a common step in analysis is restructuring and transforming raw data into tabular or matrix data that can be processed. In R, this is typically a `data.frame` or `matrix`.

Table 4.1 shows various data structures that are encountered when analyzing data.

Most data can be transformed into tabular format, but it does not follow that it can be analyzed directly: the columns of a table should denote the variables to be analyzed, and the rows should describe the members of the statistical population of interest.

4.2 Tabular Data

A table or dataset is a rectangular collection of values in which each column denotes a (measured) value of a variable. A row describes an observation and its columns contain the values for the different features or variables of the observation. Values in a column have equal type, but values on a row may have different types.

Statisticians are most familiar working with tabular data, in which a table contains all information needed for their analysis. In practice, this often means that they have to transform their collected data in tabular format or have to combine multiple tables into one table.

4.2.1 data.frame

Tabular data can be stored in text files or in databases. The R `data.frame` object was designed to store and manipulate tabular data in working memory. The R ecosystem also contains `data.table` and `tibble`, which provides optimized implementations, but all these conform to the same interface of `data.frame`. In a `data.frame`, each column is stored separately and has its own data type. This is probably one of the reasons

Table 4.1 Often encountered data types and their structure.

Data types	Data structure
Various	Tabular
Numeric	Matrix
Numeric + time	Time series
json/xml	Hierarchical
html	Text/hierarchical
Free text	Text

that R is a vector language: each column is a vector. Column-oriented storage is different from purely scientific programming environments such as matlab, which typically work with matrices in which all elements are numeric (see 4.3). While a `data.frame` stores the columns of a table, most other table storage formats store rows of data: tabular data in text files such as comma-separated values (csv) store each row on a separate text line. Also, almost all database systems store tabular data row by row. Row-based storage is convenient for looking up, viewing, and editing individual observations. However, statistical procedures are concerned with describing values of variables over many individuals, for which column-based storage is more convenient and efficient. In R, expressions work on vectors of data, so a `data.frame` is list of vectors of equal lengths.

Many packages provide functions that work on a `data.frame` or on `vectors`, so `data.frames` are very important data structures for working effectively in R.

4.2.2 Databases

Database systems are used for storing data in a structured format on disk. They handle Creating, Querying (Reading), Updating, and Deleting data, which are also known by the acronym CRUD. Most databases guarantee consistency and integrity and allow for fine-grained control over user access to the data. The most used and well-known databases are relational databases, but there also exist other types of databases, for example, object, hierarchical, and document-based databases. In a relational database, data are stored in tables. Each *table* is a set of records having the same *fields* or *attributes*. Each *record* is a set of values corresponding to its fields. For example, a record in the table "Person" could be: (Name: "John", HairColor: "Brown", ParentOf: "Michael", Age: "34"). Each column of a table represents a relation of an entity (e.g., "John (Person)") with an attribute (e.g., "hair color: Brown") or another entity (e.g., "ParentOf: Michael (Person)").

Most databases provide (a dialect of) structured query language (SQL) that can be used to query and update data of a table in the database (Table 4.2). Relational databases really shine in selecting and retrieving (small) subsets of data. An important aspect of relational databases is that SQL queries are based on relational algebra (Codd, 1970).

Table 4.2 Table 'Person' with two records.

Name	HairColor	ParentOf	Age
John	Brown	Michael	34
Michael	Blond	—	3

This algebra provides a sound theoretical foundation for querying data and makes it possible to optimize database queries. Relational algebra works on sets of records and always results in a set of records. Records within a set are of identical type and have the same attributes. Record set A can be manipulated using the following unary operators:

- A *projection* operator $\Pi_{a_1,\ldots,a_n}(A)$, which removes all attributes $a \notin (a_1, \ldots, a_n)$ of A. The projection operator selects columns of a table.
- A *selection* operator $\sigma_\phi(A)$ selects records from a table A. ϕ is a boolean proposition, which may be composed using the logical operators *or* ($\vee$), *and*($\wedge$), and *not* $\neg$. If ϕ does not hold for a record, it will be filtered out.
- A *rename* operator $\rho_{a/b}$ to rename attribute a into b.

Relation algebra defines the following binary operators for combining record set A with record set B:

- Operators *union* ($A \cup B$), *intersection* ($A \cap B$), and *set difference* ($A \setminus B$) with the additional constraint the elements of A and B must have the same attributes.
- Binary *join* operators: join
  - natural or inner join: $A \bowtie B$, which combines all records in A with records of B that have equal shared attributes
  - semijoin: $A \ltimes B$, which selects records in A that have equal shared attributes in B
  - antijoin: $A \rhd B$, which selects records in A that have no equal shared attributes in B
  - left outer join: $A ⟕ B$, which combines all records in A with records of B that have equal shared attributes. If no such B can be found, it will generate empty values for the attributes of B
- A binary *cartesian product*: $A \times B$ between two sets may not share the same attributes.

In many databases, data are normalized: redundancy in data is removed as much as possible. For example, having a People table for a married couple, their address is stored once in a separate Address table and their People records point to this address in the Address table. Less redundancy reduces consistency problems, because the database contains each datum only once. The downside is that normalized data makes tabular data less (human) readable and processable. For statistical analysis, data from databases often need to be combined or joined, to retrieve interpretable data.

In R, data can be retrieved from a database using various packages that implement the DB interface, which is provided by the package `DBI`. The generic methods of `DBI` allow for connecting to a database, create tables, and query tables using SQL statements. The results of a DB query will be a `data.frame`.

```
library(DBI)
con <- dbConnect(RSQLite::SQLite(), "test.db") # create an empty db
dbWriteTable(con, "mtcars", mtcars) # copy a table to the database

dbListFields(con, "mtcars")
dbReadTable(con, "mtcars") # retrieve data from the data base

# You can fetch all results:
res <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbFetch(res)
dbDisconnect(con)
```

The result can also be returned in chunks to allow for processing of large tables. Although the DBI interface is very useful for querying database from within R, it often results in scripts where SQL statements and R are interweaved, which increases the cognitive burden for the writer and reader of the script.

4.2.3 dplyr

The package `dplyr` offers the `tbl` object to abstract from data storage back ends. It makes it possible to work with `data.frames` and database tables with identical R code. `dplyr` (Wickham *et al.*, 2014) is used for filtering, selecting, summarizing, sorting, mutating, and joining data in tables. Table 4.3 shows that these verbs map nicely on relational algebra.

```
library(dplyr)

# project and rename columns
my_iris <- select( iris
                 , sepal_width = Sepal.Width
                 , species = Species
                 )
```

Table 4.3 Relational operators and their `dplyr` verb.

Action	dplyr
Project	`select`
Select	`filter`
Add columns	`mutate`
Summarize	`summarize`
Sort/order	`arrange`
group	`group_by`
Inner join	`inner_join`
Left join	`left_join`
Anti join	`anti_join`
Semi join	`semi_join`

```
# select rows
my_iris <- filter(my_iris, sepal_width > 3)

# group by
my_iris <- group_by(my_iris, species)

summarize( my_iris, count = n()
         , mean=mean(sepal_width)
         )  # count total
```

The magrittr package implements the *pipe* operator %>% among other nifty features. The pipe operator transforms a %>% b(...) into b(a, ...).

This syntactic sugar makes code more readable: processing steps for data are written in sequence, instead of "inside out". R code also needs less temporary variables to store intermediate results. Combined with the magrittr package, the resulting R code is readable, fast, and portable. The previous code written with dplyr results in:

```
library(dplyr)

iris %>%   # data set
  select( sepal_width = Sepal.Width # project and rename columns
        , species = Species) %>%
  filter( sepal_width > 3) %>%      # select rows
  group_by(species) %>%             # group by
  summarize( count = n()            # calculate aggregates
           , mean=mean(sepal_width)
           )  # count total
```

4.3 Matrix Data

For describing and solving pure numerical data problems, a matrix is to be preferred. Matrix data are data of the same data type laid out in columns and rows. Time-series data (see Section 4.4) and hierarchical data (see Section 4.5) can be stored and delivered in matrix format. Because all data in a matrix have the same data type, (numerical) operations and algorithms tend to be faster or allow for a faster implementation in an R extension package written in C or C++.

In R, matrix data are often read via a data.frame, converting it to a matrix object. Note that when data is large, which is typically the case with matrix data, this can be (very) inefficient.

It is possible to write a matrix to a text file with the command write, but beware that the matrix needs to be transposed to have similar column row layout in text. Often, it is more convenient to store a matrix using the save or saveRDS methods.

Large matrices tend to be sparse. The R package Matrix and slam provide implementations for sparse matrices. Both provide a triplet storage: (row index, column index, and value) assuming that matrix cells without value have value 0. Matrix data can be read and stored in triplet format.

The slam library provides two functions to read and write from CLUTO or MC format, both forms of *Compressed Column Storage*.

Conversion between data.frame and matrix can be executed using as.matrix and as.data.frame.

```
A <- matrix(1:6, nrow=3, dimnames=list(NULL, c("a", "b")))
as.data.frame(A)
##   a b
## 1 1 4
## 2 2 5
## 3 3 6
as.matrix(as.data.frame(A))
##      a b
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
```

However, the conversion contains some intricacies and can be tricky:

First of all, one often forgets that a matrix is of one type. When converting from a data.frame to a matrix, the data type will be upcasted to the most general type, which often is a character.

```
dat <- data.frame(num=1:2, text=c("a", "b"))
M <- as.matrix(dat)
M
##      num text
## [1,] "1" "a"
## [2,] "2" "b"
as.data.frame(M)
##   num text
## 1   1    a
## 2   2    b
```

In this example, matrix M is converted to a character matrix, which is probably not wanted. Other conversion problems include converting from a data.frame with one column to denote the columns in the matrix, one column to denote the rows of the matrix, and a data column to a matrix. This is a so-called long format, for which it is handy to use the function spread from tidyr.

4.4 Time Series

In many sciences, the development or change of an indicator is the main interest of study. Often, a numerical variable is observed at fixed time intervals. Such a series of values is called a time series. R has many packages that allow decomposing and/or predicting time series (e.g., forecast, stl). Analyzing time series in depth is beyond the scope of this book. We restrict ourselves here to describing the data formats useful for time-series analysis. For further reading, the reader is referred to Hyndman and Athanasopoulos (2014). R offers several packages that structure data into time series. We discuss several of them. Time-series data can be in two flavors. Most commonly, this is data observed at a regular fixed frequency, for example, each year, month, or day. The data is sampled at a fixed frequency in time. The other flavor is irregularly observed: each datum has a date and time but not at a fixed frequency.

ts is the simplest and most common form of time series and is used for regular fixed frequency data. In contrast to a data.frame, a ts object stores one series indexed in time.

$$(x_1, \dots, x_n) \tag{4.1}$$

A ts object is indexed with a fixed frequency, which is specified upon creation. Take, for example, the Nile dataset, which shows the annual flow of the river Nile at Aswan between 1871 and 1970. The data is a vector with a starting date and a frequency.

```
str(Nile)
##  Time-Series [1:100] from 1871 to 1970: 1120 1160 963 1210
##  1160 1160 813 1230 1370 1140 ...
start(Nile)
## [1] 1871    1
end(Nile)
## [1] 1970    1
frequency(Nile) #annual data so frequency == 1
## [1] 1
```

To create a quarterly time series, create a ts object with frequency 4.

```
data <- rnorm(3*4) # 3 years
(quarter <- ts(data = data, start = 2012, frequency = 4))
##            Qtr1       Qtr2       Qtr3       Qtr4
## 2012  2.0229167 -0.6791606 -1.1546599 -0.3030735
## 2013 -1.2980148 -0.1642950 -0.1558252  1.2293501
## 2014  0.3460415 -0.2698534  0.9769135  1.0121808
```

The ts object has several methods that support time-series analysis. For example, plot plots the time series and monthplot shows the year-over-year development of an indicator.

Multiple indicator time series can be created by supplying a matrix as data argument instead of a vector. In that case, ts will create an object of type mts.

```
data <- matrix(rnorm(2*4), ncol=2, dimnames=list(NULL, c("a","b")))
(quarter <- ts(data = data, start = 2012, frequency = 4))
##                  a          b
## 2012 Q1 -1.0311934 -1.9363831
## 2012 Q2 -0.3009061  0.7429729
## 2012 Q3 -0.5642486 -1.1506197
## 2012 Q4 -1.0489915  0.8579259
```

While ts is meant for regular time series with a fixed frequency, R package zoo and xts were developed to store discrete irregular time series.

zoo has handy function read.zoo for reading irregular time series. It supports skipping columns (using NULL) and merging multiple files.

Exercises for Section 4.4

Exercise 4.4.1 *Create a times series object starting at* 1990 *with value 100 and ending in* 2000 *in which for each even month, the value increases with* 2% *w.r.t. the previous month, and each odd month decreases with* 1%. *What is the final value?*

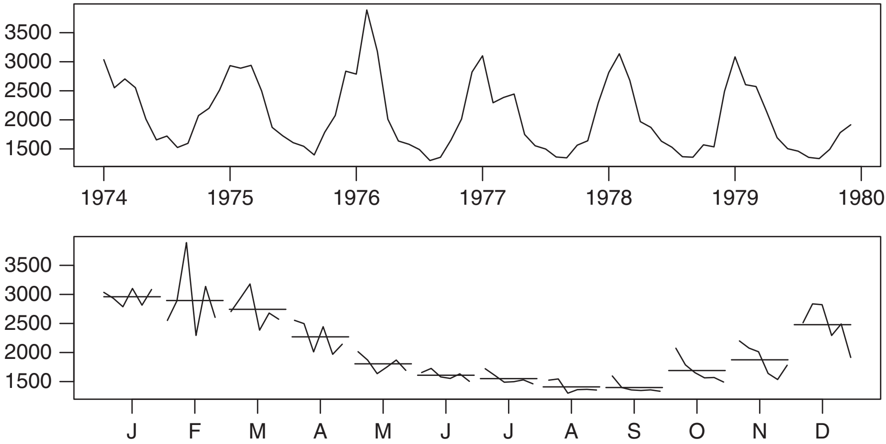


Figure 4.1 UK lung deaths.

Exercise 4.4.2

a) Recreate Figure 4.1 using the dataset `ldeaths`*.*
b) Create a `mts` *object from male and female UK lung deaths.*

4.5 Graph Data

With the uprising use of big data, many new data sources contain network information. Who is connected to whom? How do natural resources flow from country to country? What are the financial transactions? How are flights organized and what are the airports that function as a hub? All these data are network data. Network data are most easily seen as data with a mathematical graph structure. In graph theory, a graph is defined as

$$G = (V, E) \tag{4.2}$$

with V a set of vertices or nodes and E a set of edges or links. An edge connects two nodes and may have a direction or not. A graph is called *undirected* when it contains only undirected edges, *directed* when all edges are directed, and otherwise *mixed*. A *path* in a graph is a sequence of edges that connects a sequence of nodes: starting with a node follow an edge to the next node and so forth. A path in which all nodes are unique is called *acyclic*. A graph in which all paths are acyclic is called acyclic.

A special case of network data are hierarchical data. Hierarchical data are *directed acyclic* graphs. In hierarchical data, the data are shaped in a tree structure as shown in Figure 4.2.

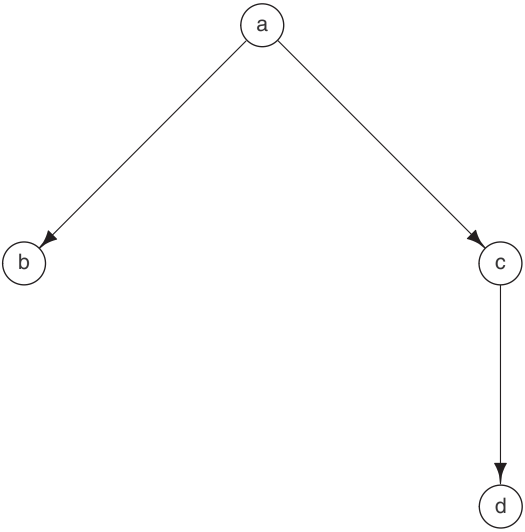
Network data can be transformed into one of the various formats:

- A table with *edges*. Each row denotes an edge, its source node, its target node, and optionally other properties of the edge.

Source	Target
a	b
a	c
c	d

This format can be stored in the tabular format, as discussed earlier.

Figure 4.2 Directed graph.



- An adjacency matrix, in which the edges are encoded as 1 between nodes.

$$\begin{array}{ccccc} & a & b & c & d \\ a & 0 & 1 & 1 & 0 \\ b & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 1 \\ d & 0 & 0 & 0 & 0 \end{array}$$

  This format is stored in a sparse or dense matrix format. Note that an undirected graph has a symmetric adjacency matrix and that an adjacency matrix allows for different weights for each edge.

- An adjacency list, in which per vertex the connected vertices are listed.

$$\begin{array}{rl} a & [b, c] \\ b & [] \\ c & [d] \\ d & [] \end{array}$$

  This is a more compact format than previous formats. Although it is not useful in analysis, it is often encountered when processing network data, for example, Twitter accounts and the other accounts they follow.

Tree structure data are often available as XML or JSON text files that mimic the tree structure of the data in a nested XML or JSON structure.

While R libraries that handle specific tree and network structures exist, for example, R package igraph, it is often handy to convert XML and JSON into tabular or matrix format, which will be discussed in Section 4.6.

4.6 Web Data

An increasing amount of data is available on the internet. Data on the internet are stored in a structured format that can be downloaded. However, two other forms of data retrieval are found more often: Web scraping and using Web APIs.

4.6.1 Web Scraping

Web scraping is extracting structured information from un- or semistructured textual web pages. For example, it can be used to extract prices from web shops to monitor price fluctuations or to extract a table with data from a website. Web pages are coded in HyperText Markup Language (HTML), which is a markup language offering a lot of freedom for structuring text in paragraphs, headers, and sections. The basics of web scraping are simple: download the text of a web page, extract specific texts from it, and transform it into structured values. Although HTML allows for structured markup, most web pages have an inconsistent structure or are not structured properly. This makes web scraping a bit of a dark art. Since R can read web pages with `readLines`, it is tempting to use regular expressions (see Section 5.2) to extract the useful text bits. Because of the variation and HTML inconsistency of many web pages, this is error prone and is not recommended. It is wiser to use a dedicated package to help you scrape. For example, the `XML` and `xml2` package can parse valid and invalid HTML and extract HTML tables into `data.frame`. The `rvest` package from RStudio helps even further with a selectorgadget with which a user selects in an HTML page the relevant parts. HTML Web scraping has a severe limitation: modern websites contain javascript, which generates HTML on the fly. These pages can not be scraped with most web scraping tools: you need to execute the javascript to retrieve the HTML. In that case the R package RSelenium can be used in combination with phantomjs. Phantomjs is a headless browser. It acts as a real browser executing the javascript and generating HTML but without visual rendering and presenting the webpage.

Modern websites, however, often have a better option than scraping: they use Web APIs that can be called directly.

4.6.2 Web API

Modern websites offer application programming interfaces (API) to let users extract data from their websites. Often Websites use their API internally to generate their content "on the fly": the website pages are templates that are filled with values returned from the API. A Web API uses the HTTP GET protocol to retrieve data. Although a Web API can support Create, Read, Update, and Delete actions comparable to a relational database, we restrict ourselves here to reading data from Web APIs. For a more complete usage of Web API, see R package `httr`. Reading data from a Web API works identical to retrieving a web page; however, data is now returned in a structured format. Most used formats are XML and JSON. For example, the US census bureau offers several Web APIs for retrieving data: `http://api.census.gov/data/2000/sf1` shows the metadata of 2000 Decennial summary file 1 in JSON format. While it is useful to know how to download and process data using a Web API, CRAN contains specialized packages for often used Web APIs,`twitteR` retrieves data from the twitter api, `censusapi` and `tidycensus` from the US Census Bureau, and `eurostat` from the European Statistical Office, to name a few. Often, it takes less time installing these specialized packages and retrieving data than finding out yourselves how to download and parse the data.

XML

XML stands for extensible markup language. It is a structured text format which looks similar to HTML but is very complex in structure. If the structure is not followed, it is an invalid XML document. An XML document forms a tree structure in which each document node has attributes and can contain text or child nodes. An XML document may declare that it follows an xml schema, which defines a vocabulary of *tags*, also known as *nodes* and *attributes*, that may be used in the document. XML document can easily be checked for conforming to a schema. Retrieving information from a document must be done by selecting document nodes and reading their text or attribute values. XML is widely supported in many programming languages. In R, the `XML` and `xml2` packages provide reading and querying XML documents. Although XML is structured, it is verbose and not very human readable. Most XML documents have a deep tree structure, making extracting values cumbersome. There is no natural mapping from an XML document to R object.

The following example is an XML from the World Bank Web API:

```
<wb:data>
<wb:indicator id="DPANUSIFS">Exchange rate (IFS), LCU per USD,
    period average</wb:indicator>
<wb:country id="BR">Brazil</wb:country>
<wb:date>2009M01</wb:date>
<wb:value/>
<wb:decimal>0</wb:decimal>
</wb:data>
```

This data can also be downloaded using the `wbstats` package.

```
{ "indicator": {"id":"DPANUSIFS","value":"Exchange rate (IFS),
  LCU per USD, period average"},
  "country":{"id":"BR","value":"Brazil"},
  "value":null,
  "decimal":"0",
  "date":"2009M01"
}
```

JSON

JSON stands for JavaScript Object Notation and is a strict subset of JavaScript, without executable parts. It only allows defining objects, arrays, and simple types. It is used as a data exchange format on the web. The JSON data format is widely supported in many programming languages including R. Packages `rjson`, `RJSONIO`, and `jsonlite` provide JSON reading and writing functionality. Ooms (2014) clearly describes how jsonlite maps JSON objects on R objects and vice versa, making the translation between JSON and R very smooth. `jsonlite` can translate back and forth `data.frames` to commonly used JSON format, turning using a Web API into a function that returns a `data.frame`.

Figure 4.2 shows the JSON result of a web query on the website of the World Bank.

```
library(jsonlite)
fromJSON("http://api.census.gov/data/2000/sf1/tags.json")
## $tags
##  [1] "aian"             "black"            "children"
##  [4] "family"           "household"        "native hawaiian"
##  [7] "nonfamily"        "pacific islander" "population"
## [10] "race"             "sex"
```

```
fromJSON("http://api.worldbank.org/countries/bra?format=json")[[2]]
##    id iso2Code   name region.id               region.value
      adminregion.id
## 1 BRA      BR Brazil      LCN Latin America & Caribbean            LAC
##                                    adminregion.value incomeLevel.id
## 1 Latin America & Caribbean (excluding high income)           UMC
##     incomeLevel.value lendingType.id lendingType.value capitalCity
## 1 Upper middle income            IBD              IBRD    Brasilia
##   longitude latitude
## 1  -47.9292 -15.7801
```

Exercises for Section 4.6

A common case is that the input data is raw data that is scraped from the internet pages. The raw data is often in html/xml or json but needs to be transformed into tabular data.

Exercise 4.6.3 *The CRAN website contains a page with a list of all R packages that are currently published on CRAN.* *`http://cran.r-project.org/web/packages/available_packages_by_date.html`.*

a) *Use the `htmlTreeParse` method from R library `XML` to extract a table with package name, description, and date.*
b) *Make a histogram of the publication dates.*
c) *Use the `rvest` package to retrieve packages instead of the `XML` library.*

4.7 Other Data

Exciting new objects of data analysis have ideosyncratic data structures. Nowadays, it is possible to do analysis on images, video, or sound. Images are stored in jpg or png formats, video in mp4 or avi, and sound in mp3 or wav files. Text analysis of social media works on raw or semistructured texts.

Although packages specialized in analyzing these data accept various media formats, many of these data sources, if not all, are transformed into a tabular, matrix, or hierarchical format for further analysis. For example, metadata of these formats can be seen as tabular data. An image is a matrix of pixels in which their color represents a value. Sounds can be analyzed as vectors or time series. Raw text is often analyzed as a bag of words, which is a vector of word frequency.

4.8 Tidying Tabular Data

Having tabular data does not mean that it is fit for analysis: your data needs to be tidy. In JSS Tidy Data, Wickham (2014b), the term **Tidy Data** is coined. A tidy dataset is tabular data in which:

- Each column is a variable.
- Each row is an observation.
- All observations of a table are of the same observational unit type.

If any of these constraints are broken, data is called untidy or messy.

For example, Table 4.4 is messy: the statistic that is counted is the number of lung deaths, but the columns contain the count for the different sexes. Table 4.5 contains the same data, but in tidy format.

Many tabular data are untidy, and often they have one or more of the following problems.

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.
- Multiple types of observational units are stored in the same table.
- A single observational unit is stored in multiple tables.

Note that this definition depends on how you define the observation unit in your analysis. Tidy data for one purpose may be messy data for another purpose if the observation definition is different. It might seem rare, but this is common when working with observations with (complex) relations. For example, a table where each row describes several features of persons including income would be considered tidy data when the object of your analysis is income distribution on persons. However, if the target object is household income, this data is messy: a row is not equal to a household but contains a member of an household.

Table 4.4 Number of lung deaths in "messy" format.

Month	Male	Female
Jan 1979	2263	821
Feb 1979	1820	785
Mar 1979	1846	727

Table 4.5 Number of lung deaths in "tidy" format.

Month	Sex	Deaths
Jan 1979	Male	2263
Feb 1979	Male	1820
Mar 1979	Male	1846
Jan 1979	Female	821
Feb 1979	Female	785
Mar 1979	Female	727

The bottom line is that often one needs to restructure tables into tidy data, which is fit for the purpose of analysis. Package `tidyr` was created to support that activity.

Consider the following example of messy data. Note that the unit of observation is a financial post or transaction, not the person concerning the transaction.

```
library(tidyr)
library(dplyr)

finance_messy <- data.frame(
  name = c("Alice", "Bob", "Carla"),
  tax_2015 = c(40, 10, 2),
  income_2015 = c(41, 90, 100)
)

finance_messy
##    name tax_2015 income_2015
## 1 Alice       40          41
## 2   Bob       10          90
## 3 Carla        2         100
```

Column Headers Are Values

It is quite common that column headers in a table are values. Tables often have squared layout that helps with reading the values from the table. This is often the case with scraped data from the internet.

The `gather` function will transpose the data:

```
finance_messy %>%
  gather(variable, amount, -name)
##    name    variable amount
## 1 Alice    tax_2015     40
## 2   Bob    tax_2015     10
## 3 Carla    tax_2015      2
## 4 Alice income_2015     41
## 5   Bob income_2015     90
## 6 Carla income_2015    100
```

The statement collects all numeric columns in the `amount` column and move the column headers into a separate column.

4.8.1 Variable Per Column

Often, raw data contains multiple variables code in one column. For example, the values of such a column can be separated with `separate`. In our example this results in:

```
finance_tidy <-
  finance_messy %>%
  gather(variable, amount, -name) %>%
  separate(variable, into = c("variable", "year"))
```

where we split the variables in two columns.

It is also possible that a variable is spread on multiple columns, for example, year, month, day, and time. The function `unite` can be used to concatenate columns.

With spread we can transform the finance_tidy data back into a dataset about individuals per year.

```
finance_tidy %>%
  spread(variable, amount)
##    name year income tax
## 1 Alice 2015     41  40
## 2   Bob 2015     90  10
## 3 Carla 2015    100   2
```

4.8.2 Single Observation Stored in Multiple Tables

This problem often occurs when working with databases in which data is normalized. It can be mitigated by joining the tables and creating a single table or view for each observation. For this purpose, dplyr joining functions, for example, inner_join and left_join, can be used.

5

Cleaning Text Data

Preparing text data takes place on a number of levels. From the bottom up there are technical (encoding) issues, string issues which are related to the structuring of the storage format, case folding, and so on, and semantic issues which relate to the meaning of text.

Cleaning on the technical level mostly concerns solving encoding issues. That is, can we properly interpret the sequence of bytes comprising a file as a set of symbols? We will find that there is no definite answer to this but there are tools that implement effective heuristics that work in many cases.

At the string level, we find issues like checking the file format (is a file proper `csv` or `HTML`?), cleaning up punctuation or spurious white spaces, and localizing typos by approximate matching against a list of known or allowed terms. It also includes extracting the interesting portions of text string, for example, by harvesting the `url` that is stored in a `<a href="[url]">` tag of an `HTML` file. Activities at the string level therefore consist of more or less classical techniques for finding, replacing, and substituting substrings as well as approximate matching of strings based on string metrics. We shall find that methods based on pattern matching and methods based on approximate matching are in a way complementary approaches to common text cleaning problems and it is effective to combine them.

The semantic domain involves activities that make use of the *meaning* of text. Here, one may distinguish many sublevels again by looking at the meaning of single terms, terms in the context of a sentence, sentences in the context of a text, and so on. Following this path we quickly move into the domain of semantic text analyses which is beyond the scope of this book. However, we discuss a few tools and techniques that are readily available such as reducing verbs to their stem and language detection.

Notation of alphabets and strings: We introduce a few concepts and notation that will be used throughout this chapter. We already encountered the term *alphabet* in Section 3.2.1, where it was defined as a nonempty finite set of abstract symbols. Alphabets are denoted with the Greek capital letter Σ. Examples include the binary alphabet $\Sigma = \{0, 1\}$ or the complete set of Unicode characters. Any finite nonempty subset of an alphabet is also an alphabet.

The symbols of an alphabet can be concatenated to form strings. The set of all strings that consist of zero or any finite number of characters is denoted as Σ^*. Elements of Σ^* are denoted as s, t, or u and string literals will be printed between quotes. For example, if $\Sigma = \{0, 1\}$, then $s =$ `"0110011"` is an element of Σ^*. The string consisting of zero characters is a special case with its own notation: ϵ. The *length* of a string is the number of characters it is composed of (in Unicode: code points). We will use the notation $|s|$ to indicate the string length. We have $|\epsilon| = 0$ and `|"001101"|` $= 5$. The set of all strings of length n is denoted as Σ^n.

The definition of Σ^n gives rise to a more formal definition of Σ^*, namely,

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n. \tag{5.1}$$

In fact, we may consider * as an operator that generates concatenations. We will encounter this operator again when we discuss regular expressions.

5.1 Character Normalization

Normalization is the process of making choices consistent. That is, in many cases, the same information may be represented in more than one way and before any further processing it is convenient to represent the same information in the same way. For example, the strings `"Sarah"`, `"sarah"`, and `"SARAH"` all refer to the same person but in a different representation.

Character normalization comes in two flavors: normalization of the encoding (technical representation of text) and normalization of symbols (character conversion). Both cases are discussed in the next section.

5.1.1 Encoding Conversion and Unicode Normalization

A fundamental step in text cleaning is making sure that all character data is stored in a single encoding scheme. In principle, any encoding that supports the alphabet the text is written in suffices. However, because of its broad support across many systems, and software, and because it extends `ASCII`, it is a good idea to use `UTF-8`, at least for internal representation of text.

Detecting in what encoding a file is stored was discussed in Section 3.2.7. Also recall that text data read in R is always internally stored as `latin1` (Windows) or `UTF-8` (others). Base R has several tools available to convert encoding, all of them based on an implementation of the `iconv` specification (The Open Group, 2004). To convert all elements of a character vector to `UTF-8`, one can use `enc2utf8()`. Converting between any pair of encoding schemes can be done with `iconv()`.

```
y <- iconv(x, to="[encoding]", from="[encoding]")
```

The encoding specifications that are supported depend on the system running R and can be requested by typing the following command.

```
iconvlist()
```

The `stringi` package is not based on `iconv` but utilizes the `ICU` library (ICU, 2014) instead, thus guaranteeing platform independence. The `stringi` equivalent of `enc2utf8` is `stri_enc_toutf8()`

```
y <- stringi::stri_encode(str, from="[encoding]", to="[encoding]")
```

The valid encoding specifications can be obtained with the following command:

```
stringi::stri_enc_list()
```

The `stringi` package supports less encoding schemes than `iconv` on most or all systems supported by R; but unlike R, it does guarantee platform independence.

As discussed in Section 3.2.3, some symbols can be represented in more than one way in Unicode. As an example, consider singly accented Latin characters. These may be represented by one byte sequence representing the accented character, or by two byte sequences where the first determines the character and the second adds the accent. Before any further processing it is a good idea to normalize such cases since, for example, find-and-replace operations performed later on in the text may be sensitive to the underlying representation.

Unicode has several normal forms. As a first example, we normalize to the NFC form which is recommended by the Internet Engineering Task Force [IETF, Klensin and Padlipski (2008)]. To demonstrate Unicode normalization, we first define the letter ‘ö’ in two ways:

```
x <- c("\uf6"         # small latin 'o' with diaereses
      , "\u6f\u0308") # small latin 'o' followed by combining diaereses
x
## [1] "ö" "ö"
```

The stored byte sequences can be revealed with `iconv`.

```
iconv(x,toRaw=TRUE)
## [[1]]
## [1] c3 b6
##
## [[2]]
## [1] 6f cc 88
```

We now normalize each element of `x` and again inspect the byte sequences.

```
y <- stringi::stri_trans_nfc(x)
iconv(y,toRaw=TRUE)
## [[1]]
## [1] c3 b6
##
## [[2]]
## [1] c3 b6
```

So in NFC the collapsed version of ‘ö’ is preferred.

The abbreviation NFC stands for Normal Form canonical Composition. The algorithm behind it first decomposes, then recomposes (accented) characters in a predefined, unique order. Not all combined characters are decomposed by NFC. For example, ligatures such as ‘ﬀ’ are left alone by NFC (a ligature combines two or more letters to a single symbol, e.g., ffi becomes ﬃ). The NFKC normalization (Normalization Form canonical compatible Composition) also takes those cases into account. Compare the following results:

```
x <- "\ufb00" # Latin small ligature ff
x
## [1] "ff"
nchar(x)
## [1] 1
nchar(stringi::stri_trans_nfc(x))
```

```
## [1] 1
nchar(stringi::stri_trans_nfkc(x))
## [1] 2
```

In the first case, the normalized x still consists of a single character representing the ligature 'ﬀ'. In the second case, the ligature character is split into two separate f's. As a consequence, the letter 'f' will only be detected after normalization.

```
grepl("f", stringi::stri_trans_nfc(x))
## [1] FALSE
grepl("f", stringi::stri_trans_nfkc(x))
## [1] TRUE
```

Here, we used `grepl` to determine whether the letter 'f' occurs in the string.

Finally, we note that there are two more normal forms which are less interesting from a text cleaning point of view, namely, NFD (Normalization Form canonical Decomposition) where (e.g., accented) symbols are decomposed into constituting parts and NFKD (normalization form compatibility decomposition) where combined (e.g., ligature) characters are decomposed.

5.1.2 Character Conversion and Transliteration

Character conversion is a simple process where a user-specified set of symbols is replaced by another set of symbols. Mathematically, it can be interpreted as a function ϕ sending symbols from an alphabet Σ to an alphabet Σ'

$$\phi : \Sigma \to \Sigma'. \tag{5.2}$$

Here, ϕ can be one-to-one or many-to-one and Σ and Σ' may or may not be equal. Since there is no such thing as an 'empty character', mappings like ϕ cannot be used to remove characters from a string. Character removal should be interpreted as an operation on strings where a substring (possibly of length 1) is replaced by the empty string. Such operations are discussed in the next section 5.2.

General conversion maps can in R be specified with the function `chartr`.

```
chartr(old = "[character]", new = "[character]", x = "[character]")
```

It takes as input a string of characters to convert, a string of characters to convert to, and a character vector of strings to convert. As an example, consider the soundex algorithm (see also Procedure 5.4.1). One step of this algorithm involves replacing every vowel (of the ASCII alphabet) by the small latin letter 'a'. Using `chartr` this can be specified as follows:

```
chartr("eiou","aaaa", c("hello world","Olá Mundo"))
## [1] "halla warld" "Olá Manda"
```

For each `old` character, a `new` version must be provided. The `chartr` function is general enough to define any map of the form (5.2). That is, any translation that can be done character by character. This example shows that `chartr` requires a precise specification: both the capital letter 'O' and the accented á are not recognized and must be specified explicitly.

There are two character translation jobs that are so common that they are accessible as a separate functionality: case conversion (also called case folding) and transliteration.

The base R functions `toupper` and `tolower` convert a string to completely upper or lowercase. The mapping between upper and lowercase is determined by locale setting (`LC_CTYPE`), see Section 3.4) and need not be specified explicitly by the user.

```
toupper(c("hello world","Olá Mundo"))
## [1] "HELLO WORLD" "OLÁ MUNDO"
tolower(c("hello world","Olá Mundo"))
## [1] "hello world" "olá mundo"
```

The `stringi` package also offers a function to translate words to upper, lower, or title case.

```
stringi::stri_trans_tolower(c("hello world","Olá Mundo"))
## [1] "hello world" "olá mundo"
stringi::stri_trans_toupper(c("hello world","Olá Mundo"))
## [1] "HELLO WORLD" "OLÁ MUNDO"
stringi::stri_trans_totitle(c("hello world","Olá Mundo"))
## [1] "Hello World" "Olá Mundo"
```

The advantages of the `stringi` package over base R are the ease of switching between locales (by the argument `locale`) and guaranteed consistency across operating systems. Moreover, the title case conversion, which necessarily involves word detection, has detailed controls for word boundary detection.

Transliteration is a mapping of the form (5.2), where Σ' is either a subset of Σ or an altogether different alphabet from Σ. A common use of transliteration is to replace accented characters by characters without accent. The actual transformations are governed by library-specific transliteration rules that determine how to convert from one alphabet to another. In base R, transliteration can be achieved by appending `//TRANSLIT` in the encoding specification of `iconv`. The `stringi` package offers a specialized function called `stri_trans_general`.

```
iconv(c("hello world","Olá Mundo"), to="ASCII//TRANSLIT")
## [1] "hello world" "Ola Mundo"
stringi::stri_trans_general(c("hello world","Olá Mundo"),
id="latin-ascii")
## [1] "hello world" "Ola Mundo"
```

In both cases, the non-ASCII character `á` has been replaced by `a`.

5.2 Pattern Matching with Regular Expressions

The ability to recognize a substring within a larger string lies at the heart of many string cleaning actions. Actions like trimming of white spaces, punctuation removal, or splitting strings into separate lines or words all rely on the fact that patterns in strings can be recognized.

The theory of pattern recognition in strings is closely tied to early developments in theoretical computer science. In fact, the so-called regular expressions, which may be considered one of the most basic tools in text processing, were first developed by Kleene (1951) in a paper on "Representation of events in nerve nets and finite automata". The algebraic theory of regular expressions is nowadays well understood and described in standard textbooks on the foundations of computer science, such as the excellent book by Hopcroft *et al.* (2007). In this chapter, we give a compact overview of the most important features of regular expressions, focusing on their use within R. For a very thorough and complete overview of practical regular expressions, we recommend the book by Friedl (2006).

For our purposes, a regular expression can be interpreted as a short notation for defining a range of strings over an alphabet. Computer programs such as `grep` or `sed`, which are a standard part of `Unix`-like operating systems, accept regular expressions and are able to very efficiently find, respectively find and replace substrings in larger pieces of text.

5.2.1 Basic Regular Expressions

A regular expression is a short notation that indicates a subset of Σ^*, the set of all strings that can be composed of the symbols in an alphabet Σ. Regular expressions are capable of denoting finite or infinite subsets of Σ^*, but not every subset of Σ^* can be denoted by a regular expression.

All regular expressions can, in principle, be built up out of a few basic elements. As an example, consider the following regular expression that already contains all basic operations:

$$\texttt{(a|b)c*} \tag{5.3}$$

This expression stands for the set of strings that start with an *a* or a *b*, followed by zero or more *c*'s. In other words, it corresponds to the set

$$L = \{\texttt{"a"}, \texttt{"b"}, \texttt{"ac"}, \texttt{"bc"}, \texttt{"acc"}, \texttt{"bcc"}, \dots\}. \tag{5.4}$$

To understand the notation, we read expression (5.3) from left to right. Between brackets, we find `a|b`, where the vertical bar `|` is the familiar 'or' operator. Since these are the first substrings in the expression, this means all strings indicated by (5.3) must begin with either `"a"` or `"b"`. Next, we find a `c`, followed by an asterisk `*`. The asterisk operator may be read as 'zero or more of the previous', where 'the previous' only extends to the character immediately previous to it in this case.

Before continuing to the formal definition of these operators, let us look at an example of how regular expressions can be used in R. The function `grepl` accepts two main arguments: a regular expression and a character vector. It returns a logical vector indicating for each element of the character vector whether a *substring* matches the regular expression.

```
str <- c("a","xacc","xdcc","xbccccxyac")
grepl("(a|b)c*",str)
## [1]  TRUE  TRUE FALSE  TRUE
```

Here, the first element of `str` matches since the whole string (`"a"`) is in *L* of Eq. (5.4). The second string, `"xacc"`, is not in *L*, but since it has a substring `"acc"` that is, it matches. The same holds for the fourth element, while the third element `"xdcc"` has no substring that is in *L*.

The function `sub` replaces the first substring that matches a regular expression with another string. For example, to replace matching substrings from the example with `"H"`, we do the following:

```
sub("(a|b)c*", "H", str)
## [1] "H"        "xH"       "xdcc"     "xHxyac"
```

Observe that, indeed, while the fourth element of `str` has two substrings matching the regular expression, only the first is replaced. The function `gsub` replaces all occurrences.

```
gsub("(a|b)c*", "H", str)
## [1] "H"      "xH"     "xdcc"   "xHxyH"
```

Formal Regular Expressions

Put together in a set, all regular expressions together form an algebra. That is, regular expressions can be built up and combined systematically by applying a few basic operations, possibly over and over again. Each such operation yields a new and valid regular expression.

The first three rules for constructing regular expressions just introduce some notation and relate regular expressions to sets of strings.

Rule 1 The symbols ϵ and $\emptyset$ are regular expressions, representing, respectively, the empty string `""` and the empty subset of Σ^*.
Rule 2 If a is a symbol in Σ, then `a` is a regular expression representing {`"a"`}.
Rule 3 If E is a regular expression, then so is (E).

The first rule defines mathematical cases that are necessary for a complete algebraic development of regular expressions. They will not be important for our purpose. Brackets have exactly the effect one expects of them: they determine the order in which combining operations are executed, just like in normal arithmetic. The next set of rules determines how regular expressions can be operated upon.

The fourth and fifth rules introduce concatenation and 'or'-ing of regular expressions.

Rule 4 If E and F are regular expressions, then so is EF.
Rule 5 If E and F are regular expressions, then so is $E|F$.

The | notation indicates that a position in a string can be occupied by either of its operands. For example, the regular expression `a(a|ab)a` indicates the set {`"aaa"`, `"aaba"`}. The set of strings defined by the concatenation of two regular expressions is the concatenation of all strings of the first with all strings of the second set. In other words, if L and M are the sets of strings defined by regular expressions E and F, then the set defined by EF is defined as $LM = \{st|s \in L, t = M\}$. For example, take

$E =$ `(a|b)c`, so $L =$ {`"ac"`, `"bc"`}

$F =$ `b(aa|a)`, so $M =$ {`"baa"`, `"ba"`}

Concatenating E with F, we get

$EF =$ `(a|b)cb(aa|a)` and $LM =$ {`"acbaa"`, `"acba"`, `"bcbaa"`, `"bcba"`}.

The operations introduced thus far only allow us to define finite sets of strings. The star operator, introduced in the last rule, changes this.

Rule 6 If E is a regular expressions, then so is $E*$.

The star operator indicates 'repeat the preceding symbol zero or more times'. The operator was first introduced by Kleene (1951), and it is therefore often referred to as the

Kleene (star) operator. For example, the regular expression `a*ba*c` corresponds to the set

{`"bc"`, `"abc"`, `"bac"`, `"aabc"`, `"abac"`, `"baac"`, … }.

To complete our discussion of formal regular expressions, we need to define the rules for operator precedence. That is, we need to define rules so we understand whether for example `a|b*` should be read as `(a|b)*`, as `a|(b*)`. The rule is that the `*` goes first, then comes concatenation, and finally `|`. In other words, the expression `ab|c*` must be read as `(ab)|(c*)`.

Limits of Regular Expressions

As stated before, regular expressions are finite notations indicating subsets of Σ^*. In particular, the Kleene operator allows us to indicate infinite sets of strings with a regular expression of finite size. A natural question is whether every possible subset of Σ^* can be denoted with a finite regular expression. The answer to this question is 'no'. In fact, subsets of Σ^* that can be defined by a regular expression are called *regular*.

There are several ways to prove this. One way is to show that the number of subsets of Σ^* is much larger (uncountably infinite) than the number of sets that can be indicated by regular expressions (countably infinite). A second way is by giving a counterexample. For example, the set of strings defined by

L = {`"ab"`, `"aabb"`, `"aaabbb"`, … }

cannot be denoted by a finite regular expression. The actual proof of this statement relies on automata theory and is beyond the scope of this book. It can be found in many textbooks on languages and automata theory, such as Hopcroft *et al.* (2007). However, the result can be made intuitively clear by attempting to build a regular expression for L. For example, the expression `aa*bb*` fails, since this also includes strings not in L, such as `"abb"` and `"aaab"`. Another natural candidate would be `ab|aabb|aaabbb|...`. However, this expression is clearly not finite and so contradicts our demand that L be expressed with a finite regular expression. The main intuition here is that regular expressions lack a certain sense of memory. To be able to tell whether a string is in L or not, one needs to count the number of `"a"`'s and then verify whether ensuing number of `"b"`'s is the same. Since we have no way to count the number of characters, it is not possible to define a regular expression that defines L. In the next section we shall see that *extended regular expressions* (EREs) add behavior and operators to relieve this limitation.

Because sets of strings can be defined in many ways, there is no easy general technique or rule to determine whether an infinite set of strings is definable by a regular expression or not. Rather, there are a set of mathematical techniques that have to be applied on a case-by-case basis. An easy to remember general rule, however, is that any set of strings where for each string in the set, the structure of a string at one position depends directly on the structure of the string at another position is not regular.

Exercises for Section 5.2.1

Exercise 5.2.1 *Describe the sets of strings defined by the following regular expressions. Give a description in English as well as in notation, as in Eq.* (5.4).

a) `a*|b*`
b) `(a|b)*`
c) `(ab)*`
d) `a(a*b)*`

Exercise 5.2.2 *Give regular expressions for the following sets of strings.*

a) For the alphabet $\{a, b\}$, all strings starting with `"a"`*.*
b) For the same alphabet, all strings starting with one or more `"b"`*'s.*
c) For the same alphabet, any string ending with an `"a"`*.*
d) For the same alphabet, any string containing at least two `"a"`*'s.*
e) For the alphabet $\{a, b, c\}$, all strings of at least length three, having exactly one `"c"`*, that is neither the first nor the last symbol of the string.*

Exercise 5.2.3 *Is it possible to build a regular expression that recognizes palindromes?*

5.2.2 Practical Regular Expressions

Thus far, we have discussed regular expressions in the more or less theoretical context of very simple alphabets such as $\{a, b\}$ or $\{a, b, c\}$. For larger alphabets it quickly becomes cumbersome to express string sets such as 'any string that begins with four arbitrary symbols' since that would involve 'or'-ing all characters of the alphabet.

For this reason, programs working with regular expressions include extra operators that indicate ranges of characters (called character classes), special versions of the Kleene operator to indicate (possibly finite) repetition, or the beginning or ending of a string.

The extensions that are supported ultimately depend on the software one uses, and there are many programs and programming languages around that support regular expressions. However, EREs specified in the Single Unix Specification (The Open Group, 1997) are supported as a minimum in many applications, including R. Perhaps, most notably, extensions added by the Perl programming language on top of ERE have become very popular and widely supported as well. Indeed, R also supports Perl regular expressions out of the box. The `stringi` package of Gagolewski and Tartanus (2014) also comes with an implementation of regular expressions. It is based on the ICU library (ICU, 2014), which offers another extension of the ERE regular expressions.

To complicate things a little further, the actual alphabet used to interpret regular expressions may depend on locale settings. For example, in R, the `LC_CTYPE` setting determines the alphabet (e.g., the Unicode alphabet) while `LC_COLLATE` determines how character equivalence classes are defined.

Character Ranges

There are four ways to indicate that a string must have any of a range of characters at a certain position. The first we have already seen: it involves using the 'or' operator. Secondly, to indicate a position where any single character may occur, one uses the `.` (period) as a 'wildcard' character: it stands for any symbol from the alphabet. For example, the regular expression `a.b` stands for all strings starting with `"a"`, followed by an arbitrary character, followed by `"b"`.

Third, expressions surrounded by square brackets `[ ]` can be used to indicate custom character ranges. So the range `[AaBb012]` is equivalent to `A|a|B|b|0|1|2`. If the closing bracket `]` is part of the range, it must be included as the first character after the opening bracket, *i.g.* `[]ABC]`. A range of consecutive characters can be indicated with a dash `-`, so, for example, `[abcd]` is equivalent to `[a-d]`. If the dash itself is part of the range, it must be included as the final character in the range, that is, `-|a|b|c|d` can be expressed as `[a-d-]`. It is also possible to negate a range of characters. For example, to express strings that start with `"a"`, then have any character, except `[c-f]` one uses the `^` at the beginning of the range: `a[^c-f]`.

Finally, a number of common character classes have been given predefined names. For example, the class `[:digit:]` is equivalent to `0-9`. Note that named character ranges have to be used within a bracketed expression, so a valid expression detecting any single digit would look like this: `[[:digit:]]`. An overview of named character classes is given in Table 5.1. Here, we give a few more examples.

Consider the problem of detecting a class of e-mail addresses from domains ending with `.com`, `.org`, or `.net`. With the operators and character classes just discussed, one may construct a regular expression like this:

```
[[:alnum:]][[:alnum:]]*@[[:alnum:]]
[[:alnum:]]*\.(com|org|net)                                  (5.5)
```

This expression detects strings that start with one or more alphanumeric characters, followed by a `@`, followed again by one or more alphanumeric characters, a period, and

Table 5.1 Character ranges in the POSIX standard (version 2), with ranges corresponding to symbols of the ASCII alphabet.

Range symbol	**ASCII range**	**Description**
. (period)	All	Any single character
`[:alpha:]`	`[a-zA-Z]`	Letters
`[:digit:]`	`[0-9]`	Numbers
`[:alnum:]`	`[a-zA-Z0-9]`	Letters and numbers
`[:lower:]`	`[a-z]`	Lowercase letters
`[:upper:]`	`[A-Z]`	Uppercase letters
`[:cntrl:]`	Characters 0–31	Control characters
`[:space:]`	Space, horizontal and vertical tab, line and form feed, and carriage return	Space characters
`[:print:]`	Printable characters	
`[:blank:]`	Space and tab	Blank characters
`[:graph:]`	`[:alnum:]` and `[:punct:]`	Graphical characters
`[:punct:]`	`! " # $ % & ' () * + , - . / : ; < = > ? @ [ ] ^ _ { | } ~`	Punctuation
`[:xdigit:]`	`[0-9a-fA-F]`	Hexadecimal numbers

The actual interpretation depends on the alphabet used, which in R depends on the locale settings.

then `com`, `net`, or `org`. Note that the period is prepended with a backslash to indicate that the period must not be interpreted as a wildcard but as a simple period.

Suppose, as a second example, that we want to detect filenames that end with `.csv`. The following regular expression detects patterns consisting of one or more arbitrary characters, followed by `.csv`.

`..*\.csv$` (5.6)

Here, we also introduced the dollar as a special character. The dollar stands for 'the end of the string'. Recall from Section 5.2.1 that R functions like `grepl` match any string that has a substring matching the regular expression. By appending the dollar sign we, explicitly specify that the `.csv` must be at the end of the string.

There are two more types of character classes that can be defined with bracketed expressions, both of them related to collation and character equivalence classes (see Section 3.2.8). For example, in one language, the `"ä"` may be a separate letter of the alphabet, whereas in another language it may be considered equivalent to `"a"`. The special notation for such equivalence classes is `[= =]`. To indicate a particular class, one specifies a single representative from an equivalence class between `[=` and `=]`. For example, if `"o"`, `"ó"`,`"ô"`, and `"ö"` belong to a single class, then the expression `[[=o=]p]` is equivalent to `[oóôöp]`.

In Section 3.2.8, we also discussed the example from the Czech alphabet, where the combination`"CH"` is considered a single character sorting between `"H"` and `"I"`. To distinguish such cases within a bracketed expression, one places multicharacter collations between `[.` and `.]`. So, for example, in `[[.CH.]abc]`, the `"CH"` is treated as a single unit.

Repetitions

The Kleene star operator adds the ability to express zero or more repetitions of a pattern in a regular expression. In the extended regular expressions of the Single Unix Specification, there are several variants that facilitate other numbers of repetitions, all of them having the same precedence order as the Kleene star. An overview is given in Table 5.2.

To demonstrate their purpose, we express the regular expressions of examples (5.5) and (5.6) more compactly as follows:

Eq. (5.5): `[[:alnum:]]+@[[:alnum:]]+\.(com|org|net)`

Eq. (5.6): `.+\.csv$`

Table 5.2 Extended regular expression notation for repetition operators.

Operator	Description
*	Zero or more
?	Zero or one
+	One or more
{*m*}	Precisely *m*
{*m*,}	*m* or more
{*m*,*n*}	Minimum *m* to maximally *n*

The combined use of character classes and repetitions can be very powerful. For example, to detect phone numbers consisting of 10 digits, the following regular expression will do.

```
        [0-9]{10}
```

Suppose we want to allow phone numbers that may have a dash between the second and third number, we might use

```
        [0-9]{2}-?[0-9]{8}
```

If we also want to include phone numbers where a blank character is used instead of a dash, we can expand the expression further.

```
        [0-9]{2}[[:blank:]-]?[0-9]{8}
```

Recognizing the Beginning or Ending of Lines

EREs have two special characters that anchor a pattern to the beginning or ending of a line.

^	indicates the beginning of a string
$	indicates the end of a string.

For formal regular expressions, such operators are not needed since they only define a precise range of strings, from beginning to end. However, programs (and R functions) such as grep recognize strings that have a substring matching a regular expression. As an example, compare the two calls to grep given here.

```
x <- c("aa","baa")
grepl("a(a|b)", x)
## [1] TRUE TRUE
grepl("^a(a|b)", x)
## [1]  TRUE FALSE
```

The first regular expression is interpreted by grep to match any string having a(a|b) as a substring. Therefore, both strings are matched. The second regular expression, ^a(a|b), explicitly requires the string to begin with an "a".

Special Symbols and Escaping them in R

In the previous sections we introduced several characters that have a special interpretation. This means that to recognize such characters simply as themselves, we have to do something extra, that is, we need to *escape* their special interpretation. As a reminder, we thus far encountered the following characters with a special interpretation.

```
        . [ ( * + ? | ^ \$
```

To interpret them as literal characters, in EREs, one can either precede them by a backslash or include them in a bracketed expression [ ] (but see the section on character ranges on how to properly include] and ^). So, for example, the expression a\.b would recognize "a.b" but not "aHb" and a\\b recognizes "a\b".

In R, however, there is a catch. Recall from Section 3.2.4 that in string literals the backslash already has a special meaning: it can be used to define characters using their Unicode code point. A double backslash indicates an actual single backslash, preventing the interpretation of "\u" as a command.

```
c("\uf6","\\uf6")
## [1] "ö"     "\\uf6"
```

Observe that R prints both backslashes at the command prompt. However, one may check, for example by exporting the string to a text file or by checking the string length with `nchar`, that the actual string contains a single backslash. For string literals that are to be interpreted as regular expressions, a double backslash must be included to indicate a single escape character.

```
# detect a.b, not a<wildcard>b
grepl("a\\.b", c("a.b", "aHb"))
## [1]  TRUE FALSE
```

To recognize a single backslash, a double backslash should be used in an ERE. In R, this means using two escaped backslashes so as to recognize the string `"a\b"`, and we can do the following:

```
# recognize "a\b" and not "ab"
grepl("a\\\\b", c("a\\b","ab"))
## [1]  TRUE FALSE
```

Such constructions quickly become unreadable as regular expressions get larger. A better alternative is to use bracketed expressions as follows:

```
grepl("a[\\]b", c("a\\b","ab"))
## [1]  TRUE FALSE
```

This does not reduce the length of an expression, but it does enhance readability. More importantly, the latter statement is compatible with extended regular expressions according to the Single Unix Specification.

Finally, we note that R has an option to escape regular expressions altogether. Functions like `sub`, `gsub`, `grep`, and `grepl` have an option called `fixed`. This option allows one to interpret the search pattern as is and not as a regular expression.

```
x <- c("a.b", "aHb", "a.b.*")
grepl("a.b.*", x)
## [1] TRUE TRUE TRUE
grepl("a.b.*", x, fixed=TRUE)
## [1] FALSE FALSE  TRUE
```

Being Lazy, Being Greedy

When a regular expression operator contains one or more of the repetition operators `+` or `*`, it is possible for a string to contain matches of varying length. For example, suppose we wish to remove the HTML tags in the following string.

```
The <em>hello world</em> programme.
```

As a first guess, we might attempt to use the pattern `<.*>` to find the tags. The problem is that there are several substrings matching this pattern: `"<em>"`, `"</em>"` and `"<em>hello world</em>"`. If one is only interested in whether a substring occurs or not, this is not a problem. It does become a problem when specific substrings need to be replaced or removed.

For this reason, EREs include a syntax to specify whether the shortest (lazy) or the longest (greedy) match should be used. The default behavior is to use the greedy match. Here is a simple demonstration using R's `gsub` function, which replaces every occurrence of a match with a replacement string.

```
s <- "The <em>hello world</em> programme"
gsub(pattern="<.*>", replacement="", x=s)
## [1] "The  programme"
```

Here, we replace each occurrence of the pattern `<.*>` with the empty string `""`. Since the default behavior is to be greedy, the pattern matches everything from the first occurrence of `"<"` to the last occurrence of `">"`. The question mark operator introduced in Table 5.2 switches the behavior to lazy matching.

```
gsub(pattern="<.*?>", replacement="", x=s)
## [1] "The hello world programme"
```

The pattern `<.*?>` translates to: first a `"<"`, followed by zero or more characters, to end with `">"`. Since we use `gsub` (the `g` stands for *global*), every such occurrence is replaced.

Regular Expressions with Memory: Groups and Back Referencing

We noted in Section 5.2.1 that regular expressions are limited since they lack a sense of memory and this is true for 'pure' regular expressions. In EREs, this restriction is relieved by introducing the concept of *groups*. The idea is that the strings that match a bracketed tag are stored for later reference. For example, to replace the string

```
The <em>hello world</em> programme
```

with

```
The <b>hello world</b> programme
```

we can devise a regular expression that stores everything between the `<em>` and `</em>` tags and places it between `<b>` and `</b>` tags.

```
 s <- "the <em>hello world</em> programme"
 gsub("<em>(.*?)</em>", "<b>\\1</b>", s)
## [1] "the <b>hello world</b> programme"
```

Here, the `\\1` refers back to the first bracketed expressions. Most regular expression engines, including R's, allow for back referencing up to nine grouped expressions in this way: `\1`, `\2`, ..., `\9` (recall that R requires the double backslash, since the first gets processed by R prior to passing it to the regular expression engine).

A second use of groups and references is to use the reference within the search pattern. For example, to match any HTML tag that is closed properly, one can use the expression `<(.*)?>.*</\\1>`. Here, the first group, defined by `(.*?)`, is referenced in the closing tag.

```
grepl("<(.*?)>.*?</\\1>", c("<li> hello","<em>hello world</em>"))
## [1] FALSE  TRUE
```

Finally, we note that some popular regular expression formats like the Perl Compatible Regular Expressions PCRE (2015) support named groups and back references. In R, support for Perl-like syntax is available through the `perl=TRUE` option.

```
grepl("<(?P<tag>.*?)>.*?</(?P=tag)>"
  , c("<li> hello","<em>hello world</em>")
  , perl=TRUE)
## [1] FALSE  TRUE
```

Here, we use the syntax `(?:<[groupname]>[pattern])` to define a named group. The syntax `(?P[groupname])` is used for back reference. Support for named groups is somewhat limited, since group names cannot be used in the replacement string of (g)`sub`, even when using the `perl=TRUE` switch.

Since any modern (read: extended) regular expression engine interprets a bracketed expression as a group to be captured, one might wonder if this capturing may be switched off. This is indeed the case, and the syntax for that is `(?:[pattern])`. One use of this is to avoid adding to the back-reference counter for groups that are not reused.

A Few Tips on Building and Maintaining Regular Expressions

Regular expressions are powerful tools that allow for fast text processing in a compactly defined way. In practice, regular expressions have the property of being fairly easy to build up, but they are typically also hard to decypher and debug.

When building regular expressions, it is a good idea to work step by step, setting up a simple expression, trying it out, check the cases you missed, and expand it further. When expressions get too long to read comfortably, it may be a good idea to split your job in two, applying two filters consecutively. In any case, it is a good idea to spend a few lines of comment explaining what you are trying to accomplish with a regular expression.

For many well-known patterns, such as zip codes, e-mail addresses or social security numbers, regular expressions may well already be around. There are many websites that list regular expression solutions to common pattern matching problems. We strongly advise to test regular expressions against examples that should, and should not match.

Finally, we emphasize that regular expressions should not be regarded as the answer to all text recognition problems. It was noted before that regular expressions have their limitations. For example, regardless of the example we have used, validating all possible e-mail addresses conforming to the standard that defines valid notation is notoriously hard to do with regular expressions (see, e.g., Friedl (2006)). Another example would be a regular expression that recognizes valid IPv4 addresses. Such addresses have the form $u.v.w.t$ where u, v, w, and t are numbers between 0 and 255. Since the possible allowed value of the second and third digit depends on the value of the first (099 is valid but 299 is not), regular expressions are not a logical choice. As an alternative, one could split the string along the periods, convert each substring to numbers, and check their ranges.

Exercises for Section 5.2.2

Exercise 5.2.4 *Dutch zip codes consist of four digits followed by two letters. Incrementally build a regular expression that recognizes cases of the following form.*

a) `1234AB` *and* `1234 AB`
b) The above and `1234-AB`
c) The above and the letters need not be capitals
d) Strings that are precisely zip codes. That is, no text comes before or after the zip code.

Exercise 5.2.5 *Extend the regular expression of Exercise 5.2.4 to recognize zipcodes where one of the letters is missing.*

Exercise 5.2.6 *Explain in words which patterns are recognized by the following regular expression.*

```
[0-9]+([.,][0-9]+)?([eE][0-9]+)?
```

Exercise 5.2.7 *In R, build a regular expression that recognizes strings starting with* `"\\mynetwork\"` *(i.e., a Windows-like network name).*

5.2.3 Generating Regular Expressions in R

The `rex` package of Ushey and Hester (2014) offers tools to generate regular expressions from a human readable syntax. The idea is to replace regular expression operators and character ranges by readable functions or keywords from which the actual regular expression is then derived. For example, the regular expression `(a|b)c*` can also be generated with the following statement.

```
rex::rex( "a" %or% "b", zero_or_more("c"))
## (?:a|b)(?:c)*
```

Observe that `rex` by default generates non-capturing groups (denoted by `(?:[regex])`.

The `rex` function takes a variable sequence of arguments, where each subsequent argument describes a new part of the pattern to be matched. Such a structure makes it much easier to alter and debug a regular expression. Moreover, since R expressions may be spread over several lines, it is much easier to comment separate parts of a pattern. The output is an object of class `regex` that holds a (generalized) regular expression that can be used with any function accepting regular expressions such as `sub` or `grep`. A `regex` object is like a string literal, but it differs in how it is printed to screen. Most notably, in `regexp` objects, the extra backslashes required by R are not printed to screen.

```
# look for . or ,:
r <- rex::rex("." %or% ",")
# the double backslash necessary in R (\\.) is not printed.
r
## (?:\.|,)
# however, using str, we see that it is actually there
str(r)
## Class 'regex'  chr "(?:\\.|,)"
```

Let us look at a more extended example. Suppose we want to extract numbers from strings. The numbers may be integers, they may be written in floating point notation with either period or comma separator (e.g., 34.05 or 34, 05), or in scientific notation, with `E` or `e` separating the mantissa from the exponent. With `rex`, we may specify this as follows. We only look for numbers that are surrounded by blanks.

```
r <- rex::rex(blank
  , one_or_more(digit)
  , maybe( "." %or% ",", one_or_more(digit))
```

```
    , maybe( one_of("e","E"), one_or_more(digit))
    , blank
  )
```

This statement is so obviously readable that it is hardly necessary to provide comment. Just pronouncing the arguments of rex, replacing every comma by 'followed by' gives a good impression of pattern being matched. Compare this with the regular expression that is produced (we break it into two lines here for presentation).

```
[[:blank:]](?:[[:digit:]])+(?:(?:.|,)
  (?:[[:digit:]])+)?(?:[eE](?:[[:digit:]])+)?[[:blank:]]
```

This example also shows a few of the operators offered by rex. The function maybe replaces the ? operator, the comma stands for concatenation, one_of stands for a bracketed range [ ], and one_or_more replaces stands for the + operator. The full list of range functions can be found by typing

```
?rex::character_class
```

at the command line. We already saw the %or% operator. The following operators are supported as well.

```
%if_next_is%     %if_prev_is%
%if_next_isnt%   %if_prev_isnt%
```

Again, the names are so obvious that they hardly require an explanation. These operators are actually part of the generalized regular expression syntax and have no equivalent operator in the regular expressions discussed until now.

Standard character ranges like those in Table 5.1 are supported as well. The complete list is quite extensive and can be found by typing

```
?rex::shortcuts
```

at the command line. In this example, we used blank ([:blank:]) and digit ([:digit:]).

The string literals provided to rex are interpreted as is. It is also possible to reuse previously defined regular expressions with arguments passed to rex, using the rex function. For example, the abovementioned extended example could also have been written as follows:

```
nr <- rex::rex(one_or_more(digit))
Ee <- rex::rex(one_of("E","e"))
r <- rex::rex(blank, nr, maybe("." %or% "," , nr), maybe(Ee,nr), blank)
```

5.3 Common String Processing Tasks in R

Among the most basic tasks in text string processing are pattern matching (detection of substrings) and localization, pattern replacement, pattern extraction, and string splitting. All of these are commonly performed with tools based on regular expressions or extensions thereof.

The traditional text processing tools in R are functions like (g)sub, grep(l), substr, regex, and strsplit. These functions combined, one can perform any of the

Table 5.3 A selection of basic text processing functions in base R and in `stringi`.

Base R	`stringi`	Description
`sub`	`stri_replace_first`	Substite the first matching instance of a pattern
	`stri_replace_last`	Substite the last matching instance of a pattern
`gsub`	`stri_replace_all`	Substitute all matching instances of a pattern
`grep`		Match strings against a pattern; returns index in vector
`grepl`	`stri_detect`	Match strings against a pattern; returns a logical vector
`regexpr`		Match strings against pattern; returns index in vector and location in strings
	`stri_match`	Match strings against a pattern; returns the first matched pattern. Also `str_match_all`, `_first`, `_last`
	`stri_locate`	Match strings against a pattern; return location of first match in string Also `stri_locate_all`, `_first`, `_last`
`substr`		Extract substring from `start` to `stop` index
	`stri_extract`	Extract first occurrence of a pattern from a string. Also `stri_extract_all`, `_first`, `_last`
`split`	`stri_split`	Split a string into substrings, using a pattern to define the separators
	`stri_reverse`	Reverse a string

basic tasks mentioned. However, the more recently published `stringi` package of Gagolewski and Tartanus (2014) makes many tasks, especially string extraction, a lot easier. The functionality of the package is richer than base R's and it uses a clear naming convention. Moreover, since it is based on the `ICU` library (ICU, 2014), it guarantees platform-independent results. At the time of writing, `stringi`'s popularity is increasing rapidly. For example, the latest versions of text processing tools like `stringr` (Wickham, 2010) and `qdapRegex` (Rinker, 2014) have switched to using `stringi` as backend for their higher level functions. Table 5.3 compares a selection of basic text processing functions from base R and `stringi`.

Replacing or Removing Substrings

The `qdapRegex` package exposes many functions (all starting with `rm_`) that allow one to remove common patterns. For example, `rm_number` removes numeric values from strings. It recognizes both integers and numbers with decimal separators.

```
qdapRegex::rm_number("Sometimes 12,5 is denoted 12.5 but we may
round it to 13")
## [1] "Sometimes 12,5 is denoted but we may round it to"
```

Scientific notation, for example, `1.25E1` is not recognized at the time of writing. The package contains similar functions for removal of e-mail addresses, urls, several date formats, twitter tags, and more.

The tm package includes the function removePunctuation, which removes occurrences of characters defined in the [:punct:] range (see Table 5.1). It has an option to prevent intra-word dashes from being removed.

```
x <- "'Is that an intra-word dash?', she asked"
tm::removePunctuation(x)
## [1] "Is that an intraword dash she asked"
tm::removePunctuation(x, preserve_intra_word_dashes=TRUE)
## [1] "Is that an intra-word dash she asked"
```

The tm package is focused on analyses of texts, which in this case means that many functions are overloaded to work on both character data and tm-defined objects that store text with some metadata.

The Regular Expression Interface of stringr

The stringr package has a utility function for removing blanks at the start and/or end of a string.[1]

```
library(stringr)
str_trim("  hello world ", side='both')
## [1] "hello world"
```

The side arguments control whether prepended and/or trailing white spaces are removed.

There are many options to replace substrings in R. The stringr function str_replace differs from R's default sub function in that it accepts a vector of regular expressions and a vector of replacements, where the shorter arguments are recycled to match the longest.

```
pat <- c("[[:digit:]]+","[^[:digit:]]+")
str_replace("Hello 12 34", pattern = pat, replacement = "")
## [1] "Hello  34" "12 34"
```

Here, we apply two regular expressions to the (single) input string and replace the matched patterns with the empty string (hence removing the matched substrings). In the first case, we remove sequences of at least one digit and in the second we precisely keep sequences of at least one digit. This function is a handy replacement for sub: it only replaces the first match. The package offers a similar replacement for gsub, which is called stringr::str_replace_all. For example, to collapse all double white spaces into a single whitespace, we could do the following:

```
pat <- "[[:blank:]]{2,}" # two or more blank characters
rep <- " "               # replace with single space
str_replace_all("Hello   many    spaces", pattern = pat, replace = " ")
## [1] "Hello many spaces"
```

Their consistent naming and vectorization over input strings, patterns, and replacements make stringr's functions more readable and versatile than base R functions.

[1] Strictly, it removes characters defined by the \s macro (this is Perl compliant). The documentation from which R's documentation is derived (http://linux.die.net/man/3/pcrepattern) is not entirely clear what symbols are included in this range. Presumably, the range is wider than just [:blank:], which is a subset of ASCII characters (Table 5.1).

Moreover, since they always have the data (here, the string) as the first input, it is possible to combine them in statements that include the `%>%` pipe operator of the `magrittr` package.

Like in base R, `stringr` allows one to escape the interpretation of a pattern as regular expression and just interpret it as is. This is done with the `fixed` function. Compare the following example with the example on page 94.

```
x <- c("a.b", "aHb", "a.b.*")
stringr::str_detect(x, "a.b.*")
## [1] TRUE TRUE TRUE
stringr::str_detect(x, stringr::fixed("a.b.*"))
## [1] FALSE FALSE  TRUE
```

Here, we use the `str_detect` function (a replacement for `grepl`), but any function in the `stringr` package that accepts a regular expression will obey the `fixed` directive. This function also allows for case-insensitive matching

```
x <- c("a.b", "aHb", "A.B.*")
str_detect(x, fixed("a.b.*"))
## [1] FALSE FALSE FALSE
str_detect(x, fixed("a.b.*", ignore_case=TRUE))
## [1] FALSE FALSE  TRUE
```

The `fixed` function causes its argument to be compared byte by byte, ignoring possible multibyte encoding. Since case-insensitive matching depends on case folding rules which are locale dependent, the `fixed` option may not always return the expected result. If you are sure that your matching pattern is in the same encoding as the string you are searching in, this is a safe option. The `coll` function makes sure that collation rules defined by the current or custom-defined locale are used. The following example from the documentation of `stringr` gives a clear example.

```
i <- c("I", "\u0130", "i")
i
## [1] "I" "İ" "i"
str_detect(i, fixed("i",ignore_case=TRUE))
## [1]  TRUE FALSE  TRUE
str_detect(i, coll("i", ignore_case=TRUE))
## [1]  TRUE FALSE  TRUE
str_detect(i, coll("i", ignore_case=TRUE, locale = "tr"))
## [1] FALSE  TRUE  TRUE
```

Here, the Turkish locale (`"tr"`) forces a different case folding, mapping `"I"` to the dotless `"ı"`. This can also be made visible with `stringr`'s case folding functions.

```
str_to_lower(i)
## [1] "i" "i?" "i"
str_to_lower(i, locale="tr")
## [1] "?" "i" "i"
```

Versatile Substitutions with `gsubfn`

A seemingly simple, but very powerful, generalization of `gsub` is implemented by the `gsubfn` package. It allows replacements of matched patterns by a (user-defined) *function* of those patterns. For example, in the given statements, the numbers in the strings `"(1,2)"` and `"x-23-y"` are multiplied by two.

```
gsubfn::gsubfn("[0-9]+", function(x) 2*as.numeric(x), c("(1,2)","x-23-y"))
## [1] "(2,4)"  "x-46-y"
```

Note that the result of the user-defined function is automatically converted to `character`.

Extracting or Splitting Substrings

The most convenient way of substring extraction in R is offered by the `stringr` package. Functions `str_extract` and `str_extract_all` return the first, respectively all matches of a regular expression.

```
x <- c("date: 11-11-2011", "Either 10-01-2001 or 07-03-2012")
stringr::str_extract(x,"[0-9]{2}-[0-9]{2}-[0-9]{2}")
## [1] "11-11-20" "10-01-20"
stringr::str_extract_all(x,"[0-9]{2}-[0-9]{2}-[0-9]{2}")
## [[1]]
## [1] "11-11-20"
##
## [[2]]
## [1] "10-01-20" "07-03-20"
```

Observe that `str_extract_all` returns a list, with one element for each entry in `x`. A nice feature of `str_extract_all` is the `simplify` option which returns the results, instead as a `character` matrix.

```
stringr::str_extract_all(x,"[0-9]{2}-[0-9]{2}-[0-9]{2}", simplify=TRUE)
##      [,1]       [,2]
## [1,] "11-11-20" ""
## [2,] "10-01-20" "07-03-20"
```

If it is known what positions in a string contain the information to extract, `substr` from base R can be used. For example, each IBAN account number starts with a two-letter (ISO 3166-1) country code which may be extracted.

```
iban <- "NLABNA987654321"
substr(start=1, stop=2,iban)
## [1] "NL"
```

It is possible to split a string at each match of a regular expression using `str_split`. The result is a vector with the original string cut in pieces and the matches left out.

```
x <- c("10-10 2010","12.12-2012")
stringr::str_split(x,"[-\\. ]")
## [[1]]
## [1] "10"   "10"   "2010"
##
## [[2]]
## [1] "12"   "12"   "2012"
```

Again, the result is a `list` with one entry for each element in `x`. There is also a variant called `str_split_fixed` that interprets the pattern as a literal string rather then a regular expression.

5.4 Approximate Text Matching

Approximate or 'fuzzy' text matching is in some ways complementary to the techniques discussed in the previous sections. Where character normalization and pattern extraction and substitution techniques are focused on bringing text closer to some standardized form, approximate text matching methods allow one to leave some slack between text and a standardized form.

As an example we consider the task of matching some input data against a list of standardized categories.

```
# standard categories
codes <- c("male", "female")
# vector of 'raw' input data
gender <- c("M", "male ", "Female", "fem.","female")
```

Here, we wish to assign each value in `gender` to a category in `codes`. If the entries in `gender` were perfect, R's native `match` function could be used.

```
# lookup of gender in the 'codes' lookup table
match(x = gender, table = codes)
## [1] NA NA NA NA  2
```

We get a single match. The last element of `gender` matches the second element of `codes`. We can do a little better by cleaning up `gender` a little.

```
# remove trailing whitespace, cast to lowercase
gender_clean <- stringi::stri_replace_all(gender,"",
regex="[[:blank:]]+$")
gender_clean <- stringi::stri_trans_tolower(gender_clean)
gender_clean
## [1] "m"      "male"   "female" "fem."   "female"
match(x = gender_clean, table = codes)
## [1] NA  1  2 NA  2
```

We now get three exact matches. The `amatch` function of the `stringdist` package (van der Loo, 2014) also matches strings, but it allows for a certain amount of slack between the data to match and entries in the lookup table.

```
stringdist::amatch(x = gender, table = codes, maxDist = 3)
## [1] NA  1  2  2  2
stringdist::amatch(x = gender_clean, table = codes, maxDist = 3)
## [1] 1 1 2 2 2
```

For raw data, four matches are found; while for the cleaned data we find a match for each data item in `gender`. The results are summarized in Figure 5.1.

In the call to `amatch`, we specified a maximum distance (`maxDist`) of three. In this case it means that for two elements to match, the optimal string alignment distance between them will be maximally 3. Roughly, this distance counts the number of allowed operations necessary to turn one string into another. The four operations include substitution, deletion, or insertion of a single character, and transposition of neighboring characters. For example, `"fem."` can be turned into `"female"` by substituting `"a"` for `"."` and inserting `"l"` and `"a"`, giving a total distance of three.

```
data.frame(
  gender        = gender
  , gender_clean = gender_clean
  , match        = codes[match(gender, codes)]
  , clean_match  = codes[match(gender_clean, codes)]
  , amatch       = codes[stringdist::amatch(gender,codes,maxDist=3)]
 , clean_amatch = codes[stringdist::amatch(gender_clean,codes,maxDist=3)]
)
##   gender gender_clean  match clean_match amatch clean_amatch
## 1      M            m   <NA>        <NA>   <NA>         male
## 2   male         male   <NA>        male   male         male
## 3 Female       female   <NA>      female female       female
## 4   fem.         fem.   <NA>        <NA> female       female
## 5 female       female female      female female       female
```

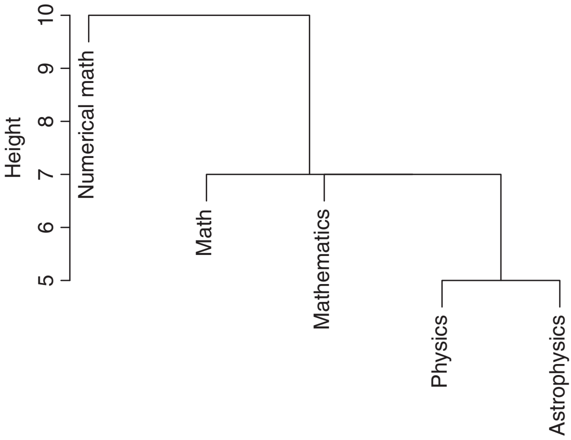
Figure 5.1 Results of exact or inexact matching against a lookup table with raw and normalized text data. The combination of normalization and approximate matching works best.

If no list of fixed codes is available, one may attempt to cluster strings with any method that accepts a distance matrix. For example, with R's hierarchical clustering implementation we can do the following:

```
# some 'raw' data to be clustered.
subject <- c("mathematics", "physics",
     "astrophysics", "math", "numerical math")
# compute distance matrix, adding strings as row/column names
M <-stringdist::stringdistmatrix(subject,subject, useNames=TRUE)
# Convert to 'dist' object and cluster hierarchically
h <- hclust(as.dist(M), method='single')
```

The resulting clustering is shown in Figure 5.2. In this case, the topics are clustered more or less as one would expect. The clustering is, of course, heavily dependent on both the chosen distance metric as well as the clustering method. For example, the same method gives unsatisfactory results for the `gender` example we introduced earlier. The category

Figure 5.2 Hierarchical clustering ('single' method) of five strings based on the optimal string alignment distance matrix.



names "male" and "female" are so close to each other than some of the polluted data items are to the right solution.

5.4.1 String Metrics

The purpose of a string metric is to introduce a sense of distance between elements of Σ^*. Formally, a distance function on Σ^* is defined as a function that assigns to each pair of strings a real number,

$$d : \Sigma^* \times \Sigma^* \mapsto \mathbb{R}. \tag{5.7}$$

Furthermore, a formal distance function satisfies the following rules:

$$\begin{aligned} &\text{nonnegativity} && d(s,t) \geq 0, && (5.8a)\\ &\text{identity} && d(s,t) = 0 \text{ if and only if } s = t, && (5.8b)\\ &\text{symmetry} && d(s,t) = d(t,s), && (5.8c)\\ &\text{triangle inequality} && d(s,u) \leq d(s,t) + d(t,u). && (5.8d) \end{aligned}$$

These rules are easily intuitively understood: the first rule states that distances cannot be negative; the second rule states that the distance between two strings is zero if and only if the two strings are the same; and the third rule states that taking a walk through Σ^* from s to t is as far as going from t to s. The fourth rule says that the value of the distance function is the value of the shortest route between two strings. Going from s to u via a third string t may not be shorter than going from s to u directly.

Over the years, many 'string metrics' have been developed. Perhaps surprisingly, not all of them fulfill the natural conditions laid down in Eq. (5.8). For example, in the following sections we show that q-gram-based distances do not always obey the identity property. That is, two strings may have distance zero between them while being different. Obviously, this means one needs to take care when choosing a distance function, for example, when clustering strings. Indeed, some of the 'metrics' we are going to discuss are not metrics in the mathematical sense. However, since it is common in literature to talk about string distance or string metrics regardless of which formal rules are satisfied, we shall apply the same convention here.

The principles from which metrics are derived vary strongly. We generally distinguish between edit-based distances, q-gram-based distances, heuristic distances, kernel-based distances, and distances based on phonetic algorithms. Here, we discuss these distances, pointing out their most important properties, including which, if any, of the rules in Eq. (5.8) are violated. We should state that there are more distance metrics than discussed in this book. Since we focus on text cleaning in this chapter, we focus on metrics that are commonly used in that area.

Edit-Based Distances

These distances are based on determining a sequence of basic transformations necessary to turn one string into another. The set of basic operations ('edits') includes one or more of the following:

- Deletion of a single character, for example, "foobar" → "fooar";
- Insertion of a single character, for example, "foobar" → "floobar";
- Substitution of a single character, for example, "foobar" → "foobaz";
- Transposition of two adjacent characters, for example, "foobar" → "foboar".

A distance function is defined by determining the shortest sequence of basic operations that turns one string into another and determine the length of the sequence. Alternatively, one may assign weights specific to each type of transformation and determine the sequence of least total weight. Such distances are referred to as *generalized* distances (Boytsov, 2011).

First note that when a too limited set of basic operations is chosen, not every pair of strings from Σ^* is connected by a sequence of allowed operations. Formally, such functions are not total functions on $\Sigma^* \times \Sigma^*$. We will follow the same convention as Navarro (2001) and define the distance between such unconnected pairs to be infinite. The advantage of this convention is that numerical comparisons ($<, \leq, =$) are then valid between all computed distances.

It can be shown that unweighted edit-based distances all obey the properties of a proper distance (Eq. 5.8), see, for example, Boytsov (2011). When different weights are assigned to insertion and deletion, distances become asymmetric: walking from `"abc"` to `"ac"` requires an insertion while walking from `"ac"` to `"abc"` requires an deletion. Distance functions with weighted insertion or deletion are symmetric under simultaneous swapping of both the input strings and deletion and insertion weights.

Table 5.4 gives an overview of commonly used edit-based string distances and their allowed operations. The distance of Hamming (1950) is the simplest and allows only for substitution of characters. Hence, it is finite only when computed between strings of the same length. For example, we have $d_{\text{hamming}}(\texttt{"fu"}, \texttt{"fo"}) = 1$ and $d_{\text{hamming}}(\texttt{"fu"}, \texttt{"foo"}) = \infty$. Since there is only one allowed operation, there is no use in defining weights. Since it is defined only on strings of equal lengths, in the context of text processing, the Hamming distance is most useful for standardized strings, such as phone numbers, product codes, or other identifying numbers.

The *longest common subsequence distance* (lcs distance) of Needleman and Wunsch (1970) is the simplest generalization of the Hamming distance and it replaces the substitution operation by allowing insertion and deletion of characters. This means that the distance function is a complete function on $\Sigma^* \times \Sigma^*$. In fact, it is easy to see that given two strings s and t, their distance will be largest if they have no characters in common. In that case, the distance from s to t is simply $|s| + |t|$, the cost of deleting one by one all characters from s and then inserting all characters of t. As an example, consider the distance d_{lcs} between `"fu"` and `"foo"`. Using only deletion and insertion, it takes three steps to go from one to the other.

Table 5.4 Edit-based string distances.

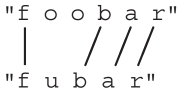
Distance	Operation			
	del	ins	sub	trn
Hamming	−	−	+	−
Longest common substring	+	+	−	−
Levenshtein	+	+	+	−
Optimal string alignment	+	+	+	+a)
Damerau–Levenshtein	+	+	+	+

a) Each character can take part in maximally a single transposition.

$$\texttt{"fu"} \xrightarrow{-\texttt{"u"}} \texttt{"f"} \xrightarrow{+\texttt{"o"}} \texttt{"fo"} \xrightarrow{+\texttt{"o"}} \texttt{"foo"}$$

This also shows that there is not necessarily a unique shortest path from the start point to the end. Other than in Euclidean geometry, for example, we could have chosen a different path through Σ^* (e.g., `"fu"` → `"fuo"` → `"fo"` → `"foo"`) and obtained a path of the same length.

As implied by its name, the lcs distance has a second interpretation. Given two strings, we pair common characters, passing through the strings from left to right but without changing their order. The distance between the two strings is then given by the number of unpaired characters in both strings. The diagram given here demonstrates this process.



Indeed, there are three unpaired characters in the two strings. Note that a subsequence is not the same as a substring. If $s = s_1 s_2 \cdots s_n$ is a string, a substring indicates a consecutive sequence $s_k s_{k+1} \cdots s_{k+l}$. A *subsequence* is a sequence of characters $s_{i(1)} s_{i(2)} \cdots s_{i(l)}$ that preserves order ($i(1) < i(2) < \cdots < i(l)$) but need not be consecutive. Hence, `"fbar"` is a subsequence of both `"fubar"` and `"foobar"` but is not a substring. Applications of the lcs distance lie mainly outside of data cleaning, although it can be interpreted as a crude (and fast) approach for typo-detection. In principle it is possible to define a generalized lcs distance by assigning different weights to insertion and deletion but this seems not very common in practice.

The *Levenshtein distance* (Levenshtein, 1966) is like the longest common substring distance but it is more flexible since it also allows for direct substitution of a character, which would take two steps in the lcs distance. This extra flexibility leads to shorter minimal distances. For example, the Levenshtein distance between `"fu"` and `"foo"` equals two:

$$\texttt{"fu"} \xrightarrow{\texttt{"u"}\to\texttt{"o"}} \texttt{"fo"} \xrightarrow{+\texttt{"o"}} \texttt{"foo"}. \tag{5.9}$$

The generalized Levenshtein distance satisfies the properties of Eq. (5.8) except when the weights for insertion and deletion are different. In that case, the distance is no longer symmetric, which is easily illustrated with an example. Suppose we weigh an insertion with 0.1 and deletion and substitution with 1. We have (denoting the cost per step underneath the arrows)

$$\texttt{"fu"} \xrightarrow[+1]{\texttt{"u"}\to\texttt{"o"}} \texttt{"fo"} \xrightarrow[+0.1]{+\texttt{"o"}} \texttt{"foo"},$$

giving a distance of 1.1. Going the other way around, however, we get

$$\texttt{"foo"} \xrightarrow[+1]{-\texttt{"o"}} \texttt{"fo"} \xrightarrow[+1]{\texttt{"o"}\to\texttt{"u"}} \texttt{"fu"},$$

giving a distance of 2.

The *optimal string alignment distance*, which is also referred to as the restricted Damerau–Levenshtein distance, allows for insertions, deletions, substitutions, and limited transpositions of adjacent characters. The limitation lies therein that each substring may be edited only once.

The most important consequence is that optimal string alignment distance does not satisfy the triangle inequality. The following counterexample is taken from Boytsov (2011). If we take $s =$ `"ab"`, $t =$ `"ba"` and $u =$ `"acb"`, the distance $d_{\text{osa}}(s,t) + d_{\text{osa}}(t,u)$ is given by

$$\texttt{"ba"} \xrightarrow[+1]{\texttt{"b"}\leftrightarrow\texttt{"a"}} \texttt{"ab"} + \texttt{"ba"} \xrightarrow[+1]{+\texttt{"c"}} \texttt{"bca"},$$

giving a total distance of 2. The intuitive direct path from `"ab"` to `"bca"`

$$\texttt{"ab"} \xrightarrow[+1]{\texttt{"b"}\leftrightarrow\texttt{"a"}} \texttt{"ba"} \xrightarrow[+1]{+\texttt{"c"}} \texttt{"bca"}. \tag{5.10}$$

However, this involves editing the subsequence `"ba"` twice: once by transposing `"a"` and `"b"` and once by inserting a `"c"` between `"a"` and `"b"`. In fact, it turns out that there is no way to go from `"ba"` to `"acb"` directly in less than three steps, and so one has to resort to the sequence

$$\texttt{"ba"} \xrightarrow{-\texttt{"a"}} \texttt{"b"} \xrightarrow{+\texttt{"c"}} \texttt{"bc"} \xrightarrow{+\texttt{"a"}} \texttt{"bca"}.$$

The optimal string alignment distance between a string variable and a known fixed string may be interpreted as the number of typing errors made in the string variable.

The *Damerau–Levenshtein* distance (Lowrance and Wagner, 1975) does allow for arbitrary transpositions of characters. That is, subsequences may be edited more than once, allowing the path of Eq. (5.10).

The maximum distance between two strings s and t, as measured by any of the discussed distances that allows for character substitution, equals $\max\{|s|, |t|\}$. Distances that allow for a larger number of basic operations generally yield numerically smaller distances. The given diagram summarizes the relation between edit-based distances and their limits.

$$\text{Dmr-Levensht.} \leq \text{Opt. Str. Align.} \leq \text{Levensht.} \leq \begin{cases} \text{Hamming} \leq \infty, \\ \text{LCS} \leq |s| + |t|, \\ \max\{|s|, |t|\}. \end{cases} \tag{5.11}$$

Note that the maxima may be used to scale distances between 0 (exactly equal) and 1 (completely different) should a method based on a string metric require it.

Finally, some notes on the computational time it takes to compute edit-based distance. The Hamming distance can obviously be computed in $\mathcal{O}(\min\{|s|, |t|\})$ time. The other edit-based distances are usually implemented using dynamic programming techniques that run in $\mathcal{O}(|s||t|)$ time. See Navarro (2001) or Boytsov (2011) for an overview.

The Jaro–Winkler Distance

The *Jaro distance* was originally developed at the US bureau of the Census as part of an application called UNIMATCH. The purpose of this program was to match records based on inexact text fields, mostly name and address data. The first description appeared in the manual of this software (Jaro, 1978), which may be a reason why it is not widely described in computer science literature. Jaro (1989), however, reports successful application of this distance for matching inexact name and address fields.

The Jaro distance is an expression based on the intuition that character mismatches and character transpositions are caused by human-typed errors and that transposition of characters that are further apart are less likely to occur. It can be expressed as follows:

$$d_{\text{Jaro}}(s,t) = \begin{cases} 0 \text{ if } s = t = \epsilon, \\ 1 \text{ if } |s| + |t| > 0 \text{ and } m = 0, \\ 1 - \frac{1}{3}\left(w_1\frac{m}{|s|} + w_2\frac{m}{|t|} + w_3\frac{m-T}{m}\right) \text{ otherwise.} \end{cases} \tag{5.12}$$

Here, m is the number of matching characters and T the number of transpositions, counted in a way to be explained later and the w_i are nonnegative weight parameters which are usually chosen to be 1. The distance ranges between 0 (s and t are identical) and 1 (complete mismatch). It is zero if both s and t are the empty string and it equals 1 if there are no matches while at least one string is nonempty. Otherwise, the distance depends on a weighted sum of character matches and transpositions.

The number of matches is computed as follows. First, pair characters from s and t as you would for the longest common subsequence distance. It now depends on the distance between the positions of paired characters whether a pair is considered a match. In particular, if $s_i = t_j$ is a pair, this is considered a match only when

$$|i-j| < \left\lfloor \frac{\max\{|s|,|t|\}}{2} \right\rfloor, \tag{5.13}$$

where $\lfloor\cdot\rfloor$ indicates the floor function and each character can be included no more than one match. For example, $s =$ `"abcd"` and $t =$ `"acdb"` the number of matches $m = 3$.

The number of transpositions T is computed as follows. First take the subsequences s' and t' of s and t consisting only of the matching characters. Then T is the number of characters in t' that have to change position to turn t' into s'. For example, take $s =$ `"abcdef"` and $t =$ `"adcbef"`, we have $m = |s| = |t| = 6$ and $s' = s$ and $t' = t$. To turn t' into s', we need two operations:

$$\texttt{"adcbef"} \xrightarrow{\text{move } \texttt{"d"}} \texttt{"acbdef"} \xrightarrow{\text{move } \texttt{"b"}} \texttt{"abcdef"}.$$

The term *transposition* here is used differently from the definition in Section 5.4.1 where it is defined for transposition of two characters. Here, it should be interpreted as transposing a single character with a substring. For example, in the first step of the abovementioned example, we may interpret the relocation of `"d"` as swapping `"d"` with `"cb"`.

Unfortunately, there seems to be some confusion in literature over the precise definition of the Jaro distance. Cohen *et al.* (2003), for example, use $\lfloor\min\{|s|,|t|\}/2\rfloor$ as an upper limit for $|i-j|$. The definition used here, and which is also implemented in `stringdist` package, follows the original definition of Jaro (1978).

Winkler (1990) proposes a simple variant of the Jaro distance that rests on the observation that typing errors are less likely to occur at the beginning of a human-typed string than near the end. If differences occur at the beginning, the heuristic is that the strings are likely to be truly dissimilar. In the *Jaro–Winkler* distance, this phenomenon is taken advantage of by multiplying the Jaro distance with a penalty factor that depends on the number of unequal characters in the first four positions. The expression is as follows:

$$d_{jw}(s,t) = d_{\text{Jaro}}(s,t)[1 - p\ell(s,t)]. \tag{5.14}$$

Here, $\ell(s,t)$ is the length of the longest common prefix of s and t, up to a maximum of 4. The value of p is a user-defined adjustable parameter but it must be between 0 and $\frac{1}{4}$ to ensure $0 \leq d_{jw}(s,t) \leq 1$. The value of p determines how strong the length of the common prefix weighs into the total distance and the Jaro distance emerges as a special case when $p = 0$. Both Winkler (1990) and Cohen *et al.* (2003) report good results in matching name–address data when setting $p = 0.1$.

As far as this author knows, the properties as described in Eq. (5.8) are not reported anywhere except in van der Loo (2014). They are however easily derived, so we will state them in the following observation.

Proposition 5.4.1 *The Jaro distance with $w_1 = w_2 = w_3 = 1$ satisfies the properties of nonnegativity, symmetry, and identity, but not the triangle inequality.*

Proof: Note first that for the number transpositions T and the number of matches m we must have $0 \leq T \leq m \leq \min\{|s|, |t|\}$. Thus, all terms in Eq. (5.12) are nonnegative, and the equation is easily seen to be invariant under transposition of s and t.

For the identity property observe that if s=t, we have $m = |s| = |t|$ and $T = 0$, giving $d_{\text{Jaro}}(s,t) = 0$. To prove that the reverse is also true, first note from Eq. (5.12) that $d_{\text{Jaro}}(s,t) = 0$ if either $s = t = \epsilon$ or

$$\frac{m}{|s|} + \frac{m}{|t|} + \frac{m-T}{m} = 3.$$

Since $0 \leq m \leq \min\{|s|, |t|\}$ and $T \geq 0$, each term in this sum is nonnegative and the maximum value for the left-hand side is reached uniquely when $m = |s| = |t|$ and $T = 0$, giving $m/|s| + m/|t| + (m - T)/m = 1 + 1 + 1 = 3$. The fact that $m = |s| = |t|$ (all characters match) and $T = 0$ (all characters are on the same position) together imply that $s = t$.

The fact that d_{Jaro} does not satisfy the triangle inequality is easily demonstrated with a counterexample. Take $s =$ `"ab"`, $t =$ `"cb"` and $u =$ `"cd"`. It is a simple calculation to confirm that

$$d_{\text{Jaro}}(\texttt{"ab"}, \texttt{"cd"}) = 1 \nleq d_{\text{Jaro}}(\texttt{"ab"}, \texttt{"cb"}) + d_{\text{Jaro}}(\texttt{"cb"}, \texttt{"cd"}) = \frac{1}{3} + \frac{1}{3}$$

thus violating the triangle inequality. □

q-Gram-Based Distances

The edit-based distances and the Jaro–Winkler distance described are typically used for the detection of misspelling errors in fairly short strings. They are based on a precise description of what operations are allowed to turn one string into another and therefore have natural paths from s to t through Σ^* associated with them.

Distances based on q-grams, on the other hand, are computed by first deriving an integer vector (the so-called *q-gram profile*) from strings s and t and defining a distance measure on the space of vectors. Well-known examples include the Manhattan distance or the cosine distance.

A *q-gram* is a string of q characters. Given a string s ($|s| \geq q$), the associated *q-gram* profile is a table of size Σ^q counting the occurrence of all q-grams in s. For example, the nonzero entries of the 2-gram profile of `"banana"` looks like this:

`"ba"`	1
`"an"`	2
`"na"`	2.

In literature on text classification, the term n-gram is often used instead of q-gram. Since this term is also used to indicate word sequences rather than character sequences, we follow the terminology of Ukkonen (1992) and use the term q-gram to avoid confusion.

We will denote the q-gram profile of a string s as a vector-valued function $\boldsymbol{v}(s;q)$. The range of $\boldsymbol{v}$ is $\mathbb{N}^{|\Sigma^q|}$, the set of integer vectors whose size is the number of possible q-grams allowed by the alphabet. In general, $\boldsymbol{v}$ is not a complete function on Σ^*. In particular, it is undefined when $|q| > s$, and we also need to decide what to do when $q = 0$. Here, we adopt the following conventions:

$$\boldsymbol{v}(s;q) = \mathbf{0} \text{ if } q > |s|$$
$$\boldsymbol{v}(s,0) = \mathbf{0}.$$

Intuitively, this corresponds to the idea that all distances are zero as long as we are not comparing anything. In particular, it means that $d(\epsilon, \epsilon; 0) = 0$ for any q-gram-based distance.

The traditional q-gram distance is computed as the Manhattan distance between two q-gram profiles:

$$d_{q\text{gram}}(s,t;q) = \boldsymbol{v}(s;q) - \boldsymbol{v}(t;q)_1 = \sum_{i=1}^{|\Sigma^q|} |v_i(s;q) - v_i(t;q)|.$$

Observe that since the $v_i(s;q)$ are counts rather than frequencies, this distance is both an expression of difference in q-gram distribution as well as a difference in string lengths. It is not hard to show that the maximum q-gram distance between two strings s and t is equal to

$$|s| - \max\{0, q-1\} + |t| - \max\{0, q-1\}\text{and } q \geq 1.$$

To accommodate for the effect of differences in string length, one can base the distance on the (cosine of the) angle between the q-gram vectors.

$$d_{\mathsf{cos}}(s,t;q) = 1 - \cos(\boldsymbol{v}(s;q)\angle\boldsymbol{v}(t;q)) = 1 - \frac{\boldsymbol{v}(s;q)\cdot\boldsymbol{v}(t;q)}{\|\boldsymbol{v}(s;q)\|_2\|\boldsymbol{v}(t;q)\|_2}.$$

This distance varies between 0 and 1, where 0 means complete similarity and 1 means complete dissimilarity (no overlap in occurring q-grams).

A third way to compare q-gram profiles is the *Jaccard distance.*

$$d_{\mathsf{Jaccard}}(s,t;q) = 1 - \frac{q|\text{gram}(s,q) \cap q\text{gram}(t,q)|}{q|\text{gram}(s,q) \cup q\text{gram}(t,q)|},$$

where $q$gram(s;q) returns the set of all q-grams occurring in s. The Jaccard distance scales from 0, indicating complete similarity to 1, indicating complete dissimilarity.

The term 'complete similarity' is not the same as equality. We already saw the case for $q = 0$, which defines all strings as equal (there is nothing to make them different when $q = 0$). The fact that unequal strings have a q-gram-based distance equal to zero is not, however, limited to $q = 0$. For example, if $q = 1$, the table of q-gram counts is the same for two words if they are anagrams. Ukkonen (1992) has shown that one can derive transformations on strings of a certain structure that leave the q-gram profile intact, for any value of q. For example, the reader can check that the strings

`"aFOObBARa"` and `"bBARaFOOb"`

have the same 2-gram profile.

We have seen that q-gram distances do not obey the identity property of distance metrics. The other properties depend on the metric that is chosen over the positive integer vectors. The distances discussed here are all nonnegative, symmetric. Both the Jaccard and q-gram distance also satisfy the triangle inequality. The cosine distance does not satisfy the triangle inequality. This is easily seen by assuming three q-gram profiles, $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ with angles $\alpha = \boldsymbol{u}\angle\boldsymbol{v} = \boldsymbol{v}\angle\boldsymbol{w}$ and $\boldsymbol{u}\angle\boldsymbol{w} = 2\alpha$. We have $d(\boldsymbol{u},\boldsymbol{w}) = 1 - \cos(2\alpha) = 2 - 2\cos^2\alpha$ and $d(\boldsymbol{u},\boldsymbol{v}) + d(\boldsymbol{v},\boldsymbol{w}) = 2 - 2\cos(\alpha)$. To show that the cosine distance does not satisfy the triangle inequality, we only need to find an α for which $1 - \cos^2\alpha \nleq 1 - \cos\alpha$. This is the case for every $0 < \alpha < \pi/2$.

q-gram profiles have been widely used as a tool for text classification. Applications and methodology have been reported where texts have been classified according to the language used (Cavnar and Trenkle, 1994), topics under discussion (Clement and Sharp, 2003), or author style and identity (Kešelj, *et al.* 2003), to name but a very few examples. In many cases, the use of bi- or trigrams seems an appropriate choice for natural language classification (Fürnkranz, 1998).

Distances Based on String Kernels

Just like q-gram-based methods, string kernels are interesting for text categorization or clustering purposes.

A string kernel is defined by a map ϕ that assigns to each string s in Σ^* a real vector $\boldsymbol{\phi}(s)$ with nonnegative coefficients. The normalized kernel function $k(s,t)$ of two strings is defined as

$$k(s,t) = \frac{\boldsymbol{\phi}(s)\cdot\boldsymbol{\phi}(t)}{\|\boldsymbol{\phi}(s)\|_2\|\boldsymbol{\phi}(t)\|_2},$$

where '$\cdot$' is the standard inner product on a real vector space. The right-hand side is the cosine between the angle of the feature vectors $\boldsymbol{\phi}(s)$ and $\boldsymbol{\phi}(t)$, and thus a measure of distance can be obtained as

$$1 - k(s,t),$$

and a suitable map $\boldsymbol{\phi}$ defines a sense of distance on Σ^*.

Several authors have defined string kernels and tested their effectiveness. Karatzoglou and Feinerer (2007) propose two string kernels. In the first kernel (simply called the *string kernel*), each element $\boldsymbol{\phi}_u$ of $\boldsymbol{\phi}$ counts the number of occurrences of subsequences in s.

$$\phi_u^{\mathsf{sk}}(s;k) = \#\text{ of times subsequence } u \text{ occurs in } s \text{ and } |u| = k.$$

The second kernel, called the *full string kernel*, is defined by concatenating the feature vectors defined by every subsequence up to length k.

$$\boldsymbol{\phi}^{\mathrm{fsk}}(s;k) = \text{concatenate } \boldsymbol{\phi}^{\mathrm{sk}}(s) \text{ for all } u \text{ with } |u| < k.$$

Lodhi *et al.* (2002) propose a string kernel that penalizes contributions of each subsequence by the number of positions traversed going from the first to the last character of a subsequence u of s.

$$\phi'_u(s;\lambda,k) = \sum_{\boldsymbol{i}:s_{\boldsymbol{i}}=u} \lambda^{\ell(\boldsymbol{i})}, \text{ with } |u| = k \text{ and } \lambda \in (0,1), \tag{5.15}$$

where the sum is over every occurrence of the subsequence u in s, and $\ell(\boldsymbol{i})$ is the length of the subsequence u defined by the index vector $\boldsymbol{i}$ in s: $\ell(\boldsymbol{i}) = i_{|u|} - i_1 + 1$.

Phonetic Algorithms

Phonetic algorithms aim to match strings that sound similar when they are pronounced. Phonetic algorithms transform a string to some kind of phonetic string that represents pronunciation features. The distance between two strings is then defined as a distance between the phonetic strings. In the simplest case, the distance is 0 when two phonetic strings are equal and 1 otherwise.

One of the most famous phonetic algorithms, called soundex, was patented for the first time in the early twentieth century (Russell, 1918, 1922). It is aimed at names, written in the ASCII alphabet, pronounced in English. One widely used version of the algorithm is published by the United States National Archives and Records Administration (NARA, 2015).

The soundex algorithm encodes a string to a phonetic string consisting of a letter and three numbers. The algorithm can be summarized as follows:

Procedure 5.4.1 soundex

Input A string s in $\{$`"a"`, `"b"`, ..., `"z"`$\}^*$.

1 Take the first character of s, in upper case.

2 Remove all characters in $\{$`"a"`, `"e"`, `"i"`, `"o"`, `"u"`, `"h"`, `"w"`, `"y"`$\}$.

3 Replace the remaining characters according to the following scheme.

$$\begin{aligned}
\{\texttt{"b"},\texttt{"f"},\texttt{"p"},\texttt{"v"}\} &\mapsto 1\\
\{\texttt{"c"},\texttt{"g"},\texttt{"j"},\texttt{"k"},\texttt{"q"},\texttt{"s"},\texttt{"x"},\texttt{"z"}\} &\mapsto 2\\
\{\texttt{"d"},\texttt{"t"}\} &\mapsto 3\\
\{\texttt{"l"},\} &\mapsto 4\\
\{\texttt{"m"},\texttt{"n"}\} &\mapsto 5\\
\{\texttt{"r"}\} &\mapsto 6.
\end{aligned}$$

The following cases need special treatment in this replacement. If two characters that are encoded with the same number are either adjacent or separated by a `"h"` or `"w"`, only the first character is encoded. The other is discarded.

4 If the resulting code has less then three numbers, pad the result with zeros so that the result is four characters long.

Output A four-character soundex code.

For example, the names "van der loo" and de jonge map to "V536" and "D252", respectively, (after removing spaces). Phonetically similar names map to similar results. For example, "john" and "joan" both map to "J500", and similarly "ashley" and "ashly" map to "A240".

The soundex algorithm has been adapted and expanded over the years by many researchers and practitioners. Like the original algorithm, such extensions are often strongly related to a particular purpose. For example, the Metaphone algorithms by Philips (2000) started as improvement on the soundex algorithm for better matching of English names, but it has been extended to other languages including Spanish, and Brazilian Portuguese. Holmes and McCabe (2002) focus on the original purpose of English name matching, using a mix of different phonetic techniques. Mokotov (1997), and later Beider and Morse (2008) developed alternatives aimed at matching Eastern European and Ashkenazic Jewish surnames, respectively. Finally, Pinto *et al.* (2012) report on a soundex-like algorithm for strings typically occurring in short text messages (sms).

5.4.2 String Metrics and Approximate Text Matching in R

Base R comes with a few basic capabilities for approximate text matching. The function adist (of the standard utils package) computes the generalized Levenshtein distance. By default, the weights for insertion, deletion, and substitution are all equal to one, but this may be set with the cost argument.

```
adist("fu","foo")
##      [,1]
## [1,]    2
adist("fu","foo", cost=c(i=0.1, d=1, s=1))
##      [,1]
## [1,]  1.1
```

It is obligated to name the elements of cost so it (partially) matches insertion, deletion, and substitution. If adist is provided with vector(s) of strings, the distance matrix between the strings is computed.

```
adist(c("hello","world"),c("olá mundo"))
##      [,1]
## [1,]    8
## [2,]    8
```

The agrep function is based on the work of Laurikari (2001) who combines the idea of approximate matching with regular expression search. Recall that a regular expression defines a subset of all possible strings that can be built up out of a certain alphabet. The function grep takes a regular expression and a string x and determines whether any substring of x is in the set defined by the regular expression. The function agrep is even more flexible and searches whether there is a substring in x that is maximally some Levenshtein distance away from an element in the set defined by the regular expression.

```
# a pattern to search for
re <- "abc"
# two strings to search in
x <- c("FOOabcBAR","FOOaXcBAR")
```

```
# grep finds only one match (in the first element)
grep(pattern=re, x=x)
## [1] 1
# agrep finds both
agrep(pattern=re, x=x, max.distance=1)
## [1] 1 2
```

In this simple example, the set of strings matched by the regular expression is precisely {"abc"}. Thus, grep returns 1, the index into x where a match is found. Int The argument max.distance tells agrep to search for strings that are maximally Levensthein distance 1 removed from the substrings defined by the pattern "abc". Therefore, agrep also matches the substring "aXc" since it is a single substitution away from "abc".

The concept behind agrep is extremely powerful and elegant. However, do note that it can be hard to visualize what strings are defined by a regular expression. Combining complex regular expressions with the flexibility of agrep may yield unexpected matches. So, just like when working with grep and related tools, it is strongly recommended to build up regular expressions piece by piece, and to create test sets of strings with cases that you expect to match (or not).

Beyond the Levenshtein Distance with stringdist

The stringdist package offers a number of string metrics which makes it easy to try and test several distance measures for your application. Metrics can be computed using the stringdist function, where the default function is the optimal string alignment distance.

```
library(stringdist)
stringdist("hello",c("hallo","wereld"))
## [1] 1 4
```

Here, the shortest argument ("hello") is recycled and the result is a vector of length 2, measuring the distance between "hello" and "hallo" and "hello" and "wereld". So unlike adist, this function does not return a matrix. A matrix of distances can be computed with the stringdistmatrix function.

```
stringdistmatrix("hello",c("hallo","wereld"), useNames="strings")
##       hallo wereld
## hello     1      4
```

To perform approximate matching, the amatch function can be used. Here, amatch is similar to R's match function. It is given some strings x and a lookup table table and returns an index in table for each element of x that can be matched. The difference is that amatch allows for a certain string distance.

```
match("hello",c("hallo", "wereld"), nomatch=0)
## [1] 0
amatch("hello",c("hallo", "wereld"), nomatch=0, maxDist=1)
## [1] 1
```

Here, match finds no match for "hello" in the vector c("hello","wereld"), while amatch matches "hello" with "hallo". The strings "hello" and "hallo" are one substitution away from each other, which is equal to the maximally allowed distance.

In R, there is a variant of match called %in%, which returns a logical vector that indicates which elements of x occur in the lookup table. The stringdist package provides an approximate matching equivalent called ain.

```
"hello" %in% c("hallo", "wereld")
## [1] FALSE
ain("hello",c("hallo", "wereld"), maxDist=1)
## [1] TRUE
```

Each function from the stringdist package discussed (stringdist, stringdistmatrix, amatch, and ain) have a method argument that allows one to switch between string metrics.

```
stringdist("hello",c("hallo","wereld"),method="hamming")
## [1]   1 Inf
amatch("hello",c("hallo", "wereld"), nomatch=0,method="hamming",
maxDist=1)
## [1] 1
```

Table 5.5 gives an overview of the available distance metrics.

Depending on the chosen distance, there are some options to set. The edit-based distances (Levenshtein, optimal string alignment, and Damerau–Levenshtein) are implemented in the generalized form, so weights for different edits may be specified in the order deletion, insertion, substitution, and transposition (if applicable).

```
stringdist("hello","hallo", method="osa", weight=c(1,1,0.1,1))
## [1] 0.1
```

For the Jaro–Winkler distance, there are weights for matched characters in the first string, matched characters in the second string, and a weight for the contribution of the transpositions [see Eq. (5.12)]. These can be passed through the same weight argument. If method is set to "jw", Eq. (5.14) is used to compute the Jaro–Winkler distance where the default value of Winkler's penalty factor p is set to 0. That is, the pure Jaro distance is computed by default. This can be controlled with the parameter p.

Table 5.5 Methods supported by the stringdist package.

method	**Distance**
"hamming"	Hamming distance
"lcs"	Longest common subsequence
"lv"	Generalized Levenshtein
"dl"	Generalized Damerau–Levenshtein
"osa"	Generalized optimal string alignment (default)
"jw"	Jaro, Jaro–Winkler distance
"qgram"	*q*-gram distance
"cosine"	Cosine distance (based on q-gram profiles)
"jaccard"	Jaccard distance (based on*q*-gram profiles)
"soundex"	Soundex distance

The term 'generalized' indicates that edit operations can be weighted.

```
# Compute the Jaro distance
stringdist("marhta", "martha", method="jw")
## [1] 0.05555556
# Compute the Jaro-Winkler distance with p=0.1
stringdist("marhta", "martha", method="jw", p=0.1)
## [1] 0.03888889
```

Like the `method` parameter, these extra options are available for `amatch`, `ain`, and `stringdistmatrix` as well. So to do an approximate lookup using the Jaro–Winkler distance with $p = 0.1$, one simply does the following.

```
# raw data
x <- c("Stan Marhs","Kyle Brovlofski")
# lookup table
sp_char <- c("Kenny McCormick","Kyle Broflovski","Stan Marsh",
"Eric Cartman")
# find matches
amatch(x=x, table=sp_char, method="jw", p=0.1, maxDist=0.2)
## [1] 3 2
```

For distances based on q-grams there is an optional parameter `q`, which by default is set to `1`.

```
amatch(x, table=sp_char, method="qgram", q=2, maxDist=5)
## [1] 3 2
stringdistmatrix(x, sp_char, method="qgram", q=2, useNames="strings")
##                 Kenny McCormick Kyle Broflovski Stan Marsh Eric Cartman
## Stan Marhs                   21              23          4           16
## Kyle Brovlofski              28               4         23           25
```

Finally, we mention that soundex codes can be computed with the `phonetic` function.

```
phonetic(sp_char)
## [1] "K552" "K416" "S356" "E626"
```

In `stringdist`, the soundex distance between two strings is equal to 0 if they have the same soundex code and 1 otherwise.

```
stringdistmatrix(x, sp_char, method="soundex", useNames="strings")
##                 Kenny McCormick Kyle Broflovski Stan Marsh Eric Cartman
## Stan Marhs                    1               1          0            1
## Kyle Brovlofski               1               0          1            1
```

String Similarity

For some applications, it can be more convenient to think in terms of string similarity than string distance. For any given distance measure, a similarity measure can be defined, but the procedure to do so depends on the distance used. In particular, it depends on the range of the distance function, and we distinguish two cases.

The `stringdist` package implements a function that maps any string distance between two strings to a value in $[0, 1]$, where 0 means complete dissimilarity and 1 means complete similarity. Note that for distances depending on q-grams, complete similarity does not necessarily mean equality since those distances do not satisfy the identity property of Eq. (5.8). If $d(s,t)$ is the distance between two strings, the string similarity is defined as

$$1 - \frac{d(s,t)}{\max\{d(s',t') : |s'| = |s|, |t'| = |t|\}} \text{if } d(s,t) < \infty \text{ else } 0. \tag{5.16}$$

That is, the distance is divided by the maximum possible distance between two strings of the same length as the input strings. It depends on the distance how these maxima are computed. In exceptional cases where the maximum equals 0 (e.g., q-gram distances and $q = 0$), the ratio in the abovementioned equation equals 0/0, which is defined as equal to one for this purpose. This definition includes the simple case where distances by definition yield values in [0, 1], such as the Jaro–Winkler distance. In that case, Eq. (5.16) reduces to $1 - d(s,t)$.

Computing string similarities is done with the `stringsim` function, which works similar to the `stringdist` function.

```
stringdist::stringsim("hello",c("hallo", "olá"), method="lcs")
## [1] 0.80 0.25
```

Here, lcs distance between `"hello"` and `"hallo"` equals 2 since the characters `"e"` and `"a"` are unmatched. The maximum lcs distance for two strings of length 5 equals $5 + 5 = 10$ [see Eq. (5.11)], so the string similarity equals $1 - 2/10 = 0.8$.

The advantage of such a normalized string similarity over a string distance is that string similarities can be better compared across different string metrics. The normalization to a maximum possible distance yields a more neutral degree of similarity.

q-gram-Based Distances in the `textcat` Package

The `textcat` package offers a number of q-gram-based distances that are especially interesting for text categorization purposes such as language detection. Being based on a method of Cavnar and Trenkle (1994), the computed profile for a string s is the concatenation of profiles for $q = 1$ to $q = 5$. The function `textcat_xdist` computes a distance matrix over a `character` vector based on their profiles.

```
sp_char <- c("Kenny McCormick","Kyle Broflovski","Stan Marsh",
"Eric Cartman")
library(textcat)
textcat_xdist(sp_char)
##      [,1] [,2] [,3] [,4]
## [1,]    0 1891  954 1286
## [2,] 1894    0  980 1349
## [3,]  910  934    0  509
## [4,] 1244 1334  557    0
```

Using the `method` argument, several distance functions between q-gram profiles can be chosen. The default method is the out-of-place measure of Cavnar and Trenkle (1994), which is based on a simple rank statistic over the profiles.

`textcat` supports a number of distance measures on the Cavner–Trenkle profiles, including measures based on the Kullback–Leibler divergence and the cosine distance.

Computing String Kernel Distances with `kernlab`

To compute a string kernel distance, one proceeds in two steps. The function `stringdot` creates a kernel function for subsequences of length q.

```
library(kernlab)
sk <- stringdot()
```

Here, `sk` is a function that computes the cosine between two string kernel profiles. Since the cosine expresses a similarity, we can compute a distance between two strings as follows.

```
1 - sk("kyle brovlofski", "kyle broflovski")
## [1] 0.5384615
```

The default values for λ and q are 1.1 and 4, respectively. These parameters may be altered with the `lambda` and `length` parameter.

```
sk3 <- stringdot(length=3,lambda=0.5)
1 - sk3("kyle brovlofski", "kyle broflovski")
## [1] 0.4285714
```

Deciding on Matching Method and Parameters

The metric and parameter set to be used depends both on the application and the type of strings being matched. Both q-gram-based metrics and string kernels have been widely researched in the context categorization of text corpora. For example, Cavnar and Trenkle (1994), Huffman (1995), Huffman and Damashek (1995), Lodhi *et al.* (2002), and Karatzoglou and Feinerer (2007) all categorize texts based on combinations of q-gram profiles spanning multiple values of q.

In the context of data cleaning, the most relevant application is lookup of polluted items in a list of known valid values. A closely related application is statistical matching where approximate key matching is an important subtask. Several authors have assessed the accuracy of (combinations) of string metrics, in the context of finding spelling mistakes (Zamora *et al.*, 1981), bibliography (Lawrence *et al.*, 1999), name matching (Cohen *et al.*, 2003; Winkler, 1990), and toponym matching (Recchia and Louwerse, 2013; Sehgal *et al.*, 2006). It is hard to point out a single optimal method for each, or all cases. For example, Cohen *et al.* (2003) report good results using the Jaro–Winkler distance (setting $p = 0.1$), while Sehgal *et al.* (2006) obtain better results using edit-based distances.

As a result, the choice of matching algorithm will be a practical trade-off between performance, necessary accuracy, and the amount of resources that can be invested in developing a matching methodology. There are however a few basic considerations.

If strings have an imposed structure such as a fixed length (think of phone numbers, IBAN bank accounts, credit card numbers), the Hamming distance may be of use. If not, (word) order, the supposed error mechanism, string length, and performance are of consideration.

For some strings, like name data or short product descriptions, the order of words may be of less relevance. For example, the strings `"Farnsworth, Hubert J."` and `"Hubert J. Farnsworth"` represent the same person. To compare such strings, a measure that is less dependent on (word) order is to be preferred. As an example, compare the cosine similarity (based on q-gram profiles) with the Jaro–Winkler similarity.

```
stringsim("Farnsworth, Hubert J.", "Hubert J. Farnsworth",
method="cosine", q=2)
## [1] 0.88396
```

```
stringsim("Farnsworth, Hubert J.", "Hubert J. Farnsworth",
method="jw", p=0.1)
## [1] 0.465873
```

Clearly, the fact that q-gram profiles are to an extent independent from in which order the q-grams appear in the string results in a high similarity measure. The Jaro–Winkler distance, which measures a certain type of typing mistakes, results in a much smaller similarity.

When there is reason to suspect that strings differ in only a few characters, an edit-based distance or the Jaro–Winkler distance is a good option. For edit-based distances, long strings may be prohibitive as the computational time increases with the product of the lengths of the strings compared. In Figure 5.3 we compare the computation time necessary to compute the lower tridiagonal distance matrix on 400 randomly generated names with lengths ranging from 8 to 28 characters. The boxplots show timing distributions over 250 repetitions of the experiment, conducted with the `stringdist` package. One immediately recognizes why the Jaro–Winkler distance has become so popular: it is even faster than the fairly simple longest common substring distance. If a more accurate edit-based distance is necessary, one may choose the optimal string alignment distance. It is just a little slower than the Levenshtein method but much faster than the full Damerau–Levenshtein distance. The q-gram distance takes about 40% longer than the optimal string alignment distance in this benchmark; but for longer strings, this disadvantage quickly disappears: running time for q-gram-based distances depends on the sum of the input string lengths which makes them ideal candidates for ploughing through long texts.

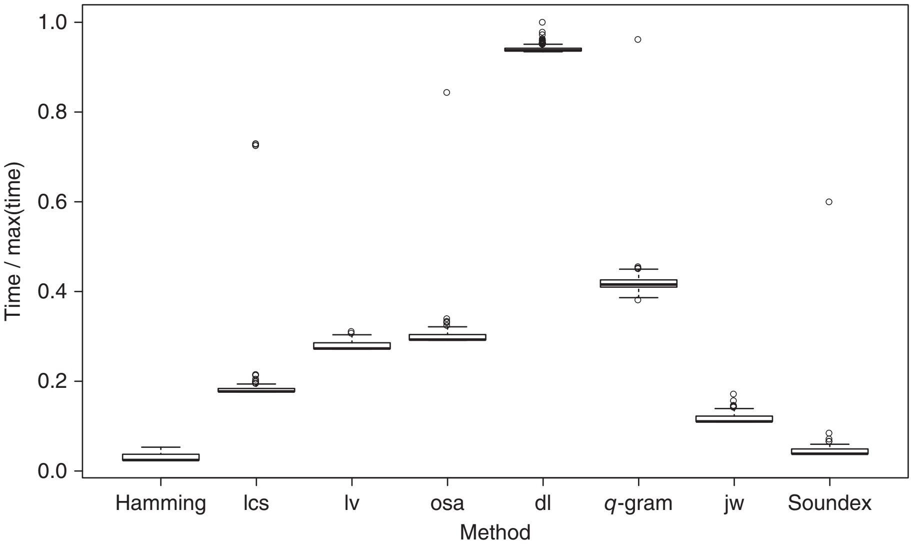


Figure 5.3 Benchmark of times to compute string distance measures in the `stringdist` package (smaller is faster). In this benchmark, the lower triangular of 400 × 400 distance matrix was computed 250 times using `stringdist::stringdistmatrix`. The durations were measured with the `microbenchmark` package. The strings consisted of randomly generated names of lengths varying between 8 and 28 characters.

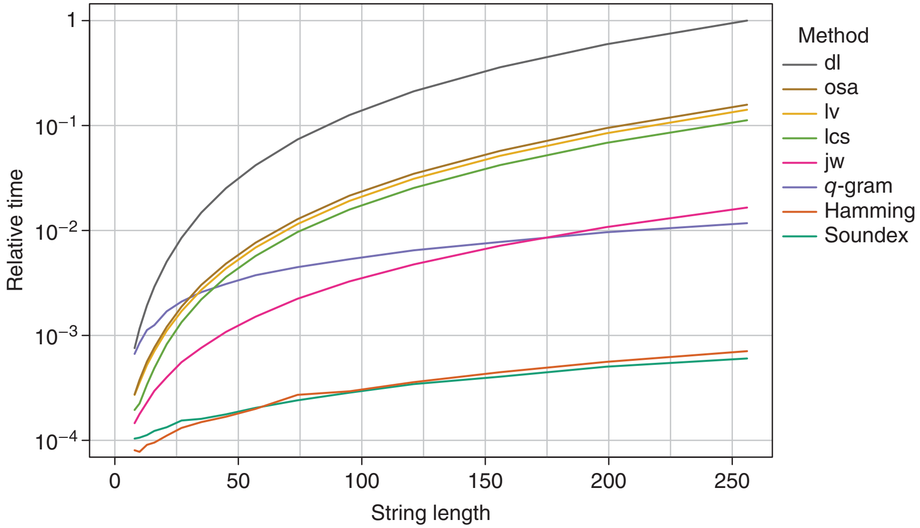


Figure 5.4 Benchmarks of computing string distance measures in the `stringdist` package (smaller is faster). In this benchmark, 100 × 100 lower triangular distance matrices were computed between random strings of lengths varying from 8 to 256 characters. The lines represent median relative times over 10 repetitions. Results have been normalized to the maximum median time. Results for the soundex distance were not plotted as it nearly coincides with that of the Hamming distance.

Figure 5.4 shows a benchmark for string distance calculations between pairs of equally sized random strings of lengths ranging from 8 to 256 characters. At 200 characters length, the asymptotic computational complexity has been reached. The soundex distance is a little faster to compute than the Hamming distance. The full Damerau–Levenshtein distance is slowest in all cases, while the q-gram-based method surpasses the optimal string alignment, Levenshtein, and lcs distance somewhere between 30 and 50 characters. At about 175 characters, q-gram-based distances are computed faster then the Jaro–Winkler distance, which is faster to compute than any edit-based distance for any length of string. In general, accuracy is probably of more concern than performance; but if accuracy is of no or little concern, Figure 5.4 provides a guide of what distance measure to choose. The fact that the Jaro–Winkler distance is about 3–7 times faster than the optimal string alignment while providing acceptable accuracy probably explains its wide popularity in the record linkage community [see, e.g., Cohen *et al.* (2003)]. Finally, we note that functionality of the `stringdist` automatically uses multiple cores so performance can be improved by simply switching to a larger machine.

Measuring Matching Accuracy

Suppose we have a list S of polluted strings and a lookup table T. The task is to match as many polluted strings as possible correctly to elements in T. We assume that for every element in S there should be a corresponding element in T. This matching may be one-to-many, that is, multiple elements of S might match the same element of T. For example, S can be a long list of possibly misspelled place names and T a short list of official place names, and $|S| > |T|$. In general, a matching algorithm generates a subset of pairs $A \subset S \times T$. Suppose that the true match is given by the set $B \subset S \times T$. The *precision* of the approximate matching algorithm is defined as the fraction of elements in A that are correctly matched.

$$\text{Precision}(A|B) = \frac{|A \cap B|}{|A|}.$$

The *recall* is defined as the fraction elements in B that are correctly matched:

$$\text{Recall}(B|A) = \frac{|A \cap B|}{|B|}.$$

Observe that both measures scale from 0 to 1, where 0 corresponds to not a single match ($A \cap B = \emptyset$). A precision equal to 1 means that every found pair is also a true match ($A \subset B$), but not necessarily that every possible match has been found. A recall equal to 1 means that every true match is in A ($B \subset A$) but not necessarily that every pair in A represents a true match.

The precision and recall can usefully be combined into what is called the F_1 measure, which is defined as

$$\begin{aligned} F_1(A, B) &= 2\frac{\text{Precision}(A|B) \times \text{Recall}(B|A)}{\text{Precision}(A|B) + \text{Recall}(B|A)} \\ &= 2\left(\frac{1}{\text{Precision}(A|B)} + \frac{1}{\text{Recall}(B|A)}\right)^{-1}. \end{aligned} \tag{5.17}$$

Here, we silently agree that $F_1 = 0$ if the precision and/or the recall equals zero. It is easy to see that $F_1 = 1$ if and only if $A = B$ and decreases to zero when either the precision or the recall approaches zero.

If test sets with known matches are available, the abovementioned measures can easily be computed for different metrics and parameters. In general, it is a good strategy to use cross-validation techniques to evaluate a matching methodology: split the data into training and test sets, and evaluate F_1 on the test sets. Discussion of cross-validation techniques is beyond the scope of this book, but there are several textbooks that introduce the topic. James *et al.* (2013) discuss basic cross-validation techniques with applications in R, while Hastie *et al.* (2001) focus on theory. A comprehensive overview of cross-validation theory and techniques is given in the paper by Arlot *et al.* (2010).

Encoding Issues

The accuracy of string distance computation may decrease when encoding is not taken into account. In particular, some implementations of (edit-based) string metrics compute distances between the byte-wise representation of strings rather then taking account of possible multi-byte characters. Both the distance functions in base R and in the `stringdist` package by default take account of encoding and have the option to ignore it.

```
stringdist(enc2utf8("Motörhead"), "Motorhead", method="lv")
## [1] 1
stringdist(enc2utf8("Motörhead"), "Motorhead", method="lv", useBytes=TRUE)
## [1] 2
```

Here, we use `enc2utf8` to ensure that the results are the same on all operating systems supported by R (R assumes the OS's native encoding if it is not explicitly specified, see

Section 3.2.4). In the first example, the Levenshtein distance between `"Motörhead"` and `"Motorhead"` equals 1, corresponding to the replacement `"ö"` → `"o"`. In the second example, the strings are treated byte-wise. Since `"ö"` is stored as a two-byte character in `UTF-8`, going from `"ö"` to `"o"` equals removing one byte and substituting another. The situation is even worse than this, since `UTF-8` has a second way to represent `"o"`, namely, as an `"o"`, followed by a modifying symbol that adds the umlaut to the previous character. The two different representations are generally not automatically recognized by implementations of distance metrics.

Therefore, before computing string distances, one should ensure at least that all strings are stored in the same encoding. Here `UTF-8` is a good choice since it both supports (nearly) every symbol invented and is widely supported across different softwares. Second, one needs to ensure that the encoding is normalized so that (combined) characters have a unique byte-wise representation. Techniques for Unicode normalization and encoding conversion were discussed in Section 5.1.1.

6

Data Validation

6.1 Introduction

Data validation—confirming whether data satisfies certain assumptions from domain knowledge—is an essential part of any statistical production process. In fact, a recent survey among the 28 national statistical institutes of the European Union shows that an estimated 10–30% of the total workload for producing a statistic concerns data validation (ESS, 2015). Even though these numbers can be considered only as rough estimates, their order of magnitude clearly indicates that it is a substantial part of the workload. For this reason, the topic of data validation deserves separate attention.

The demands that a dataset must satisfy before it is considered fit for analyses can usually be expressed as a set of short statements or rules rooted in domain knowledge. Typically, an analyst will formulate a number of such assumptions and check them prior to estimation. Some practical examples taken from the survey mentioned earlier (rephrased by us) are as follows:

- If a respondent declares to have income from 'other activities'; fields under 'other activities' must be filled.
- Yield per area (for a certain crop) must be between 40 and 60 metric tons/ha.
- A person below the age of 15 cannot take part in economic activity.
- The field 'type of ownership' (for buildings) may not be empty.
- The submitted 'regional code' must occur in the official code list.
- The sum of reported profits and costs must add up to the total revenue.
- The persons in a married couple must have the same year of marriage.
- If a person is a child of a reference person, then the code of the person's father must be the reference person's code.
- The number of employees must be equal to or greater than zero.
- Date of birth must be larger than December 30, 2012 (for a farm animal).
- Married persons must be at least 15 years old.
- If the number of employees is positive, the amount of salary paid must be positive.
- The current average price divided by last period's average price must lie between 0.9 and 1.1.

The examples include rules where values are compared with constants, past values, values of other variables, (complex) aggregates, and even values coming from different domains or datasets.

The set of knowledge-based validation rules for a dataset can thus be varied and, depending on the number of variables and known relationships between them, may be large. Moreover, since any single variable may occur in several rules, validation rules are often interconnected and may therefore give rise to redundancies or contradictions. For this reason, a systematic way of defining rules, confronting data with them, and maintaining and analyzing rule sets is desirable. From a theoretical point of view, this means that we need a solid (mathematical) definition of what data validation rules are, investigate their properties, and find out in what ways they may be manipulated. From a practical perspective, it means that we need a system where rules can be defined and manipulated independent of the data. Such a system has the added benefits of being able to reuse rule sets across (similar) datasets and allowing for unambiguous communication of rules between different parties working with the data or statements based on them.

To get a feel of what such system can do, we start this chapter by treating a few practical examples using the `validate` package (van der Loo and de Jonge, 2015b). Next, we dive in a little deeper and discuss a formal definition of data validation and give some general properties of sets of data validation rules, followed by a more extensive discussion of `validate`.

6.2 A First Look at the `validate` Package

The `validate` package implements a set of objects and methods that together form an infrastructure for data validation. The main purpose of `validate` is to provide a solid basis for rule declaration (separate from data) either from the command line or file, managing rule metadata, confrontating data with the rules and gathering the results, and summarizing and analyzing such results in a meaningful way.

In this chapter we will demonstrate functionality `validate` using the `retailers` dataset that comes with the package. This raw data file consists of 60 records of anonymized economic information on retailers (supermarkets), with many commonly occurring errors and missing values. Variables include the number of staff, the amount of turnover and other revenues, the total revenue, staff costs, total costs, and the amount of profit in thousands of guilders. The data can be loaded as follows:

```
library(validate)
data(retailers)
```

A complete description of the dataset can be found in the help file by typing `?retailers` at the command line.

6.2.1 Quick Checks with `check_that`

The simplest way to perform validation tasks is to use the `check_that` function. It accepts a `data.frame` and one or more rules. In the following example we check the following assumptions:

- Turnover is larger than or equal to zero.
- The costs per staff are below fl 50,000.

Using `check_that`, this is done as follows:

```
cf <- check_that(retailers, turnover > 0, staff.costs/staff < 50)
```

The object `cf`, storing the validation results, is an object of class `confrontation`, and it stores all the results of confronting the data in `retailers` with the validation rules. Using `summary`, we get an overview per rule.

```
summary(cf)
##   rule items passes fails nNA error warning            expression
## 1   V1    60     56     0   4 FALSE   FALSE          turnover > 0
## 2   V2    60     39     5  16 FALSE   FALSE staff.costs/staff < 50
```

This tells us the following: 60 records were tested against each rule. The first rule, testing positivity of the `turnover` variable, was satisfied by 56 records. It was failed by none, but in four cases it yielded `NA`. Evaluating the rule did not result in errors or warnings. The second rule, a sanity check on costs per staff, was satisfied by 39 records, failed by 5, and could not be computed because of missing values in 16 cases. Furthermore, the last column shows the R expression that was evaluated to compute the outcome of each rule, and the sixth and seventh columns indicate whether an evaluation resulted in a warning or an error. The first column shows the name of the rules which in this case was assigned automatically.

Observe that in the above statement the rules have been defined without a reference to the data. We could use almost the same function call, changing only the dataset to another dataset also containing the variables `turnover`, `staff.costs`, and `staff`. This separation of rules and data is critical for performing data validation in a systematic way.

The rules that can be handled by `validate` may be arbitrarily complex as long as they are actual validation rules (more on that later). As a demonstration we add two rules to the running example. The first rule states that the average profit should be positive, reflecting our intuition that even though individual profits may be negative, the sector as a whole is profitable. The second rule states our expectation that if a retailer has staff, its staff costs should exceed zero.

Introducing the above rules in the running examples yields the following check:

```
cf <-  check_that(retailers
    , to = turnover > 0
    , sc = staff.costs/staff < 50
    , cd = if ( staff > 0 ) staff.costs > 0
    , mn = mean(profit,na.rm=TRUE) > 0
  )
```

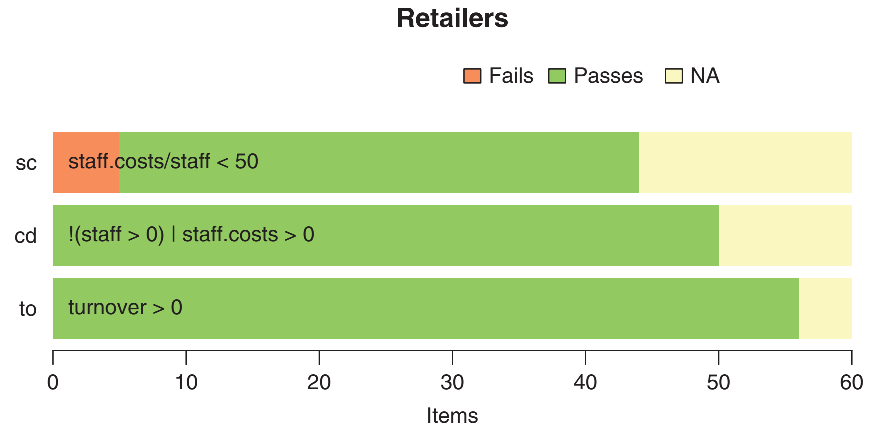
Here, we also gave our rules a short name (for printing purposes) that is copied in the summary.

```
summary(cf)
##   rule items passes fails nNA error warning        expression
## 1   to    60     56     0   4 FALSE   FALSE      turnover > 0
## 2   sc    60     39     5  16 FALSE   FALSE      staff.costs/
                                                     staff < 50
## 3   cd    60     50     0  10 FALSE   FALSE    !(staff > 0) |
                                                staff.costs > 0
## 4   mn     1      1     0   0 FALSE   FALSE      mean(profit,
                                              na.rm = TRUE) > 0
```

Observe that for the rule we named 'mn', only one item was checked. The rule applies to the whole turnover column and not to individual values. Furthermore, observe that the expression for the conditional rule printed in the summary is different from the one we used as input. The input rule if (staff > 0) staff.costs > 0 is translated to the equivalent !(staff>0)|staff.costs > 0 by validate for faster (vectorized) execution. The summary function will always show the expression as it was used during evaluation.

We can create a bar plot for the multi-item rules by selecting the first three results and passing them to barplot.

```
barplot(cf[1:3],main="Retailers")
```



This horizontally stacked bar chart represents the number of items (records) failing, passing, or resulting in missing values for each rule. In fact, we could have plotted the whole cf object not just the results of the first three rules. We leave as an exercise for the reader to try this out and see how barplot handles validation outputs with different dimension structures.

6.2.2 The Basic Workflow: validator and confront

The above examples are nice for quick checks on data on the command line or in simple scripts. For more systematic data validation and rule manipulation, it is better to separate the definition, storage, and retrieval of rules from other activities.

Validation rules can be stored in a validator object. The function validator reads rules from the command line or from file.

```
v <- validator(
    turnover > 0
  , staff.costs / staff < 50
  , total.costs >= 0
  , staff >= 0
  , turnover + other.rev == total.rev
)
v
## Object of class 'validator' with 5 elements:
##  V1: turnover > 0
##  V2: staff.costs/staff < 50
##  V3: total.costs >= 0
##  V4: staff >= 0
##  V5: turnover + other.rev == total.rev
```

Data can be confronted with these rules using confront (not check_that).

```
cf <- confront(retailers, v)
cf
## Object of class 'validation'
## Call:
##     confront(x = retailers, dat = v)
##
## Confrontations: 5
## With fails    : 2
## Warnings      : 0
## Errors        : 0
```

The result is a confrontation object that stores all information relevant to the validation just performed. When printed to the screen, it shows the call that generated it, how many rules have been executed, how many of them were failed at least once, how many of them resulted in errors, and how many raised a warning.

We have seen in the previous section how results stored in a confrontation object can be analyzed using summary or barplot. More detailed output can be obtained with the values function (we use head to limit the printed output).

```
head(values(cf))
##         V1   V2   V3   V4    V5
## [1,]   NA   NA TRUE TRUE    NA
## [2,] TRUE TRUE TRUE TRUE    NA
## [3,] TRUE   NA TRUE   NA FALSE
## [4,] TRUE   NA TRUE   NA  TRUE
## [5,]   NA   NA TRUE   NA    NA
## [6,] TRUE   NA TRUE TRUE    NA
```

The values are returned in an array of class logical, where each column represents a rule, and each row represents a record. The value TRUE means that a rule is satisfied by the validated item, and the value FALSE means that the rule is violated. A missing value (NA) indicates that a rule is evaluated to missing. For example, from the above output, we read that in the third record, V1 (turnover positivity) is satisfied, V5 (a balance check) is violated, and V2 (a bound on the costs per staff member) could not be evaluated. We may inspect the record to spot the cause of this.

```
retailers[3,c("staff","staff.costs")]
##   staff staff.costs
## 3    NA         324
```

In this case, the variable staff is missing.

Summarizing, the basic work flow for data validation suggested by the validate package is as follows:

1) Define a set of rules; using validator.
2) Confront data with the rules using confront.

3) Analyze the results, either with built-in functions or with your own methods after extracting the results.

The `check_that` function is just a simple wrapper that combines the first two steps. In Section 6.5, we elaborate further on these steps, including how to store and retrieve rules from text files and several ways to analyze the results.

6.2.3 A Little Background on `validate` and DSLs

The `validate` package implements a *domain-specific language* (DSL) for the purpose of data validation. Fowler (2010) defines a DSL as

> a computer programming language of limited expressiveness focused on a particular domain.

Indeed, many DSLs lack a lot of the properties one expects from a 'proper' programming language such as abstractions for recursion, iteration, or branching. Nevertheless, many DSLs are considered highly successful. Examples include regular expressions and generalizations thereof and markup languages such as markdown, html, or yaml.

Over time, a number of DSLs have been implemented in R (and previously S), both in the R core distributions as well as in packages. Examples include base R's formula–data interface for the specification of statistical models, the R implementation of the grammar of graphics (Wickham, 2009; Wilkinson, 2005), and the `editrules` package for in-record validation and error localization (de Jonge and van der Loo, 2015), the predecessor of `validate`.

The above examples demonstrate that there are two kinds of DSLs: those defined on its own that allow for different implementations and those defined as a subset or reinterpretation of an existing language. Gibbons (2013) refers to the latter category as *embedded* and to the former as *standalone* DSLs. Clearly, `validate`'s DSL is embedded in R, since the validating statements are formulated in R syntax and reinterpreted by the package to produce validation outcomes. The fact that R hosts a number of DSLs comes as no surprise since, according to the same author 'it turns out that functional programming languages are particularly well suited for hosting embedded DSLs'.

There are many advantages to implementing a data validation syntax embedded into R. First, R's facilities to compute on the language including access to the abstract syntax tree of statements and nonstandard evaluation make experimenting with such an implementation a breeze. In particular, it makes it easy to experiment with different ideas and test them out in practice, something which is much harder while developing a standalone DSL. Second, using R as a host language means access to the truly enormous data processing and statistical capabilities that come with R and its packages at no cost whatsoever. Third, many users interested in data validation are already familiar with R and will be able to use the DSL with relative ease.

The downside of embedding a DSL into another language includes leakage: a user may (unwittingly) 'escape' the DSL and use more advanced features of the host language. A second downside is that the syntax of the host language may be too limited to accurately capture the concepts for which the DSL was designed. It is of interest to note that R is more flexible than many other languages, since it allows for the definition of user-defined infix operators. The most famous example is probably the pipe operator `%>%` of the `magrittr` package.

To accommodate for the possible leakage problem, the validate package filters out statements that are not 'validating statements' when defining a `validator`. For example, trying to include a rule that does not 'validate' anything results in a warning.

```
v <- validator( x > 0, mean(y))
```

To be able to decide what statements actually form a validating rule and to decide what are the relevant (algebraic) operations on sets of such rules, we are going to need a more formal definition of data validation.

Exercises for Section 6.2

Exercise 6.2.1 *Create a* `validator` *object for the* `retailers` *dataset with the following rules:*

- *Nonnegativity rules for variables* staff, staff.cost, *and* total.cost
- `if ( staff > 0 ) staff.cost > 0`
- *The total cost must be larger than or equal to the staff cost.*

Confront this validator object with the `retailers` *dataset and summarize the result with* `summary`. *Answer the following questions:*

a) Which rule is violated most often?
b) Which rule resulted in the most missings?
c) Discuss the expression for the conditional rule on staff *and* staff.cost*.*
d) Create a bar chart of the `confrontation` *object.*

6.3 Defining Data Validation

At the core, data validation is a falsification process. That is, one tries as much as possible to make assumptions about data explicit and verifies whether they hold up in practice. Once sufficient assumptions have been tried and tested, one considers a dataset fit for use, that is, valid. Not performing any validation corresponds to blindly assuming that the values recorded in a dataset are acceptable as a facts, in the sense that they are usable for the production of statistical statements.

Every assumption made explicit through a data validation process limits the set of acceptable values or value combinations. In words, data validation can therefore be defined as follows (ESS, 2015; van der Loo, 2015b):

> Data validation is an activity that consists of verifying whether a collection of values comes from a set of predefined collections of values.

As we saw in the earlier examples, the term 'collection of values' may consist of a single value only (as in: *turnover* must be nonnegative). Also, the term 'predefined' should be interpreted generally. In many cases, the set of 'acceptable values' or combinations thereof is not defined explicitly but rather as a set that is implied by an involved calculation. As an example, consider the rule where we deem a numerical record invalid when it is more than some distance d_0 removed from the mean vector of the data, where the distance is computed as the Mahalanobis distance. In this case, the calculation involves estimating the (inverse) covariance matrix, computing the distance with the mean vector, and comparing the result with d_0. The set of allowed or disallowed records is not defined explicitly. However, since the procedure of validation is fully deterministic (and we assume this to be the case in all validation procedures), we deem the collection of valid value combinations predetermined.

The above definition includes validation activities made explicit through (mathematical) validation rules as well as procedural validation such as expert review based on fixed methodology. In this work, we limit ourselves to data validation procedures that can be fully automated, which is why we move on to discuss a formal definition that underlies choices made in the `validate` package.

6.3.1 Formal Definition of Data Validation

Since validation is a process that consists of decision-making about data, it is tempting to define a validation function mathematically as a function that takes a collection of data and returns 0 (invalid) or 1 (valid). There are, however, some subtleties to take care of when constructing such a definition. Most importantly, we need to construct precisely what the domain of such a function is. Considering the examples in Section 6.2, the domain must encompass single values, records, columns, multiple tables, or any other collection of data points. Simply stating that the domain is 'the set of all datasets' will not do, because this is not a proper set-theoretical definition: the set of all datasets contains itself recursively.

To solve this, we first define a data point as follows:

Definition 6.3.1 *A* data point *is a pair* (k, v)*, where* k *is a* key*, from a set of keys* K*, and* v *is a* value *from a predefined domain* D*.*

The purpose of the key is to make the corresponding value identifiable in a collection of data. At the very least it identifies the variable of which the value is a realization, but a key is often represented by a collection of subkeys that together identify the statistical unit, the variable, the measurement, and so on. For the moment the precise information stored in the key is unimportant, but in Section 6.4 we will discuss a general set of keys that allow us to classify data validation functions. The value domain, D, depends on the context in which the value is obtained. It is the basic domain on top of which further assumptions must be checked. For example, if we know all values are integers, the domain D is $\mathbb{Z}$, and if no type-checking took place, one can set the domain to the set of strings Σ^* over a suitable alphabet Σ. If the value can be either text or integer, the domain can be defined as $\mathbb{Z} \cup \Sigma^*$.

Given a set of keys K and a domain D, the set of all possible key-value pairs is the Cartesian product $K \times D$. A dataset can now be defined as follows:

Definition 6.3.2 *A* dataset *is a subset of* $K \times D$*, where every key in* K *occurs exactly once.*

An informal way of stating this definition is to say that a unique measurement must yield a unique value. A more technical way of stating this is to say that a dataset is a function from K to D. We denote the set of all datasets associated with K and D as D^K.

Example 6.3.3 *Suppose that we perform a survey among a city's population. We draw two people randomly from the city's population register. Their social security numbers happen to be 1 and 2. We ask each person two questions: (1) do you currently have a job? and (2) what is your age in years? To identify a value from this measurement, we need two types of information in our key: the social security number and the name of the measured variable. The set of keys, K, for this measurement is given by*

$$K = \{(1, \text{job}), (1, \text{age}), (2, \text{job}), (2, \text{age})\}.$$

The domain is the union of the domains for the variables job *and* age*:*

$$D = \{\text{true}, \text{false}\} \cup \mathbb{Z}.$$

In this formulation, the set of all possible outcomes D^K can be written as a table

$$T = \begin{bmatrix} T_{1,job} & T_{1,age} \\ T_{2,job} & T_{2,age} \end{bmatrix},$$

where each row corresponds to a person and each column to a variable. Each $T_{ij} \in D$, so we can write formally that $T \in D^K = D^{2\times2}$.

The above example shows that our formulation of a dataset is very tolerant in the sense that it allows to store a logical value for a numerical variable and vice versa. As a consequence, our definition of validation will include type-checking.

Definition 6.3.4 *Let D^K be the set of possible datasets as defined in Section 6.3.2. A* data validation function *v is a surjective function*

$$v : D^K \twoheadrightarrow \{0, 1\}. \tag{6.1}$$

So a formal data validation function accepts a complete dataset in D^K and returns 0 (invalid or 'false') or 1 (valid or 'true'). Validation functions are defined to be surjective (onto) since functions that by their definition always return 1 are not informative (all data satisfies the assumption expressed by the function), and functions that always return 0 are contradictions: no collection of data in D^K can satisfy the assumption expressed by the function. Surjectivity is thus implied by the notion that validation is really an attempt to falsify assumptions we might have about the properties of a dataset. Any test that by definition is always failed or always passed cannot be an attempt at falsification.

Example 6.3.5 (Continued from 6.3.3) *The following functions are validation functions:*

$$\begin{aligned} v_1 &: D^{2\times2} \to \{0, 1\}, \\ &T \mapsto \begin{cases} 1 \textit{ if } T_{1,job} \in \{\text{true}, \text{false}\} \\ \textit{else } 0. \end{cases} \\ v_2 &: D^{2\times2} \to \{0, 1\} \\ &T \mapsto \begin{cases} 1 \textit{ if } \forall i \in \{1, 2\} : T_{i,age} \geq 0 \\ \textit{else } 0. \end{cases} \end{aligned}$$

$$
\begin{aligned}
v_3 : D^{2\times 2} &\to \{0,1\}\\
T &\mapsto \begin{cases} 1 \textit{ if } (T_{1,job} = \text{true}) \Longrightarrow T_{1,age} > 15\\ \textit{else } 0. \end{cases}
\end{aligned}
$$

Here, v_1 *checks whether the variable* job *in the first record is of the correct type. Function* v_2 *checks whether all age values are nonnegative, and* v_3 *expresses that if person 1 has a job, age should be over 15. The following functions are not validating functions:*

$$
\begin{aligned}
v_4 : D^{2\times 2} &\to \{0,1\}\\
T &\mapsto \begin{cases} 1 \textit{ if } T_{1,job} \in D\}\\ \textit{else } 0 \end{cases}\\
v_5 : D^{2\times 2} &\to \{0,1\}\\
T &\mapsto \begin{cases} 1 \textit{ if } T_{1,age} \neq T_{1,age}\\ \textit{else } 0. \end{cases}
\end{aligned}
$$

Here, v_4 *does not check anything. Recall that D was defined as the domain of measurement, so by definition any value of the dataset is recorded as a value of D. Hence, the outcome of* v_4 *can only be 1. The function* v_5 *is a contradiction that will always yield 0.*

Any validation function v by definition separates D^K into two regions. The *valid region* is defined as the preimage $v^{-1}(1) = \{T \in D^K : v(T) = 1\}$. The corresponding *invalid region* is defined as $v^{-1}(0) = \{T \in D^K : v(T) = 0\}$. Given a set of validation functions, the valid region is the intersection of their respective valid regions. Since it is in general not guaranteed that the intersection of two or more valid regions is nonempty, care must be taken when formulating such sets. We will return to this in the following section, but let us first conclude with some remarks on practical issues.

The example validation functions in Section 6.3.5 are defined in a formal way which is not how one usually defines or discusses them. For example, instead of writing a function that checks the relation between *job* status and *age* for each individual record, one formulates the rule

$$job = true \Longrightarrow age > 15,$$

and simply implies that it must hold for each record separately.

6.3.2 Operations on Validation Functions

For effective reasoning about validation functions, it is important to consider the basic operations under which the set of validation functions on a certain domain D^K is closed. Since the result of a validation function is boolean, it is tempting to try to combine validation functions with the standard boolean operators, negation ($\neg$), conjugation ($\wedge$), and disjunction ($\vee$) by defining $(\neg v)(T) = \neg(v(T))$, $(v_1 \wedge v_2)(T) = v_1(T) \wedge v_2(T)$, and $(v_1 \vee v_2)(T) = v_1(T) \vee v_2(T)$.

Observation 6.3.6 *If* v *is a validation function according to Definition 6.3.4, then* $\neg v$ *is the validation function of which the valid region is equal to the invalid region of* v.

Proof: First observe that $\neg v$ must be surjective on $\{0, 1\}$ if and only if v is, since the action of $\neg$ is to swap 0 and 1 in the outcome of v. Using the definition of the valid region of a validation function v, we can simply compute the valid region of $\neg v$.

$$\begin{aligned}(\neg v)^{-1}(1) &= \{T \in D^K\ :\ (\neg v)(T) = 1\}\\ &= \{T \in D^K\ :\ \neg(v(T)) = 1\}\\ &= \{T \in D^K\ :\ v(T) = 0\} = v^{-1}(0),\end{aligned}$$

which completes the proof. □

For the $\vee$ operation, we have the following observation, which is proven in Exercise 6.3.2.

Observation 6.3.7 *Given two validation functions v_1 and v_2, the valid region of $v_1 \vee v_2$ is equal to the union of the valid regions of v_1 and v_2.*

It follows that combining validation functions by disjunction does not necessarily result in another validation function since the property of surjectivity may be lost. That is, one can think of cases where the outcome of $v_1 \vee v_2$ equals 1 for every dataset in D^K.

Corollary 6.3.8 *If v_1 and v_2 are validation functions, then in general $v_1 \vee v_2$ are not.*

Proof: Consider the case of a single measured numerical value x so that $D^K = \mathbb{R}$. We take

$$v_1(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ \text{else } 0, \end{cases}$$

$$v_2(x) = \begin{cases} 1 \text{ if } x \leq 1 \\ \text{else } 0. \end{cases}$$

Here, the valid region of v_1 is the range $[0, \infty)$ and the valid region of v_2 is $(-\infty, 1]$. Since the valid region of $v_1 \vee v_2$ equals $[0, \infty) \cup (-\infty, 1] = \mathbb{R} = D^K$, it is not surjective on $\{0, 1\}$ (it always yields 1 by definition) and therefore not a validation function. □

The most important practical consequence of the above result is that one needs to be careful when defining validation rules consisting of several disjunctions. Especially for validation functions that involve complex calculations or derivations, determining the valid region explicitly may be hard.

The following observation also has practical consequences since it limits what validation functions can be put together in a set.

Observation 6.3.9 *Given two validation functions v_1 and v_2, the valid region of the function $v_1 \wedge v_2$ is equal to the intersection of the valid regions of v_1 and v_2.*

Proof:

$$\begin{aligned}(v_1 \wedge v_2)^{-1}(1) &= \{T \in D^K\ :\ (v_1 \wedge v_2)(T) = 1\}\\ &= \{T \in D^K\ :\ v_1(T) = 1 \wedge v_2(T) = 1\}\\ &= \{T \in D^K\ :\ v_1(T) = 1\} \cap \{T \in D^K\ :\ v_2(T) = 1\}\\ &= v_1^{-1}(1) \cap v_2^{-1}(1).\end{aligned}$$

□

It follows from this observation that two validation functions cannot in general be conjuncted to form another validation function. In particular, this happens when v_1 and v_2 contradict.

Corollary 6.3.10 *If v_1 and v_2 are validation rules, then $v_1 \wedge v_2$ is not necessarily a validation rule.*

Proof: Consider the case of a single measured numerical value x so that $D^K = \mathbb{R}$. We take

$$v_1(x) = \begin{cases} 1 \text{ if } x \leq 0 \\ \text{else } 0, \end{cases}$$

$$v_2(x) = \begin{cases} 1 \text{ if } x \geq 1 \\ \text{else } 0. \end{cases}$$

Here, the valid region of v_1 is the range $(-\infty, 0]$, and the valid region of v_2 is $[1, \infty)$. Since the valid region of $v_1 \wedge v_2$ equals $[0, \infty) \cap (-\infty, 1] = \emptyset$, it is not surjective on $\{0, 1\}$ (it yields 0 by definition) and therefore not a validation function. □

A practical consequence of the above result is that when one defines multiple validation functions on a dataset, which amounts to joining them by conjunction, one needs to check if no contradictions occur. In principle, given a set of validation functions $\{v_1, v_2, \ldots, v_m\}$, one can confirm that it is internally consistent by computing the preimage

$$(v_1 \wedge v_2 \wedge \cdots \wedge v_m)^{-1}(1) = \cap_{i=1}^{m} v_i^{-1}(1).$$

It is easy to see that $\cap_{i=1}^{m} v_i^{-1} = \emptyset$ if and only if for some $s \subseteq \{1, 2, \ldots, m\}$, we have $\cap_{i \in s} v_i^{-1} = \emptyset$. In reality, however, computing the valid region explicitly can be a daunting if not an impossible task. In practice, algorithms for confirming or rejecting the internal consistency for a set of rules exist only for some classes of validation functions such as linear (in)equalities and certain logical assertions.

Let us briefly summarize what we have found so far. After stating that validation is an attempt to falsify assumptions about a dataset, we found that this notion could be formalized as a surjective function that has as codomain the boolean set $\{0, 1\}$. This surjectivity then, places strong restrictions on how we may combine such functions to form new ones. Essentially, we can only guarantee with certainty that the negation of a validation function is also 'validating', but with opposite results, and we have to be careful when defining sets of validation functions.

6.3.3 Validation and Missing Values

In the previous discussion we have quietly assumed that validation functions can always be computed, and in the strict sense this is true. If one allows the missingness indicator NA in the measurement domain D, one can always define validation functions such that NA is handled as a specific case. For example, consider a single numerical variable x with the measurement domain $D = \mathbb{R} \cup \text{NA}$, so x might be missing after measurement. Suppose that we have the rule that x must be positive. We could define the validation function as follows:

$$v(x) = \begin{cases} 1 \; \textit{if} \; x \notin \{\text{NA}\} \wedge x > 0 \\ \textit{else} \; 0. \end{cases}$$

Here, the result is explicitly set to 0 if x is missing. Such definitions quickly become cumbersome when v involves multiple variables, since the check for missingness has to be included explicitly for every variable. It is therefore desirable to choose a default value for such cases.

Two obvious choices present themselves: either every calculation that cannot be completed because of a missing value results in 0, and the definition of validation functions remains the same, or missingness is propagated through the calculation, and the codomain of validation functions is extended from $\{0, 1\}$ to $\{0, 1, \mathsf{NA}\}$. There is a third option that maps every missing value to 1 (valid), but this does not seem to have a meaningful interpretation.

The advantage of extending the codomain to $\{0, 1, \mathsf{NA}\}$ is that one obtains a more detailed picture from the validation results. The disadvantage is that analyses of validation results become slightly more complex. In the R package `validate`, propagation of missingness is the default (as it is in R) but this behavior can be controlled by the user.

6.3.4 Structure of Validation Functions

As shown in the examples of Sections 6.1 and 6.2, validation functions may involve arbitrarily complex calculations. The result of such a calculation is then compared with a range of valid calculation results. As an example, reconsider the following rule, mentioned in the introduction:

> The current average price divided by last period's average price must lie between 0.9 and 1.1.

This rule can be stated as (y denoting *price*).

$$\frac{\bar{y}_t}{\bar{y}_{t-1}} \in (0.9, 1.1), \tag{6.2}$$

where $(0.9, 1.1)$ denotes the range $\{x \in \mathbb{R} : 0.9 < x < 1.1\}$. To evaluate this rule, first the average prices at time t and $t - 1$ are determined, and next, the result is compared with the range $(0.9, 0.1)$. As another example, consider the rule

> the sum of cost and profit must equal revenue.

This can be written as (denoting c for cost, p for profit, and r for revenue)

$$c + p - r \in \{0\}.$$

We first compute the linear combination on the left-hand side and then check set membership with the valid computed value set $\{0\}$. Finally, consider the rule

> if the number of staff is larger than zero, the amount of salary paid must be larger than zero.

Using the implication replacement rule, we may write

$$\neg(\mathit{staff} > 0) \vee (\mathit{salary} > 0) \in \{\texttt{true}\}.$$

Again, we need to perform a calculation on the left-hand side and compare the result with a valid computed set ($\{\texttt{true}\}$) in this case.

This above suggest that it may be useful write a validation function υ as a compound function (van der Loo and Pannekoek, 2014)

$$\upsilon = i_V \circ s, \tag{6.3}$$

that is, $\upsilon(x) = i_V(s(x))$. Here, s is often referred to as a *score function* that computes an indicator, usually a single number or other (categorical) values from the input dataset $x \in S$. The function i_V is the *set indicator* function, returning 1 if its argument is in V and 0 otherwise. In practice, it may occur that s cannot be computed because of missing values. In such cases, we define that the result is NA and $i_V(\mathsf{NA}) = \mathsf{NA}$ as well.

To see what this all means, let us work through the above examples. In the example of Eq. (6.2), we have

$$s(\boldsymbol{y}_t, \boldsymbol{y}_{t-1}) = \frac{E(\bar{y}_t|\boldsymbol{y}_t)}{E(\bar{y}_{t-1}|\boldsymbol{y}_{t-1})} \text{ and } V = (0.9, 1.1),$$

where $\boldsymbol{y}_t$ and $\boldsymbol{y}_{t-1}$ are vectors of observed values and $E(y_t|\boldsymbol{y}_{t,t-1})$ their estimated means. In example (6.3.4), we have

$$s(c, p, r) = c + p - r \text{ and } V = \{0\}.$$

For the final example, we may define s and V as follows:

$$s(salary, staff) = \begin{cases} \texttt{true} \text{ when } (staff \leq 0) \vee (salary > 0) \\ \texttt{false} \text{ otherwise.} \end{cases} \quad \text{and } V = \{\texttt{true}\}.$$

Using the decomposition of Eq. (6.3), we see that a validation function can be fully defined by specifying a score function s and a region V of valid scores. An interesting special case occurs when we choose s to be the identity function: $s(x) = x$. In that case V is equal to the valid region in S: $\upsilon^{-1}(1)$. Simple range checks fall in this category. For example, the rule $y \geq 0$ can be written as $y \in [0, \infty)$.

6.3.5 Demarcating Validation Rules in `validate`

When constructing a set of validation rules with the `validator` function, the parser of `validate` must decide whether the statements the user specifies are actually validating rules (functions) as in Definition 6.3.4. This definition comprises two aspects that are checked by the constructor: validation rules must yield a value in $\{\mathsf{FALSE}, \mathsf{TRUE}, \mathsf{NA}\}$, and validation rules must be surjective.

The latter condition is checked by verifying that the expressions submitted by the user contain at least one variable, so the result can in principle vary when confronted with different values for that variable. This means that statements like

```
1 > 0
```

are filtered from the input (with a warning).

To test whether the first condition is satisfied, that is, whether the rule expresses a validating function, we need to check the following options. The final operation when evaluating a validation function is either

- a set membership function (i.e., `%in%`, <,==, some specialized R functions),
- a logical quantification or existence operator (`all`, `any`, some specialized R functions) or
- the negation of such an expression.

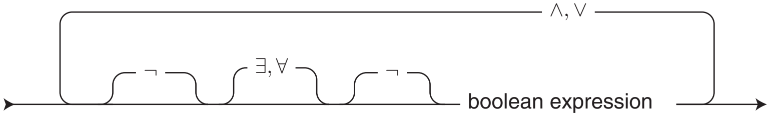
Due to R's flexibility, there are in principle two loopholes that may cause this check to fail. Users may locally overwrite standard functions such as `'<'` with an arbitrary two-parameter function. To cope with this situation, the constructor checks whether this has happened and emits a warning if it is the case. Alternatively, users may define a constant score function, thereby hiding the nonsurjectivity of the rule from this analysis. For example, the rule

```
x - x == 0
```

will always yield true and is accepted by `validate`. Although it is impossible to rule out all such cases, in Chapter 8, we shall see how for certain classes of rule sets contradictions and redundancies can be flushed out automatically.

The above operations are a translation of the strict definition of validation functions (6.3.4) and the allowed operations on them. As a compromise to usability, the strictly allowed expressions may still be combined using boolean operators (and, or, exclusive or, implication) and quantifiers (for all, exists) in `validate`. As discussed before, it can not be guaranteed that boolean combinations of validation functions are also validation functions in the strict sense of Definition 6.3.4. It is therefore up to the user to make sure that no rules that are constant by definition are submitted.

The allowed syntax can be symbolically summarized with the following syntax diagram:



Here, `boolean expression` is an R expression that is confirmed to result in a `logical` and contain at least one variable. Note that this diagram allows for implication ($\Longrightarrow$) since $P \Longrightarrow Q = \neg P \vee Q$ and exclusive or ($\oplus$) since $P \oplus Q = \neg P \wedge Q \vee P \wedge \neg Q$. There are several 'syntactic sugar' options to support such statements. For example `if` and `xor` are allowed specifically. A more complete description of the options is given in Section 6.5.

Exercises for Section 6.3

Exercise 6.3.2 *Proof Observation 6.3.7. Tip: compute the valid region as in Observation 6.3.6.*

Exercise 6.3.3 *We define a* generalized *validation function $\tilde{v}$ as a function $\tilde{v} : D^K \twoheadrightarrow \{0, 1, NA\}$, analogously to the validation function of Definition 6.3.4. Prove that Observations 6.3.6, 6.3.7 and their corollaries still hold.*

6.4 A Formal Typology of Data Validation Functions

In the previous sections we saw several kinds of validation rules such as in-record and cross-record rules, cross-dataset rules, and so on. On one hand, this subdivision in rule types is more or less intuitive as it is closely related to everyday thinking about data in terms of fields, records, and tables. On the other hand, it is not a very robust subdivision. For example, merging two files can turn a cross-dataset validation rule into an in-dataset validation rule.

Having a more robust typology has benefits. It allows one to compare validation processes over different statistical production processes or to compare the capabilities of various softwares implementing validation methods. In the following sections we discuss a method for classifying data validation functions by studying the type of sets of data points they take as input. For this, we need to have a minimal set of keys that identify a value, which is the topic of the following section. The discussion in this section is largely based on a paper by van der Loo (2015b), which in itself was inspired partly by a paper of Gelsema (2012),

6.4.1 A Closer Look at Measurement

Recall that we defined a *data point* as the combination of a set of *keys* and a single `value`. The purpose of the key set is to identify the value: what variable is it, the property of which statistical object it measures, and so on. In this section, we discuss a minimum set of keys that describe a data point obtained by a statistical measurement. In the following section, these keys will be used to classify validation rules.

To find a set of keys that identify a value, let us have a closer look at how a data point is obtained by measurement. Figure 6.1 gives an overview of the timeline involved in creating a data point. At some time, t_u a statistical object is born or created. This may be a person, a company, a phone call, an e-mail, or any other event or object of interest. In any case, we can think of u as an element of a set U of objects of equal type, where U contains all objects that ever lived, live now, and ever will live. From t_u onward, the object of interest has some properties, say X, that may or may not vary over time. At time τ, we choose to select u for measurement and to measure the value of X. The actual measurement may pertain to the value of X at time $t = \tau$, or it may pertain to an earlier time (e.g., we may ask a person, did you have a job last year?) or even a time in the future (do you expect to have a job next year?). For the current discussion these times are immaterial: we are only interested in the time of measurement τ since that determines on which population the final statistical statement will be based. The time to which a measurement pertains is interpreted as belonging to the definition of the measured variable X. Finally, sometime after the measurement took place, the element u disappears from the current population at t'_u.

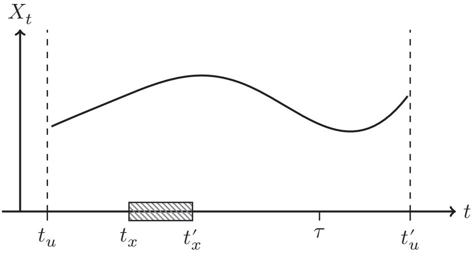


Figure 6.1 The various times involved in a measurement process. A population member u exists over the period $[t_u, t'_u)$. At the time of measurement τ, a value of X_t is observed pertaining to the period $[t_x, t'_x)$. In principle, τ may be before, within, or after this period. Also, instead of a period, one may choose a moment in time by letting $t_x \rightarrow t'_x$.

Based on this discussion, we find four aspects that identify a data point. These are as follows:

- The set U that represents all objects of a certain type that ever were, are, and will be;
- The time of measurement τ, which determines what part of U is active;
- The element $u \in U$, chosen at time τ;
- The variable X that is measured.

If we denote the timeline with T and the domain of X with D, a measurement can be represented as a series of maps

$$T \xrightarrow{p} 2^U \xrightarrow{I_u} U \xrightarrow{X_\tau} D, \tag{6.4}$$

where $p(\tau)$ is the population at time τ, I_u selects element u from $p(\tau)$, and $X_\tau(u)$ results in a value in D.

As an example, let us see what the values for these keys are for the `retailers` dataset in the `validate` package. The set U consists of all Dutch retailers ever, past, present, and future. Now that is a big set, and it is impossible to construct explicitly, but since it only serves to indicate with what objects we are dealing that is fine. The time of measurement τ is the time at which the survey was executed (including establishing the survey frame). In practice, this time is hard to establish already because survey execution will take a time span rather than a moment in time. That is no problem for this model: τ may be implemented as a unique identifier for the survey as long as it uniquely identifies the current population from which the survey was drawn. The element u corresponds to a single interviewed retailer and therefore with one record in the actual data. The measured variable is any of the columns obtained in the survey, for example *staff costs* (and it pertains to the period preceding the period of measurement).

To check the validity of values in the `retailers` dataset, one may perform cell-by-cell checks, check the variable averages against each other, or compare aggregates of variables with aggregates of a dataset pertaining to a different population (wholesalers, say). Depending on the validation rule, different slices of a dataset are needed, or to put it otherwise, different keys need to be varied over in order to be able to compute the validation function. This is the idea behind the following classification.

6.4.2 Classification of Validation Rules

The idea behind the classification discussed here is that we label each basic key in (U, τ, u, X) whether its value must be varied over in order to compute a particular validation function. Naively, we obtain $2^4 = 16$ categories, labeled according to whether one must vary no keys (1 option), a single key (4 options), two keys (6 options), three keys (4 options), or four keys (1 option). However, this number is diminished since the keys cannot be varied completely independent of each other.

First, once the domain is chosen, the type of statistical objects is fixed, which implies that the chosen properties (variables) are fixed as well. As a consequence, validation rules that pertain to different domains must also pertain to different variables. Confusingly, variables for different object types often have similar names. For example, the variable name *income* may relate to a household, a person, or a company. Indeed, these are all some types of incomes, but since they relate to a different object type and are measured in a different way, they must be considered as separate variables nonetheless. For this reason, validation rules that pertain to a multiple domains must also pertain to different variables.

Second, observe that even though an object can be represented in multiple domains (e.g., companies and companies with more than 100 employees), it makes no sense to compare data points where the only difference is that the same object is selected from different (nested) domains. It is assumed that the selection itself does not interfere with measuring the value. This means that validation rules that pertain to different domains but the same statistical unit are meaningless.

Taking these two considerations into account, we arrive at 10 feasible and mutually exclusive categories. To denote them, we will use the labels s for 'single' and m for 'multiple' key values. For example, the label $U\tau uX = sssm$ indicates a rule that ties to a single domain, a single measurement time, a single statistical unit, and multiple variables. An example of such a rule is *turnover* > *costs.*

Table 6.1 summarizes the possible categories. The categories have been further put in groups, where in each group the same number of keys are varied. The idea of these 'validation levels' is not that a higher level indicates a better higher quality of validation procedure. Rather, it is aimed to qualitatively indicate the breadth of data that is required for a single validation. Going from level 0 to level 4 is not totally unrelated to common practice in statistical analyses and production. One usually starts with simple tests on data points, checking against ranges or code lists, and only then continues with more involved tests relating to multiple variables, earlier versions of the data, and so on. At higher levels, one may compare (complex) aggregates of a dataset in one domain (e.g., retailers) with aggregates from another domain (e.g., wholesalers). Let us work through some examples from the list in the beginning of the chapter.

Example 6.4.11 *The rules*

- *If a respondent declares to have income from 'other activities', fields under 'other activities' must be filled.*
- *Yield per area (for a certain crop) must be between 40 and 60 metric tons/ha.*
- *A person below the age of 15 cannot take part in economic activity.*
- *Married persons must be at least 15 years old.*
- *If the number of employees is positive, the amount of salary paid must be positive.*

all pertain to a single statistical unit (and therefore domain), a single measurement time, and two variables. The category is sssm.

Table 6.1 Classification of validation functions, based on the combination of data being validated, comes from a single (s) or multiple (m) domains U, times of measurement τ, statistical objects u, or variables X.

Validation level				
0	**1**	**2**	**3**	**4**
ssss	sssm	ssmm	smmm	mmmm
	ssms	smsm	msmm	
	smss	smms		

Example 6.4.12 *The rules*

- *The field 'type of ownership' (for buildings) may not be empty.*
- *The submitted 'regional code' must occur in the official code list.*
- *The number of employees must be equal to or greater than zero.*
- *Date of birth must be larger than December 30, 2012 (for a farm animal).*

pertain to single variable, unit, and measurement time (and thus domain). The category is ssss.

Example 6.4.13 *The rules*

- *The persons in a married couple must have the same year of marriage.*
- *If a person is a child of a reference person, then the code of the person's father must be the reference person's code.*

combine two or more statistical units and two variables (we assume that relations between records are encoded by suitable variables, e.g., a marriage registration number). The category is ssmm.

Example 6.4.14 *The rule*

- *The current average price divided by last period's average price must lie between 0.9 and 1.1.*

combines data over multiple units and multiple measurement times, within a single domain, and in a single variable. The category is smms.

6.5 Validating Data with the `validate` Package

The vocabulary for data validation in the validate package is simple and fairly limited: a `validator` object stores a set of validation rules, which can be confronted with data using the `confront` function. The result is an object of class `validation`, which stores the results.

6.5.1 Validation Rules in the Console and the `validator` Object

A set of rules can be defined using the `validator` function.

```
v <- validator(
    x > 0
  , y > 0
  , x + y == z
  , u + v == w
  , mean(u) > mean(v))
```

A validator object can be inspected by printing it to the command line or by summarizing it.

```
summary(v)
##   block nvar rules linear
## 1     1    3     3      3
## 2     2    3     2      1
```

The summary method separates the rules into separate blocks (subsets of rules that do not share any variables) and prints some basic information on the rules: the number of separate variables occurring in each block, the number of rules in each block, and the number of linear rules per block.

Like ordinary R vectors, a `validator` object can be subsetted, using `logical`, `character`, or `numeric` vectors in single square brackets.

```
v[c(1,3)]
## Object of class 'validator' with 2 elements:
##  V1: x > 0
##  V3: x + y == z
## Options:
##  raise: none
##  lin.eq.eps: 1e-08
##  lin.ineq.eps: 1e-08
##  na.value: NA
##  sequential: TRUE
##  na.condition: FALSE
```

Using the double bracket operator, more information about a certain rule can be extracted.

```
v[[1]]
##
## Object of class rule.
##  expr       : x > 0
##  name       : V1
##  label      :
##  description:
##  origin     : command-line
##  created    : 2017-06-30 16:34:39
```

This reveals the full information stored for each rule in `v`. A rule contains at least an expression and a `name` that can be used for referencing the rule in a `validator` object. The origin of the rule (here: `command-line`) is stored as well as a time stamp of creation. Optionally, a `label` (short description) and a long explanation can be added. This information is intended to be used for rule maintenance or when compiling automated data validation reports.

Users will normally not manipulate `rule` objects directly, but it can be useful to access them for inspection. The following functions extract information on rules from `validator` objects:

`names`	Name of each rule in the object
`origin`	Origin of each rule in the object
`label`	Label of each rule in the object
`description`	(long) Description of each rule in the object
`created`	Timestamp (`POSIXct`) of each rule in the object
`length`	The number of rules in the object
`variables`	The variables occurring in the object

The functions `names`, `origin`, `label`, `description`, and `created` also have a property setting equivalent. For example, to set a label on the first rule of `v`, do the following:

```
label(v)[1] <- "x positivity"
```

When present, the labels are printed when a validator object is printed to screen.

The `variables` function retrieves the list of all variables referenced in a validator object.

```
variables(v)
## [1] "x" "y" "z" "u" "v" "w"
```

Optionally, variables can be retrieved per rule, either as a list or as a matrix.

```
variables(v, as="list")
variables(v, as="matrix")
```

With the matrix option set, `variables` returns a logical matrix where each row represents a rule, and each column represents a variable.

Objects of class `validator` are reference objects, which means that they are not usually copied when you pass them to a function. Specifically, the assignment operator does not make a copy. For example, we may set

```
w <- v
```

and query the names.

```
names(v)
## [1] "V1" "V2" "V3" "V4" "V5"
names(w)
## [1] "V1" "V2" "V3" "V4" "V5"
```

Now, since `v` is a reference object, `w` is just a pointer to the same physical object as `v`. This can be made visible by altering one of the attributes of `v` and reading it from `w`.

```
names(v)[1] <- "foo"
names(w)
## [1] "foo" "V2"   "V3"   "V4"   "V5"
```

An exception to this rule is when a subset of a validator object is selected using the bracket operators. In that case, the resulting object is completely new. So the following trick produces a physical copy of `v`.

```
w <- v[TRUE]
```

6.5.2 Validating in the Pipeline

The pipe operator (`%>%`) of the `magrittr` package (Bache and Wickham, 2014) makes it easy to perform consecutive data manipulations on a dataset. The functions `check_that` and `confront` have been designed to conform to the pipe operator. For example, we can do

```
retailers %>%
  check_that(turnover >= 0, staff >= 0) %>%
  summary()
##   rule items passes fails nNA error warning                expression
## 1   V1    60     56     0   4 FALSE   FALSE (turnover - 0) >= -1e-08
## 2   V2    60     54     0   6 FALSE   FALSE    (staff - 0) >= -1e-08
```

For more involved checks, it is more convenient to define a validator object first.

```
v <- validator(turnover >= 0, staff >= 0)
retailers %>% confront(v) %>% summary()
##   rule items passes fails nNA error warning                expression
## 1   V1    60     56     0   4 FALSE   FALSE (turnover - 0) >= -1e-08
## 2   V2    60     54     0   6 FALSE   FALSE    (staff - 0) >= -1e-08
```

6.5.3 Raising Errors or Warnings

It may occur that rules contain mistakes such as spelling mistakes in variables. In such cases, a rule cannot be evaluated with the data for which it is aimed. By default, `validate` catches errors and warnings raised during a confrontation and stores them. The number of errors and warnings occurring at a confrontation is reported when a confrontation object is printed to screen. As an example, let us specify a check on a variable not occurring in the `retailers` dataset.

```
v <- validator(employees >= 0)
cf <- confront(retailers, v)
cf
## Object of class 'validation'
## Call:
##     confront(x = retailers, dat = v)
##
## Confrontations: 1
## With fails    : 0
## Warnings      : 0
## Errors        : 1
```

The actual error message can be obtained with the command `errors(cf)`. Alternatively, one may raise errors and/or warnings immediately by passing the `raise` option. Using

```
confront(retailers, v, raise="error")
```

will raise errors (so stop if one has occurred) but catch warnings. Using

```
confront(retailers, v, raise="all")
```

will also print warnings as they occur.

6.5.4 Tolerance for Testing Linear Equalities

Testing linear balance equations of the form

$$\sum_i a_i x_i = b,$$

in a strict sense may trigger a lot of false violations. When the numeric values a_i, x_i, or b result from earlier calculations, machine roundoff errors may cause differences from equality on the order of 10^{-16}, a precision that is almost never achieved in (physical) measurements. It therefore stands the reason to ignore such roundoff errors to a certain degree. In the `validate` package this is achieved by interpreting such linear checks as

$$\left|\sum_i a_i x_i - b\right| < \varepsilon$$

with ε a small positive constant.

The value of this constant equals 10^{-8} by default. This is a commonly used value that is close to the square root of the maximum precision of IEEE double precision numbers. It can be altered by passing the option `lin.eq.eps` to `confront`.

```
v <- validator(x+y==1)
d <- data.frame(x=0.5,y=0.50001)
summary(confront(d,v))
##   rule items passes fails nNA error warning             expression
## 1   V1     1      0     1   0 FALSE   FALSE abs(x + y - 1) < 1e-08
summary(confront(d,v,lin.eq.eps=0.01))
##   rule items passes fails nNA error warning            expression
## 1   V1     1      1     0   0 FALSE   FALSE abs(x + y - 1) < 0.01
```

We warn the reader that the value of `lin.eq.eps` is not aimed at allowing for tolerances that are caused by roundoff errors that are on the order of the unit of measurement. Since such cases are artifacts of the measurement, it is better to keep track of them by explicitly defining rules in the form of (in)equations. Machine rounding errors can be expected to be of a similar order of magnitude for most of the data (i.e., very small), while rounding errors at measurement are of the order of magnitude of the unit of measurement, which may differ per variable.

As an example, consider a business survey asking for total revenue r, total costs c, and total profit p, rounded to thousands. We have the rule

$$p + c = r.$$

Suppose that the actual values are ($p = 5600, c = 8700, r = 14{,}300$). If we round off the individual variables, we get (6000, 9000, 14,000), which does not add up. To handle such cases, one should define the rule

$$|p + c - r| < 2000,$$

or, alternatively (assuming or demanding that $p, c, r \geq 0$)

$$\begin{aligned} p + c - r &< 2000 \\ -p - c + r &< 2000, \end{aligned}$$

where the latter form has advantages for further automatic processing such as error localization or performing consistency checks.

6.5.5 Setting and Resetting Options

In the previous two sections, we demonstrated how to set options that are valid during execution of a particular confrontation. The same option can be set for a particular `validator` object. By setting

```
voptions(v, raise="all")
```

every time `v` is confronted with a dataset, all exceptions (errors and warnings) are raised immediately. By setting the global option

```
voptions(raise="all")
```

all confrontations will raise every exception, except if the `validator` object in question has a specific option value set.

The options set globally or for a specific `validate` object can be queried using the same function.

```
# query the global option setting (no argument prints all options)
voptions("raise")
## [1] "all"
# query settings for a specific object
voptions(v,"raise")
## [1] "all"
```

Options can be restored to their defaults with the `validate_reset` function. Use

```
validate::reset()
```

To reset global options to their default or

```
validate::reset(v)
```

to reset the options of `v` to the defaults.

Summarizing, the `validate` package has three levels of options.

- Options defined at the *global level* are used in all confrontations for which no specific options are set.
- Options defined at *object level* are used in all confrontations in which that object is used and locally overrule the global options.
- Options defined at *function call level* are used only for that specific function call and locally overrule the options at global and object levels.

6.5.6 Importing and Exporting Validation Rules from and to File

In production environments, it is a good practice to manage rules separately from the code executing it. This is why the validate package can import rules and settings from text files. The file-processing engine of `validate` supports easy definition of rule properties such as `name`, `label` and `description`, and file inclusion. Rules can be defined simply in free format, allowing for based comments (preceded by #) or in a structured format allowing for specification of rule properties and options.

The simplest way to specify rules in a file is simply stating them in free format. For example, here is a small text file stating a few rules for the `retailers` dataset.

```
# rules.txt

# nonnegativity rules
turnover >= 0
staff >= 0

# According to Nancy the following
# statement must hold for every retailer
turnover / staff > 1
```

Here, comments are used to describe the rules. The rules can be read by specifying the file location as follows:

```
v <- validator(.file="rules.txt")
```

In this case, all options are taken from the global options rule properties, such as names are generated automatically.

To specify properties for each rule, `validate` relies on the widely used `YAML` format (yaml.org 2015), an example of which is given in this format is used for exchanging structured data in a way that is much more human-readable (and human editable) than, for example, `JSON` or `XML`. In fact, `YAML` is compatible with `JSON` in the sense that the former is a superset of the latter: every `JSON` file is also a valid `YAML` file (the opposite is not true). Before discussing the yaml format for `validate`, we give a few tips on editing yaml files.

- Files in `YAML` format usually have the `.yaml` extension. This makes it easy for text editors to recognize such files and load the appropriate indentation and highlighting rules.
- Indentation is important. Like `python`, `YAML` uses indentation to identify syntax elements.
- Never use `tab` for indentation. The `tab` is not a part of the `YAML` standard. Most text editors or development environments allow you to turn tabs into spaces automatically when typed.
- The exclamation mark is part of the `YAML` syntax. Enquote expressions (with single quotes) when it includes an exclamation mark.

In Figure 6.2, an example rule definition in `YAML` format is shown. Comments are again preceded with a `#`. One starts defining a rule set with the keyword `rules` followed by a colon and a newline. Each rule is preceded by a dash (`-`). Here, we follow each dash by a new line to make the difference between rules more clear. Next, the rule elements are defined with key-value pairs, each on a new line and indented at least one space, in the form `[key] : [value]`. If a key is not specified, it will be left empty or filled with a default upon reading. In the example, two spaces are used for indentation to enhance readability and to better see the difference when multiple indentations are used. For the `description` field it may be desirable to have a multiline entry. This can be achieved by preceding the entry with a | (vertical bar) or > and starting the entry on a new line, indented doubly. Reading the file shown in Figure 6.2 is as easy as reading a free-form text file.

```
# yamlrules.yaml
rules:
-
  expr: turnover + other.rev == total.rev
  name: inc
  label: income balance
  description: Income posts must add up to total revenue.
-
  expr: if ( staff > 0 ) staff.costs > 0
  name: stf
  label: staff/cost sanity
  description: |
    It is not possible for a retailer to employ staff
    without having any staff costs. Contact Nancy for
    details on this rule.
```

Figure 6.2 Defining rules and their properties in `YAML` format.

```
v <- validator(.file="yamlrules.yaml")
```

Options for the validator object created when reading a file and inclusion of other files are achieved by including a header of the following form:

```
---
include:
  - child1.yaml
  - child2.yaml
options:
  raise: all
---
rules:
# start rule definitions here
```

This block has to start and end with a line containing three dashes (---). Note that each include entry is indicated with an indented dash, followed by a file location, and that the options are listed as indented [key]: [value] pairs.

The path to included files may be given relative to the path where the including file is stored or as a full file path starting from the system's root directory. Files may be included recursively (an included file may include other files), and the order of reading them is determined by a topological sort. That is, suppose that we have the following situation:

- top.yaml includes child1.yaml and child2.yaml.
- child2.yaml includes child3.yaml

The order of reading is determined by going through the inclusion lists from top to bottom and for each element of each list, reading the inclusion stack top to bottom. In the above example, this implies the following reading order.

```
child1.yaml, child3.yaml, child2.yaml, top.yaml
```

Such inclusion functionality can be used, for example, when a general set of rules applies to a full dataset, but there are specific rules for certain subsets (strata) of the dataset. Subset-specific additional rules may be defined in a file that includes another file with the general rules.

There is a second use of the dashes, namely, it allows users to use both the structured and the free form for rule definitions in the same file. An example of what such a file could look like is given in Figure 6.3. The section with structured rules may in principle be followed again by a section with free-form rules and so on.

Validator objects can be exported to file with export_yaml:

```
export_yaml(v, file="myfile.yaml")
```

or one can create the yaml string using as_yaml.

```
str <- as_yaml(v)
```

In the latter case, the string str can be written to file or any other connection accepting strings (tip: use cat(str) to print the string to screen in a readable format). To translate R objects from and to YAML format, validate depends on the R

```
---
options:
  raise: all
---
# some free form rules
turnover >= 0
staff >= 0
---
# structured rules
rules:
-
  expr : if (staff > 0) staff.costs > 0
  label: staff/cost sanity
```

Figure 6.3 Free-form and YAML-structured rule definitions can be mixed in a single file.

package yaml of Stephens (2014). Both export_yaml and as_yaml accept optional arguments that are passed to yaml::as.yaml, which is the basic function used for translation.

6.5.7 Checking Variable Types and Metadata

Checking whether a column of data is of the correct type is easy with validate. Relying only on the basic definition of validation, one can define rules of the form

```
class(x) == "numeric"
## [1] FALSE
class(x) %in% c("numeric", "complex")
## [1] FALSE
```

However, to make life easier, all R functions starting with is. (is-dot) are also allowed (it is thus assumed that all such functions return a logical or NA). As a reminder, we list the basic type-checking functions available in R.

```
is.integer   is.factor
is.numeric   is.character
is.complex   is.raw,
```

Within a validation rule it is possible to reference the dataset as a whole, using the .. For example, to check whether a data frame contains at least 10 rows, do

```
check_that(iris, nrow(.) >= 10) %>% summary()
##   rule items passes fails nNA error warning    expression
## 1   V1     1      1     0   0 FALSE   FALSE nrow(.) >= 10
```

As a second example, we check that the fraction of missing values is below 20%.

```
data("retailers")
check_that(retailers, sum(is.na(.))/prod(dim(.)) < 0.2) %>%
summary()
##   rule items passes fails nNA error warning
## 1   V1     1      1     0   0 FALSE   FALSE
##                               expression
## 1 sum(is.na(.))/prod(dim(.)) < 0.2
```

6.5.8 Checking Value Ranges and Code Lists

Most variables in a dataset will have a natural range that limits the values that can reasonably be expected. For example, a variable recording a person's age in years can be stored as a `numeric` or `integer`, but we would not expect it to be negative or larger than say, 120. For numerical data, range checks can be specified with the usual comparison operators for numeric data: <, <=, >= and >. This means that we need two rules to define a range. In the example `age >= 0` and `age <= 120`.

The range for categorical (`factor`) or `character` data can be verified using R's `%in%` operator. For example, the rule

```
gender %in% c('male','female')
```

checks whether the `gender` variable is in the allowed set of values. Although it is better to separate data cleanup from data validation, it is possible to combine this with text normalization:

```
tolower(gender) %in% c('male', 'female')
```

or with fuzzy matching.

```
stringdist::ain(gender, c('male','female'), maxDist=2) == TRUE
```

In the latter example, we check whether values in *gender* are within two typos of the allowed values (see Section 5.4).

Code lists (lists of valid categories) can be lengthy, and it is convenient to define them separately for reuse. In `validate`, the `:=` operator can be used to store variables for the duration of the validation procedure. Let us create some example data and a rule set to demonstrate this.

```
# create a data frame
d <- data.frame( gender = c('female','female','male','unknown'))
# create a validator object
v <- validator(
  gender_codes:= c('female', 'male') # store a list of valid codes
  , gender %in% gender_codes         # reuse the list.
)
# confront and summarize
confront(d,v) %>% summary()
##   rule items passes fails nNA error warning
## 1   V2     4      3     1   0 FALSE   FALSE
##                             expression
## 1 gender %in% c("female", "male")
```

From the summary we see that the confrontation between data and rules yielded a single validation. The actual expression that was evaluated during the confrontation is generated by `confront` by substituting the right-hand side of the `:=` operator in the rule.

6.5.9 Checking In-Record Consistency Rules

In-record validation rules are among the most common and best-studied rules. With in-record validation rules we understand rules whose truth value only depends on a single record (in a single table) but possibly on multiple values within that record. The truth value of an in-record rule therefore does not change when a dataset is extended with extra records. In `validate`, in-record rules yield a truth value for each record in the dataset confronted with it. The range and type-checks discussed above are special cases of in-record rules that pertain to a single variable only.

There are three classes of rules that have been widely studied because they allow for automated error localization: linear equalities and inequalities, conditional rules on categorical data, and conditional rules where at least one of the conditions and the consequent contain linear (in)equalities. Although `validate` can handle arbitrary in-record validation rules, we will highlight examples from these three classes while emphasizing the role of precision in evaluating linear equality (sub)expressions.

Linear equalities or inequalities occur frequently in economic data, subject to detailed balance rules. We have encountered examples for the `retailers` dataset where we defined rules such as the following:

```
v <- validator(
   turnover + other.rev == total.rev
  , other.rev >= turnover
)
```

By default, `validate` will check linear equalities to a precision that is stored in the option `lin.eq.eps` of which the default value is 10^{-8}. One can note this by inspecting the expressions that were actually used to perform the validation.

```
cf <- confront(retailers,v)
summary(cf)['expression']
##                                          expression
## 1 abs(turnover + other.rev - total.rev) < 1e-08
## 2                  (other.rev - turnover) <= 1e-08
```

Since the variables in the `retailers` dataset are all rounded to integers, we may force a strict equality check by setting the option `'lin.eq.eps'` to zero. In that case the equality restriction is evaluated as is.

Multivariate rules on categorical or textual data can often be written in a conditional form. For example, *if gender is male, then pregnant must be* `false`. In the validate package, such rules can be specified as

```
if ( gender == "male") pregnant == FALSE
```

The above rule will be interpreted as the logical implication

$$gender = \text{male} \Longrightarrow pregnant = \texttt{false},$$

which is defined by the usual truth table shown below.

$gender = \text{male}$	$pregnant = \texttt{false}$	$gender = \text{male} \Longrightarrow pregnant == \texttt{false}$
true	true	true
true	false	false
false	true	true
false	false	true

When confronted with data, such a rule is translated to

```
!(gender == "male") | pregnant == FALSE
```

using the implication replacement rule from elementary logic ($P \implies Q$ equals $\neg P \vee Q$). The reader may confirm this by inspecting the output of the following code:

```
v <- validator(if(gender == "male") pregnant == FALSE)
d <- data.frame(gender = "female", pregnant = FALSE)
summary(confront(d,v))
```

For rules combining linear and nonlinear or conditional tests, all subexpressions that are linear equalities are checked within the precision defined in `lin.eq.eps`. As an example, let us define a validator and a record of data.

```
v <- validator(test = if ( x + y == 10) z > 0)
d <- data.frame(x = 4, y = 5, z = -1)
```

Using the default setting for `lin.eq.eps`, the condition `x + y == 10` evaluates to `abs(5 + 4 - 10) < 1e-8` (hence, `FALSE`). By inspecting the truth table, we see that the rule in the consequent, `z > 0`, does not need to be satisfied, and hence the data satisfies the condition.

```
values(confront(d,v))
##      test
## [1,] TRUE
```

If we set the precision parameter to a large enough value, the condition in the rule is satisfied, and the confrontation evaluates to another value.

```
values(confront(d,v,lin.eq.eps=2))
##       test
## [1,] FALSE
```

6.5.10 Checking Cross-Record Validation Rules

Contrary to in-record validation rules, cross-record rules have the property that their truth values may change when the number of records in a dataset is increased or decreased. The defining property for cross-record rules is that values from multiple records are needed to evaluate them. In `validate`, they may yield a value for each record or a single truth value for the whole dataset.

As an example of a rule that evaluates to a single value, consider the rule that states our expectation that there is a positive covariance between two variables.

```
v <- validator(cov(height,weight) > 0)
values(confront(women,v))
##        V1
## [1,] TRUE
```

Obviously, multiple records are necessary to compute the covariance, and the value of covariance is likely to change when records are added or removed.

Cross-record rules that yield a truth value for each record in the dataset often take the form of a check for outliers. For example, we may check that values in a column of data do not exceed the median plus 1.5 times the interquartile range.

```
v <- validator( height < median(height) + 1.5*IQR(height))
cf <- confront(women,v)
head(values(cf),3)
##         V1
## [1,] TRUE
## [2,] TRUE
## [3,] TRUE
```

Again, the individual truth values may change when records are added or removed: the median to which individual *height* values are compared depends on all available records.

Even though a truth value is obtained for each record, one can not be sure that a record that gets labeled `false` is the location of the error. It may still be that other records used in computing the interquartile range or the median contain flaws. Naturally, when the input data table is sufficiently large, and the aggregates can be computed robustly and with enough precision, it is likely that records mapping to `false` do contain the error. It should be emphasized however that this is an extra assumption that deserves to be checked.

6.5.11 Checking Functional Dependencies

Functional dependency checks are a special class of cross-record validation rules that can be used to track conflicting data across records. An example of a functional dependency is the statement that 'if two persons have the same zip code, they should live in the same city'. The concept of functional dependencies was first introduced by Armstrong (1974) as a tool and mathematical model for describing and designing databases. Given a data table with columns $\{A, B, \ldots\}$, the functional dependency

$$A \rightarrow B$$

expresses that 'if two records have the same value for A, they must have the same value for B'. For example, to express that a married couple must have the same year of marriage, one could write

$$\textit{Marriage registration number} \rightarrow \textit{Marriage year},$$

given that these data are stored in the same table. The notation can be extended to express relations between more than two variables. For example,

$$A, B \rightarrow C$$

expresses that 'if two records have the same value for A and for B, then they must have the same value for C. A practical example is the relation

$$\textit{city}, \textit{street} \rightarrow \textit{zipcode}. \tag{6.5}$$

It is also possible to have multiple variables on the depending side, so

$$\textit{zipcode} \rightarrow \textit{city}, \textit{street}$$

is also a valid functional dependency.

Functional dependencies can thus be interpreted as checks to find out whether certain contradictions appear in a data table. In `validate`, functional dependencies are expressed with the ~ (tilde) operator. Variables on the left- or right-hand side can be combined using the + operator. This means that the rule of Eq. (6.5) can be defined as follows.

```
v <- validator(city + street ~ zipcode)
```

Confronting a data table with a functional dependency yields a `logical` vector with an entry for each record. If the ith value of the output is `TRUE`, this means that that record does not conflict with any of the $i - 1$ records before it. If the ith value is `FALSE`, at least one conflict with an earlier record is detected. As an example, we confront the above rule with a few records.

```
d <- data.frame(
  street = rep("Spui",4)
  , city   = c("The Hague", "The Hague","Amsterdam", "The Hague")
  , zipcode= c(2511,2513,2511,2511)
)
cbind(d, fd_value=values(confront(d,v))[,1])
##   street      city zipcode fd_value
## 1   Spui The Hague    2511     TRUE
## 2   Spui The Hague    2513    FALSE
## 3   Spui Amsterdam    2511     TRUE
## 4   Spui The Hague    2511     TRUE
```

Both the cities, Amsterdam and The Hague, have a street named 'Spui', each with its own zip code. The functional dependency check works by going through the records top to bottom, and testing whether there is a conflict with an earlier record. At the first record, obviously no conflict is found since the street–city and zip code combination is unique. The second record has the same street and city name as the first record, but the zip code is different. Hence, it conflicts with the first record. The third record has the same street and zip code as the first record, but not the same city. Hence, there is no conflict with the functional dependency from Eq. (6.5). The last record has the same values for street, city, and zip code as the first and therefore does not conflict.

Observe that the order of records matters for the outcome. Combinations of variables on the left-hand side of the functional dependency are matched against their first unique occurrence. If we were to switch the order of the first and second records, the number of conflicts would increase from one to two. To make the output order independent, we could reduce the output dimension and define the rule as

```
all(street + city ~ zipcode)
```

It is difficult to automatically correct for violations of functional dependencies, especially when multiple interrelated functional dependencies (and other rules) are involved. Even pointing out the erroneous variable or variables and record(s) can be a daunting task. For example, in the toy example described above there are several ways to resolve the conflict. One can alter the zip code of the second record, but altering the city or street of the same record would resolve the conflict as well, or one could alter the city, street, or zip code in the first record, although that could trigger a conflict elsewhere. The number of possibilities to resolve the conflict grows fast if one also allows multiple changes on multiple records.

6.5.12 Cross-Dataset Validation

Cross-dataset validation involves comparisons of the data under scrutiny with (functions of) one or more external datasets. As a first example, we validate the heights in the `women` dataset against the height of Dutch women of the same age (30–39 years [1]). We validate that the height average height of American women does not differ more than 10% from that of Dutch women.

```
  dutch_women <- data.frame(heightCM = 176.2)
  v <- validator(
   inch:= 1/2.54
   , us_mean:= mean(height)
   , upplim = us_mean < 1.1 * ref$heightCM/inch
   , lowlim = us_mean > 0.9 * ref$heightCM/inch
  )
  summary(
    confront(dat = women,x=v, ref = dutch_women)
  )[1:5]
  ##     rule items passes fails nNA
  ## 1 upplim     1      1     0   0
  ## 2 lowlim     1      0     1   0
```

For the data under scrutiny, we may use variable names directly while the reference data must be indexed with the `$` operator. The default name of the reference data is `ref`. This can optionally be changed, but the reference data must be passed in a `list`.

```
  v <- validator(
   inch:= 1/2.54
   , us_mean:= mean(height)
     # we use 'dw' in stead of 'ref' now.
   , upplim = us_mean < 1.1 * dw$heightCM/inch
   , lowlim = us_mean > 0.9 * dw$heightCM/inch
  )
   # the reference data must be named correspondingly when call-
ing 'confront'
  cf <-  confront(women, v, ref=list(dw = dutch_women))
```

In fact, it is possible to pass multiple reference datasets to confront, by storing them in a named `list`, as shown above, or in an `environment`.

```
  # a reference environment in which we may store reference
datasets or variables
  refdat <- new.env()
  # the name of the reference data is 'dw' in the reference envi-
ronment
  refdat$dw <- dutch_women
  cf <- confront(women,v, ref=refdat)
```

It is also possible to do record-wise comparisons between datasets. If no `key` argument is specified, it is assumed that the records in the data being validated match exactly with the records in the reference data.

```
  v <- validator(weight == ref$weight)
  cf <- confront(women,v, ref=women)
```

[1] Statistics Netherlands, 2014.

When a `key` is specified, records in the reference data are sorted to match against the validated data. Records with keys that occur in the reference data but not in the validated data are left out. Records that occur in the validated data but not in the reference data are added as empty records to the reference data.

6.5.13 Macros, Variable Groups, Keys

The `validate` syntax supports a few features that make rule definition and reuse of statements easier. The first one, based on the `:=` operator was briefly introduced in Section 6.5.8. The effect of adding a statement like `A:= [expression]` is that upon confrontation with data, in every rule after this macro occurrences of a variable `A` is substituted by `[expression]`. This is also true if the macro is defined in a file that is included by another file. So for example, the sequence of rules

```
m := mean(x, na.rm=TRUE)
y < m
z < m
```

is equivalent to

```
y < mean(x, na.rm=TRUE)
z < mean(x, na.rm=TRUE)
```

The second feature can be used to apply the same rule to multiple variables, after defining a variable group. For example, the rule

```
var_group(a,b) > 0
```

will be expanded to

```
a > 0
b > 0
```

Variable group definitions may be combined with macros. The following is a more typical example of using variable groups.

```
mygroup := var_group(a,b)
mygroup > 0
```

When multiple variable groups are used in a single statement, the number of resulting statements is equal to the size of the Cartesian product of the variables in the variable groups. So the rule

```
var_group(a,b) > var_group(c,d)
```

is equivalent to

```
a > c
a > d
b > c
b > d
```

6.5.14 Analyzing Output: `validation` Objects

Depending on the number of validation rules specified, the amount of output of a confrontation of data with a set of rules can be very large. There are several ways to aggregate results so they become more understandable and manageable.

We already discussed the summary method applied to the result of a call to confront. After confronting data with a validator object, the summary function lists for each validation rule, the number of datasets that result in pass, fail, or NA, whether the rule could be executed or resulted in an error (or warning), and the actual R expression that was executed to evaluate the rule.

We usually expect that none of our rules result in an error, and all rules evaluate to true, false, or NA. In-record rules result in a value for each record in the dataset, so for each rule, the following aggregates are obvious indicators of data quality:

- the number of records for which a rule evaluated to true;
- the number of records for which a rule evaluated to false;
- the number of records for which a rule evaluated to NA.

On the other hand, for each record, we may compute the following:

- the number of rules that evaluated to true;
- the number of rules that evaluated to false;
- the number of rules that evaluated to NA.

The above statistics can be computed with the aggregate function. As an example, consider the following validation rules on for the retailers dataset.

```
v <- validator(
  other.rev > 0
  , turnover > 0
  , total.rev > 0
  , staff.costs > 0
  , total.costs > 0
  , turnover + other.rev == total.rev
)
```

After confronting the data with the rules, we may aggregate the results either rule-wise or record-wise using the aggregate function. We add a key to the retailers dataset so that records can be recognized after sorting.

```
retailers$id <- paste0("r-",1:nrow(retailers))
cf <- confront(retailers,v,key='id')
# rule-wise aggregation
aggregate(cf)
##    npass nfail nNA  rel.pass   rel.fail     rel.NA
## V1    23     1  36 0.3833333 0.01666667 0.60000000
## V2    56     0   4 0.9333333 0.00000000 0.06666667
## V3    58     0   2 0.9666667 0.00000000 0.03333333
## V4    50     0  10 0.8333333 0.00000000 0.16666667
## V5    55     0   5 0.9166667 0.00000000 0.08333333
## V6    19     4  37 0.3166667 0.06666667 0.61666667
```

Note that the row names of the output data.frame indicate the names of the validation rules. Record-wise aggregation can be achieved as follows:

```
head(aggregate(cf, by='record'),n=3)
##     npass nfail nNA  rel.pass  rel.fail    rel.NA
## r-1     2     0   4 0.3333333 0.0000000 0.6666667
## r-2      4      0    2 0.6666667 0.0000000 0.3333333
## r-3      4      2    0 0.6666667 0.3333333 0.0000000
```

Here, the row names correspond to the key that was specified in the call to confront.

For efficient data cleaning, it is often beneficial to find records or rules that result in the most fails. The sort function does the same as aggregate but also sorts the results, putting items with the least passes on top.

```
# by default aggregation is over rules:
sort(cf)
##    npass nfail nNA  rel.pass   rel.fail     rel.NA
## V6    19     4  37 0.3166667 0.06666667 0.61666667
## V1    23     1  36 0.3833333 0.01666667 0.60000000
## V4    50     0  10 0.8333333 0.00000000 0.16666667
## V5    55     0   5 0.9166667 0.00000000 0.08333333
## V2    56     0   4 0.9333333 0.00000000 0.06666667
## V3    58     0   2 0.9666667 0.00000000 0.03333333
```

So here, V6 (the balance rule) was satisfied the least: only 19 records satisfy this rule. Note that the low number of passes is also due to the high number of NA's. That is, in 37 cases, the validation rule could not be checked because not all necessary values were available in the record.

Sorting over records puts the records that pass the least number of validation rules on top.

```
head(sort(cf,by='record'), n=3)
##       npass nfail nNA  rel.pass rel.fail    rel.NA
## r-10      0     0   6 0.0000000        0 1.0000000
## r-1       2     0   4 0.3333333        0 0.6666667
## r-15      2     0   4 0.3333333        0 0.6666667
```

The order of sorting can be set using the decreasing argument.

When confronting a set of k in-record validation rules with n records, the result is an $n \times k$ matrix taking values in {true, false, NA}. When so desired, this matrix can be obtained with the values function. This function extracts the raw results from a validation objects and combines them conveniently into a logical array. Below, we only show the first three rows for compactness.

```
head( values(cf), n=3 )
##         V1   V2   V3   V4   V5    V6
## r-1     NA   NA TRUE   NA TRUE    NA
## r-2     NA TRUE TRUE TRUE TRUE    NA
## r-3 FALSE TRUE TRUE TRUE TRUE FALSE
```

Not all validation rules yield record-wise results and will therefore produce output of different dimensionality. Dealing with data structures of different dimensionality is a nuisance in the sense that it almost always requires some detailed programming by users for all cases to be handled properly. The next section discusses how values and aggregates with different dimension structures are handled in the validate package.

6.5.15 Output Dimensionality and Output Selection

Typically, rules with two types of output dimension structures are encountered: rules that yield a vector of results, one for each record in a dataset, and rules that yield a single result. The validation object stores these results for each rule with which a dataset is confronted. If an error or warning occurred, these are stored (by default) as well.

The `values` function gathers these results while skipping errors and combines results with equal dimension structure into an array, much like R's `sapply` does. When multiple dimension structures are encountered, a `list` of arrays is generated. By default, if only a single dimension structure is encountered, an array is returned rather than a `list`, but this behavior can be switched off by passing the argument `drop=FALSE` to `values`. Both `sort` and `aggregate` accept a `drop` argument as well.

To avoid handling lists, one may choose to separate the types of rules and divide them over several validation objects or, using square brackets, one may select the output from a `validation` object. As an example we consider the following rules on the `women` dataset.

```
v <- validator(
  height > 0          # record-wise rule
  , sum(weight) > 0   # sum-rule
)
cf <- confront(women[1:3,],v)
```

Since the results cannot be combined into a single array, `values` returns a `list` with two arrays.

```
values(cf)
## [[1]]
##        V1
## [1,] TRUE
## [2,] TRUE
## [3,] TRUE
##
## [[2]]
##        V2
## [1,] TRUE
```

Alternatively, we may choose to extract only the values from the second rule like so

```
values(cf["V2"])
##        V2
## [1,] TRUE
```

Here, we used the name of the second rule as index, but one may use any type of index (`integer`, `logical`) common to base R objects.

Exercises for Section 6.5

Exercise 6.5.4 *Write a free-format text file with about 10 rules for the* `retailers` *dataset. Read the file using* `validator`*; export the file to a yaml file; fill in the missing fields for* `name`, `label`, *and* `description`*; reread the file and inspect the contents of the* `validator` *object using* `names`, `labels`, `description`, `summary`, *and so on.*

Exercise 6.5.5 *The street called 'Spui' in Amsterdam does not have the same zip code as the street of the same name in The Hague. Implement a functional dependency that reveals this as a conflict.*

It is a basic design principle behind `validate` that rules are manipulated only when necessary. This means that rules defined with variable groups are only expanded when confronted with data.

Exercise 6.5.6 *Define a validator object containing nonnegativity rules for all numerical variables in the* `retailers` *dataset. Confront this validator with the data. Create a summary to see which expressions were evaluated. Also note the name of the rules before and after expansion.*

Exercise 6.5.7 *Both the variables,* `total.rev` *and* `turnover` *of the* `retailers` *dataset, must be larger than or equal to the number of* `staff` *and the amount of* `profit`*. Define two groups and a single inequality rule that expresses all these rules.*

7

Localizing Errors in Data Records

7.1 Error Localization

Data can be invalidated with validation rules. If data is marked as invalid, what are the subsequent actions one should take? One possible action is to remove all invalid observations from the data. Although this may seem a sound advice, often this has severe consequences: removing all observations that contain invalid values may lead to the following:

- remove many valid and useful values,
- leave too little data to be analyzed,
- introduce selection bias in the resulting data.

Removing observations with invalid data may result in a dataset that is not representative. In many cases, the erroneous records are very relevant for the research question at hand. In statistical practice, such observations often describe difficult-to-measure subpopulations that differ from the rest of the the population and are of interest for policy makers or from a scientific point of view. Examples of interesting populations in social statistics where survey data often suffer from incomplete and erroneous observations are elderly, migrants, foreign workers, students, and homeless people. Examples in economic data are startups, flex workers, and customer-to-customer sales. Removing such records can heavily bias the statistical estimates that follow. As an extreme example, suppose that every case contains a single invalid or missing value. Removing each invalid case would then result in no data at all, thereby wasting all the observed values that are valid.

An alternative for removing invalid data is to "correct" the invalid data through data editing so that the edited data can be used in the data analysis. Changing (scientific) data for further analysis may seem wrong but often is the lesser of the two evils: having no or heavily biased data versus minimally edited data. Moreover, in some cases, the measurement error causing violation of the validation rules can be reconstructed or properly estimated. In this sense, the whole editing process can be viewed as part of the estimation procedure (van der Loo *et al.*, 2017).

The overall data-editing strategy we follow is to detect invalid data, determine which values are considered faulty, and remove those values so that they can be imputed in later steps. This procedure thus generates missing values, and they should be chosen so that these can be imputed in a way so that no rule violations remain. As a guiding principle, we therefore demand that the original data is altered as little as possible while ensuring that the data is no longer invalidated by validation rules.

The ability to correct values requires that data errors can be identified. The procedure to find errors is referred to as *error localization*: which values of the data are causing invalidation and are therefore considered to be errors?

Definition 7.1.1 *Error localization is a procedure that points out fields in a dataset that can be altered or imputed in such a way that all validation rules can be satisfied.*

An error localization procedure must thus search the space of field combinations to find those combinations of fields that can be altered to satisfy a set of validation rules. The space of possibilities is large, and in principle, the solution need not be unique. Most error localization procedures are restricted to classes of rules that can be evaluated for each case (record) separately. That is, to check whether a case contains valid data, one only needs access to the data in that particular record. No reference to other records or datasets is necessary. Since there are less fields to consider, this restriction on the ruleset makes the problem of error localization more manageable. Moreover, error localization procedures are aimed at minimizing the (weighted) number of fields to adapt, to stay as close to the original data as possible. To clarify this, let us have a look at some examples.

For univariate rules, error localization is trivial: if an observation is invalidated by an univariate validation rule, the value of the corresponding variable is invalid.

Example 7.1.2 **(Univariate rule)** *In a survey, we encounter the following observation:*

Name	***Age***
John	*−10*
Peter	*500*

The negative age of John is clearly an error. The rule set should include a validation rule stating that "age must be positive". Peter's age is also problematic, while in theory not impossible, it is highly implausible: the set should also include a rule that forbids ages higher than 130. Hence, our rule set could look like this.

$$age \geq 0$$

$$age \leq 130$$

If any rule is violated, age *must have an invalid value.*

Univariate rules are variable domain definitions: they define what values are allowed. For numeric variables, univariate rules are defined as one or more ranges, and for categorical variables they are defined as the set of allowed discrete values.

Error localization in a multivariate validation rule is more complex: in principle, any combination of values can be invalid.

Example 7.1.3 (Multivariate rule) *Let us look at respondent Muriel:*

Name	Age	Married
Muriel	6	Yes

and we have the following validation rule:

if a person is married, *then* age ≥ 16

At least one of the two, Married *or* Age, *is invalid, because both valid cannot be true.*

This example illustrates that solutions to error localization problems are in general not unique. We can make the data satisfy the validation rule by changing the value of either *Age* or *Married* (or both). It is sometimes possible to leverage extra information. For example, if a person's date of birth is known from another reliable source, then this can be used to point out the field that most likely contains the error.

Things become more complicated still when multiple rules are involved that pertain to an overlapping set of variables. In that case, changing a value to satisfy one rule may cause the violation of another.

Example 7.1.4 (Multivariate rule sets) *We have observed that Eric is 3 years old, married, and attends kindergarten.*

Name	Age	Married	Attends
Eric	3	Yes	Kindergarten

and we have the following validation rules:

if a person is married, *then* age ≥ 16

if a person attends kindergarten, *then* age ≤ 6

Which of the values is correct? If we change Age *to a value that does not invalidate the first rule, the second rule is invalidated. So either* Married *is incorrect, or both* Age *and* Attends.

This example shows that an error localization procedure has to take all validation rules into account, including the rules that are not violated.

These examples motivate the following definition of the error localization procedure in terms of an optimization problem.

Definition 7.1.5 (Principle of Fellegi and Holt) *Given a record subject to m validation rules, of which $m' \leq m$ are violated, find the smallest (weighted) subset of variables that can be altered such that the record satisfies all validation rules.*

This minimization problem is named after (Fellegi and Holt, 1976), who first formulated and solved the problem for the case of categorical data under logical restrictions.

Assigning a weight to each variable allows domain experts to influence which variables are more likely to be part of a solution. Since variables with a higher assigned weight are less likely to end up in the optimal solution, the weights are often referred to as 'reliability weights'. Domain experts can assign higher reliability weights to variables they deem to have higher quality.

Over time, several algorithms have been developed for solving the error localization problem. An extensive overview of the theory is given in de Waal *et al.* (2011, Chapters 3–5). On one hand, there are specialized approaches that explicitly manipulate the rule set such as the branch-and-bound algorithm of de Waal and Quere (2003) and the original method of Fellegi and Holt (1976). On the other hand, there are approaches that translate the problem to a well-known general optimization problem (mixed-integer programming (MIP)). Some specialized approaches that focus on a single data type and a single type of rules exist as well.

In this chapter, we focus on the MIP approach. The advantage of a MIP approach is that it allows one to leverage existing, proven implementations of MIP solvers, either commercial or open source. The main downside of the MIP approach compared to the branch-and-bound algorithm is that users have less fine-grained influence on the chosen solution. Indeed, MIP solvers often return just one of the optimal solutions when multiple equivalent solutions exist, while implementations of the branch-and-bound method often return all solutions.

In the rest of this chapter, we first describe the usage of the `errorlocate` package. This section is then followed by a description of the translation of the error localization problem into a MIP-problem for the linear, categorical, and conditional validation rules. We end the chapter with describing the numerical stability of the implemented algorithm and give suggestions on handling potential issues.

7.2 Error Localization with R

7.2.1 The Errorlocate Package

The `errorlocate` package offers a small set of functions that allows users to localize errors in records that are to be validated by a set of linear, categorical, or conditional validation rules using a fast implementation of the Fellegi–Holt paradigm. Erroneous values can be automatically replaced with `NA` or another chosen value. The package is extensible by allowing users to implement their own error location routines. The package integrates with the `validate` package, which is used for rule definition and manipulation.

In `errorlocate`, three types of rules can be treated. First, linear rules express linear (in)equality conditions on numerical variables. Examples include (account) balance equations or nonnegativity restrictions. Second, categorical rules express conditional relations (logical implications) between two or more categorical variables. They are expressed in `if-then` syntax, for example, `if ( age < 16) marital_status == "unmarried"`. Third, there are rules of mixed type. These are logical conditions where each clause consists of either a linear restriction or a restriction on categorical values. The following example illustrates how such rules can be defined in the `validate` package.

```
rules <- validator(
  employees >= 0,                                  # numerical
  profit == turnover - cost,                       # numerical
  sector   %in% c("nonprofit", "industry"),        # categorical
  bankrupt %in% c(TRUE, FALSE),                    # categorical
  if (bankrupt == TRUE) sector == "industry",      # categorical
  if ( sector    == "nonprofit"
     | bankrupt  == TRUE
     ) profit    <= 0,     # conditional
  if (turnover > 0) employees >= 1                 # conditional
)
```

The variables used in the rules are taken from the observation to be checked. It is also possible to supply one or more reference datasets that can be used in the error localization. The variables of these reference datasets are then excluded from the error localization procedure.

A Simple Example

Finding error locations with errorlocate is done with the locate_errors function. This function needs at least two parameters: a data.frame and a validator object.

The function locate_errors returns an errorlocation object that contains not only the location of errors but also some metadata and data on the process of finding the error. The location of the errors can be retrieved using the values method of errorlocation.

```
library(errorlocate)
rules <- validator(
  age >= 0
)
raw_data <- data.frame(age = -1, married = TRUE)
le <- locate_errors(raw_data, rules)
values(le)
##       age married
## [1,] TRUE   FALSE
```

In the example, the variable age is erroneous, since an age cannot be negative. Let us expand the example a little bit:

```
library(errorlocate)
rules <- validator(
  age >= 0,
  if (age <= 16) married == FALSE
)
raw_data <- data.frame(age = -1, married = TRUE)
le <- locate_errors(raw_data, rules)
values(le)
##       age married
## [1,] TRUE   FALSE
```

We have added a conditional restriction describing that a person cannot be married at the age of 16 years or younger. `error_locate` correctly identifies `age` to be a value that is causing the invalidation of both rules.

We can automatically remove errors using the `replace_errors` function.

```
replace_errors(raw_data, rules)
##   age married
## 1  NA    TRUE
```

`replace_errors` internally calls the `error_locate` function. Any extra argument supplied to `replace_errors` will be passed through to `error_locate`.

What happens if `age` is `4`?

```
library(errorlocate)
v <- validator(
  age >= 0,
  if (married==TRUE) age >= 16
)
raw_data <- data.frame( age     = 4
                      , married = TRUE)
le <- locate_errors(raw_data, v)

values(le)
##          age married
## [1,] FALSE    TRUE
```

Note that both `age` and `married` can be wrong. `error_locate` decides at random which of the variables is considered to be wrong. This is sensible since `error_locate` has no further information whether `age` or `married` is to be preferred. However, it is possible to supply a `weight` vector, matrix, or data.frame to `error_locate` to indicate in which value `error_locate` should put more trust.

```
raw_data <- data.frame(age = 4, married = TRUE)
weight <- c(age = 2, married = 1) # assuming we have a
registration of marriages
le <- locate_errors(raw_data, v, weight=weight)
values(le)
##          age married
## [1,] FALSE    TRUE
```

Some Words on the Implementation of `errorlocate`

The implementation of the Fellegi–Holt algorithm is located in an R class `FHErrorLocator`, which is derived from the base class `ErrorLocator`. Each `ErrorLocator` object has to implement two methods.

The `editrules` package, the predecessor of `errorlocate`, implemented error localization with the branch-and-bound algorithm described by de Waal (2003) and a `MIP` formulation.

The main disadvantage of the branch-and-bound algorithm is that it has $\mathcal{O}(2^n)$ worst-case time and memory complexity, where n is the number of variables occurring in a connected set of rules. Moreover, the branch-and-bound solver was written in pure R, making it intrinsically slower than a compiled language implementation. The main advantages of this approach were the ease of implementation and the opportunity for users to exert fine-grained control over the algorithm.

7.3 Error Localization as MIP-Problem

The error localization problem can be translated to a MIP-problem. This allows us to reuse well-established results from the field of linear and MIP. Indeed, many advanced algorithms for solving such problems have been developed, and in many cases, implementations in a compiled language are available under a permissive license. In `errorlocate`, the solver of the `lp_solve` library (Berkelaar *et al.*, 2010) is used through R's `lpSolveAPI` package (Konis, 2011). The `lp_solve` library is written in `ANSI C` and has been tried and tested extensively.

The strategy to solve error localization problems through this library from R therefore consists of translating the problem to a suitable MIP-problem, feeding this problem to `lpSolveAPI`, and translating the results back to an error location. It is necessary to distinguish among the following:

- linear restrictions on purely numerical data,
- restrictions on purely categorical data, and
- conditional restrictions on mixed-type data,

since restriction for each data type calls for a different translation to a MIP problem.

We will focus on how to translate these types of error localization problems to a mixed-integer formulation, paying attention to both theoretical and practical details. In Section 7.4, attention is paid to numerical stability issues and Section 7.2 is devoted to examples in R code.

7.3.1 Error Localization and Mixed-Integer Programming

A MIP-problem is an optimization problem that can be written in the form

$$\begin{aligned} \text{Minimize } & f(\boldsymbol{z}) = \boldsymbol{c}^T\boldsymbol{z}; \\ & \text{s.t. } \boldsymbol{R}\boldsymbol{z} \leq \boldsymbol{d}, \end{aligned} \tag{7.1}$$

where $\boldsymbol{c}$ is a constant vector and $\boldsymbol{z}$ is a vector consisting of real and integer coefficients. One usually refers to $\boldsymbol{z}$ as the *decision vector* and the inner product $\boldsymbol{c}^T\boldsymbol{z}$ as the *objective function*. Furthermore, $\boldsymbol{R}$ is a coefficient matrix and $\boldsymbol{d}$ a vector of upper bounds. Formally, the elements of $\boldsymbol{c}$, $\boldsymbol{R}$, and $\boldsymbol{d}$ are limited to the rational numbers (Schrijver, 1998). This is never a problem in practice since we are always working with a computer representation of numbers.

The name *MIP* stems from the fact that $\boldsymbol{z}$ contains continuous as well as integer variables. When $\boldsymbol{z}$ consists solely of continuous or integer variables, Problem (7.1) reduces, respectively, to a *linear* or an *integer programming* problem. An important special case occurs when the integer coefficients of $\boldsymbol{z}$ may only take values from $\{0,1\}$. Such variables are often called binary variables. It occurs as a special case since defining $\boldsymbol{z}$ to be integer and applying the appropriate upper bounds yield the same problem.

MIP is well understood, and several software packages are available that implement efficient solvers. Most MIP software support a broader, but equivalent, formulation of the MIP problem, allowing the set of restrictions to include inequalities as well as equalities. As a side note we mention that under equality restrictions, solutions for the integer part of $\boldsymbol{z}$ are only guaranteed to exist when the equality restrictions pertaining to the integer part of $\boldsymbol{z}$ are *totally unimodular*.[1] However, as we will see below, restrictions on $\boldsymbol{z}$ are always inequalities in our case, so this is of no particular concern to us.

We reformulate Fellegi–Holt error localization (Fellegi and Holt, 1976) for numerical, categorical, and mixed-type restrictions in terms of MIP problems. The precise reformulations of the error localization problem for the three types of rules are different, but in each case the objective function is of the form

$$\boldsymbol{w}^T\boldsymbol{\Delta}, \tag{7.2}$$

where $\boldsymbol{w}$ is a vector of positive weights and $\boldsymbol{\Delta}$ a vector of binary variables, one for each variable in the original record, that indicates whether its value should be replaced. More precisely, for a record $\boldsymbol{r} = (r_1, r_2, \ldots, r_n)$ of n variables, we define

$$\Delta_i = \begin{cases} 1 \text{ if the value of } r_i \text{ must be replaced,} \\ 0 \text{ otherwise.} \end{cases} \tag{7.3}$$

This objective function obviously meets the requirement that the minimal (weighted) number of variables should be replaced. In general, a record may contain numeric, categorical, or both types of data, and restrictions may pertain to either one or both data types. To distinguish between the data types below, we shall write $\boldsymbol{r} = (\boldsymbol{v}, \boldsymbol{x})$, where $\boldsymbol{v}$ represents the categorical and $\boldsymbol{x}$ the numerical part of $\boldsymbol{r}$.

For an error localization problem, the restrictions of Problem (7.1) consist of two parts, which we denote

$$\begin{bmatrix} \boldsymbol{R}^H \\ \boldsymbol{R}^0 \end{bmatrix} \boldsymbol{z} \leq \begin{bmatrix} \boldsymbol{d}^H \\ \boldsymbol{d}^0 \end{bmatrix}. \tag{7.4}$$

Here, the restrictions indicated with H represent a matrix representation of the user-defined (hard) restrictions that the original record $\boldsymbol{r}$ must obey. The vector $\boldsymbol{z}$ contains at least a numerical representation of the values in a record $\boldsymbol{r}$ and the binary variables $\boldsymbol{\Delta}$. An algorithmic MIP solver will iteratively alter the values of $\boldsymbol{z}$ until a solution satisfying (7.4) is reached. To make sure that the objective function reflects the (weighted) number of variables altered in the process, the restrictions in $\boldsymbol{R}^0$ serve to make sure that the values in $\boldsymbol{z}$ that represent values in $\boldsymbol{r}$ cannot be altered without setting the corresponding value in $\boldsymbol{\Delta}$ to 1.

Summarizing, in order to translate the error localization problem for the special cases of linear, categorical, or conditional mixed-type restrictions to a general MIP-problem, for each case we need to properly define $\boldsymbol{z}$, the restriction set $\boldsymbol{R}^H\boldsymbol{z} \leq \boldsymbol{d}^H$, and the restriction set $\boldsymbol{R}^0\boldsymbol{z} \leq \boldsymbol{d}^0$.

7.3.2 Linear Restrictions

For a numerical record $\boldsymbol{x}$ taking values in $\mathbb{R}^n$, a set of linear restrictions can be written as

$$\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \tag{7.5}$$

[1] This means that every square submatrix of the coefficient matrix $\boldsymbol{R}$ pertaining to the integer part of $\boldsymbol{z}$ has determinant 0 or ±1.

where in `errorlocate`, we allow the set of restrictions to contain equalities, inequalities ($\leq$), and strict inequalities ($<$). The formulation of these rules is very close to the formulation of the original MIP problem of Eq. (7.1). The vector to minimize over is defined as follows:

$$\boldsymbol{z} = (x_1, x_2, \ldots, x_n, \Delta_1, \Delta_2, \ldots, \Delta_n) \tag{7.6}$$

with the Δ_i as in Eq. (7.3). The set of restrictions $\boldsymbol{R}^H\boldsymbol{z} \leq \boldsymbol{d}^H$ is equal to the set of restrictions of Eq. (7.5), except in the case of strict inequalities. The reason is that while `errorlocate` allows the user to define strict inequalities ($<$), the `lpsolve` library used by `errorlocate` only allows for inclusive inequalities ($\leq$). For this reason, strict inequalities of the form $\boldsymbol{a}^T\boldsymbol{x} < b$ are rewritten as $\boldsymbol{a}^T\boldsymbol{x} \leq b - \epsilon$, with ϵ a suitably small positive constant.

In the case of linear rules, the set of constraints $\boldsymbol{R}^0\boldsymbol{z} \leq \boldsymbol{d}^0$ consists of pairs of the form

$$\begin{aligned} x_i - M\Delta_i &\leq x_i^0, \\ -x_i - M\Delta_i &\leq -x_i^0 \end{aligned} \tag{7.7}$$

for $i = 1, 2, \ldots, n$. Here, x_i^0 are the actual observed values in the record and M a suitably large positive constant allowing x_i to vary between $x_i^0 - M$ and $x_i^0 + M$. It is not difficult to see that if x_i is different from x_i^0, then Δ_i must equal 1. For, if we choose $\Delta_i = 0$, we obtain the set of restrictions

$$x_i^0 \leq x_i \leq x_i^0, \tag{7.8}$$

which states that x_i equals x_i^0.

Example 7.3.6 *Consider a record with business survey data, consisting of the variables* Number of staff *p and* Personnel cost *c. We have the rules $p \geq 0$, $c \geq 0$, and $c \geq p$. The latter rule expresses the notion that for each staff member, more than one monetary unit is spent. Given two observed values p^0 and c^0, disobeying one or more of the rules, the MIP problem for error localization has the following form:*

$$\underset{(\boldsymbol{x},\boldsymbol{\Delta})\in\mathbb{R}^2\times\{0,1\}^2}{\text{Minimize}} \Delta_p + \Delta_c$$

$$s.t. \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & -M & 0 \\ -1 & 0 & -M & 0 \\ 0 & 1 & 0 & -M \\ -1 & 1 & 0 & -M \end{bmatrix} \begin{bmatrix} p \\ c \\ \Delta_p \\ \Delta_c \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ p^0 \\ -p^0 \\ c^0 \\ -c^0 \end{bmatrix}.$$

Here, the first three rows in the set of restrictions represent the consistency rules, while the other rows connect the indicator variables $\boldsymbol{\Delta} = (\Delta_p, \Delta_c)$ with p and c.

7.3.3 Categorical Restrictions

Categorical records $\boldsymbol{v} \in D$ take values in a Cartesian product domain

$$D = D_1 \times D_2 \times \cdots \times D_m, \tag{7.9}$$

where each D_i is a finite set of categories for the ith categorical variable. The category names are unimportant, so we write

$$D_i = \{1, 2, \ldots, |D_i|\}. \tag{7.10}$$

The total number of possible value combinations $|D|$ is equal to the product of the $|D_i|$.

A categorical edit is a subset F of D where records are considered invalid, and we may write

$$F = F_1 \times F_2 \times \cdots \times F_m, \tag{7.11}$$

where each F_i is a subset of D_i. It is understood that if a record $\boldsymbol{v} \in F$, then the record violates the edit. Hence, categorical rules are negatively formulated (they specify the region of D where $\boldsymbol{v}$ may not be) in contrast to linear rules that are positively formulated (they specify the region of $\mathbb{R}^n$ where $\boldsymbol{x}$ must be). To be able to translate categorical rules to a MIP problem, we need to specify $\overline{F}$ such that if $\boldsymbol{v} \in \overline{F}$, then $\boldsymbol{v}$ satisfies e. Here, $\overline{F}$ is the complement of F in D, which can be written as

$$\begin{aligned}\overline{F} = {} & \overline{F}_1 \times D_2 \times \cdots \times D_m \\ & \cup D_1 \times \overline{F}_2 \times \cdots \times D_m \cup \cdots \cup D_1 \times D_2 \times \cdots \times \overline{F}_m,\end{aligned} \tag{7.12}$$

where for each variable v_i, $\overline{F}_i$ is the complement of F_i in D_i. Observe that Eq. (7.12) states that if at least one $v_i \in \overline{F}_i$, then $\boldsymbol{v}$ satisfies e. Below, we will use this property and construct a linear relation that counts the number of $v_i \in \overline{F}_i$ over all variables.

To be able to formulate the Fellegi-Holt problem in terms of a MIP problem, we first associate with each categorical variable v_i a binary vector $\boldsymbol{d}$ of which the coefficients are defined as follows (see also Eq. (7.9)):

$$d_\lambda(v_i) = \begin{cases} 1 \text{ if } v_i = \lambda, \\ 0 \text{ otherwise,} \end{cases} \tag{7.13}$$

where $\lambda \in D_i$. Thus, each element of $\boldsymbol{d}(v_i)$ corresponds to one category in D_i. It is zero everywhere except at the value of $v_i \in D_i$. We will write $\boldsymbol{d}(\boldsymbol{v})$ to indicate the concatenated vector $(\boldsymbol{d}(v_1), \ldots, \boldsymbol{d}(v_m))$, which represents a complete record. Similarly, each edit can be represented by a binary vector $\boldsymbol{e}$ given by

$$\boldsymbol{e} = \left(\bigvee_{\lambda \in \overline{F}_1} \boldsymbol{d}(\lambda), \ldots, \bigvee_{\lambda \in \overline{F}_m} \boldsymbol{d}(\lambda) \right), \tag{7.14}$$

where we interpret 1 and 0 as `TRUE` and `FALSE`, respectively, and the logical ’or’ ($\vee$) is applied element-wise to the coefficients of $\boldsymbol{d}$. The above relation can be interpreted as stating that $\boldsymbol{e}$ represents the valid value combinations of variables contained in the edit.

To set up the hard restriction matrix $\boldsymbol{R}^H$ of Eq. (7.4), we first impose the obvious restriction that each variable can take but a single value:

$$\sum_{\lambda \in D_i} d_\lambda(v_i) = 1 \tag{7.15}$$

for $i = 1, 2, \ldots, m$. It is now not difficult to see that the demand (Eq. (7.12)) that at least one of the $v_i \in \overline{F}_i$ may be written as

$$\boldsymbol{e}^T \boldsymbol{d}(\boldsymbol{v}) \geq 1. \tag{7.16}$$

Eqs. (7.15) and (7.16) constitute the hard restrictions, stored in $\boldsymbol{R}^H$.

Using the binary vector notation for $\boldsymbol{v}$, and adding the Δ variables that indicate variable change, the vector to minimize over (Eq. (7.1)) is written as

$$\boldsymbol{z} = (\boldsymbol{d}(\boldsymbol{v}), \Delta_1, \Delta_2, \dots, \Delta_m). \tag{7.17}$$

To ensure that a change in v_i results in a change in Δ_i, the matrix $\boldsymbol{R}^0$ contains the restrictions

$$d_{\lambda^0}(v_i) = 1 - \Delta_i, \tag{7.18}$$

for $i = 1, 2, \dots, m$. Here, $\lambda^0 \in D_i$ is the observed value for variable v_i. One may check, using Eq. (7.13), that the above equation can only hold when either $v_i = \lambda^0$ and $\Delta_i = 0$ (the original value is retained) or $v_i \neq \lambda^0$ and $\Delta_i = 1$ (the value changes).

Example 7.3.7 *Consider a two-variable record from the census with the variables* Marital status *m and* Age class *a. We have $\boldsymbol{v} = (m, a) \in D$ where*

$$D = D_m \times D_a = \{\textsf{married, unmarried}\} \times \{\textsf{child, adult}\}.$$

Using the binary representation, we see that a married adult is represented by the vector $\boldsymbol{v}^0 = (\boldsymbol{d}(\textsf{married}), \boldsymbol{d}(\textsf{adult})) = (1, 0, 0, 1)$. The rule that states "A child cannot be married" *translates to*

$$F = F_m \times F_a = \{\textsf{married}\} \times \{\textsf{child}\},$$

which gives $\overline{F}_m = \{\textsf{unmarried}\}$ and $\overline{F}_a = \{\textsf{adult}\}$. Using Eq. (7.14), we get $\mathbf{e} = (0, 1, 0, 1)$, and one may verify that $\mathbf{e}^T\boldsymbol{d}(\textsf{married, child}) = 0$ and therefore invalid (see Eq. (7.16)). For $\boldsymbol{v}^0$, the MIP problem for error localization now looks like this.

$$\underset{(\boldsymbol{v},\boldsymbol{\Delta})\in D\times\{0,1\}^2}{\textsf{Minimize}} \Delta_m + \Delta_a$$

$$\text{s.t.} \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} d_{married}(m) \\ d_{unmarried}(m) \\ d_{child}(a) \\ d_{adult}(a) \\ \Delta_m \\ \Delta_a \end{bmatrix} \begin{matrix} \geq \\ = \\ = \\ = \\ = \end{matrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}. \tag{7.19}$$

Here, the first row represents the edit rule, the second and third force that each variable can take but one value (Eq. (7.16)), and the last two rows connect the indicator variables Δ_m and Δ_a with the value of m and a (Eq. (7.18)).

7.3.4 Mixed-Type Restrictions

Records $\boldsymbol{r}$ containing both numerical and categorical data can be denoted as a concatenation of categorical and numerical variables taking values in $D \times \mathbb{R}^n$:

$$\boldsymbol{r} = (v_1, \dots, v_m, x_1, \dots, x_n) = (\boldsymbol{v}, \boldsymbol{x}), \tag{7.20}$$

where D is defined in Eq. (7.9). As stated earlier, categorical edits are usually defined negatively as a region of D that is disallowed, while linear rules define regions in $\mathbb{R}^n$ that are allowed. We may choose a negative formulation of rules containing both variable types by defining a single rule E as follows:

$$E = \{\boldsymbol{r} \in D \times \mathbb{R}^n : \boldsymbol{v} \in F \wedge \boldsymbol{x} \in P\}, \tag{7.21}$$

where $F \subseteq D$ and P is a convex subset of $\mathbb{R}^n$ defined by a (possibly empty) set of k linear inequalities of the form $\boldsymbol{a}^T\boldsymbol{x} > b$. It is understood that if $\boldsymbol{r} \in E$, then $\boldsymbol{r}$ violates the rule. An example of a restriction pertaining to a categorical and a numerical variable is 'A company employs staff if and only if it has positive personnel cost'. The corresponding rule can be denoted as $\{\text{no staff}\} \times \{c > 0\}$.

To obtain a positive reformulation, we first negate the set membership condition and apply basic rules of proposition logic:

$$\begin{aligned} &\neg(\boldsymbol{v} \in F \wedge \boldsymbol{x} \in P) \\ \Leftrightarrow &\neg(\boldsymbol{v} \in F \wedge \boldsymbol{a}_1^T\boldsymbol{x} > b_1 \wedge \cdots \wedge \boldsymbol{a}_k^T\boldsymbol{x} > b_k) \\ \Leftrightarrow &\,\boldsymbol{v} \in \overline{F} \vee \boldsymbol{a}_1^T\boldsymbol{x} \leq b_1 \vee \cdots \vee \boldsymbol{a}_k^T\boldsymbol{x} \leq b_k. \end{aligned} \tag{7.22}$$

This then yields a positive formulation of E. That is, a record $\boldsymbol{r}$ satisfies E if and only if

$$\boldsymbol{r} \in \overline{E} \Leftrightarrow \bigvee_{i=1}^{m} v_i \in \overline{F}_i \ \vee \ \bigvee_{j=1}^{k} \boldsymbol{a}_j^T\boldsymbol{x} \leq b_j. \tag{7.23}$$

Observe that this formulation allows one to define multiple disconnected regions in $D \times \mathbb{R}^n$ containing valid records using just a single rule. For example, one may define a numeric variable to be either smaller than 0 or larger than 1. This type of restriction cannot be formulated using just linear numerical restrictions.

This formulation is both a generalization of linear inequality (Eq. (7.5)) and categorical rules (Eq. (7.11)). Choosing $k = 0$, we get $P = \mathbb{R}^n$, and only the categorical part remains. Similarly, choosing $F = \emptyset$, only the disjunction of linear inequalities remains. A system of linear equations that must simultaneously be obeyed like in Eq. (7.5) can be obtained by defining multiple rules E, each containing a single linear restriction.

The definition in Eq. (7.22) can be rewritten as a 'conditional rule' using the implication replacement rule from propositional logic, which states that $\neg p \vee q$ may be replaced by $p \Longrightarrow q$. If we limit Eq. (7.22) to a single inequality, we obtain the normal form of de Waal (2003).

$$\boldsymbol{v} \in F \Longrightarrow \boldsymbol{a}^T\boldsymbol{x} \leq b. \tag{7.24}$$

If we choose $F = \emptyset$ and leave two inequalities, we obtain a conditional rule on numerical data:

$$\boldsymbol{a}_1^T\boldsymbol{x} > b_1 \Longrightarrow \boldsymbol{a}_2^T\boldsymbol{x} \leq b_2. \tag{7.25}$$

Writing mixed-type rules in conditional form seems more user-friendly as they can directly be translated into an `if` statement in a scripting language. Finally, note that equalities can be introduced by defining pairs of rules like the following:

$$\begin{cases} \boldsymbol{v} \in F \Longrightarrow \boldsymbol{a}^T\boldsymbol{x} \leq b, \\ \boldsymbol{v} \in F \Longrightarrow -\boldsymbol{a}^T\boldsymbol{x} \leq -b. \end{cases} \tag{7.26}$$

To reformulate Eq. (7.22) as a MIP problem, we first define binary variables ℓ_j that indicate whether $\boldsymbol{x}$ obeys $\boldsymbol{a}_k^T\boldsymbol{x} > b$:

$$\ell_j = \begin{cases} 0 \text{ when } \boldsymbol{a}_j^T\boldsymbol{x} \leq b_j, \\ 1 \text{ when } \boldsymbol{a}_j^T\boldsymbol{x} > b_j \end{cases} \tag{7.27}$$

Using the or-form of the set condition (Eq. (7.22)), we can write the mixed-data rule as

$$\boldsymbol{e}^T\boldsymbol{d}(\boldsymbol{v}) + \sum_{j=1}^{k}(1-\ell_j) \geq 1. \tag{7.28}$$

Recall from Eqs. (7.14) and (7.13) that $\boldsymbol{e}$ is the binary vector representation of a categorical rule and $\boldsymbol{d}(\boldsymbol{v})$ the binary vector representation of a categorical record. In the above equation, the '+' is the arithmetic translation of the logical '∨' operator in Eq. (7.22) that connects the categorical with the linear restrictions. When any one of the two terms is positive, record r satisfies rule E.

Rules of this form constitute the user-defined part of the $\boldsymbol{R}^H$ part of the restriction matrix. To explicitly identify ℓ_j with the linear restrictions, we also add

$$\boldsymbol{a}_j^T\boldsymbol{x} \leq b_j + M\ell_j, \tag{7.29}$$

to $\boldsymbol{R}^H$ with M a suitably large positive constant. Indeed, if $\ell_j = 0$, the inequality $\boldsymbol{a}_j^T\boldsymbol{x} \leq b_j$ is enforced, and Eq. (7.28) always is satisfied. When $\ell_j = 1$, the whole restriction can hold regardless of whether the inequality holds. Finally, similar to the purely categorical case, we need to add restrictions on the binary representation of $\boldsymbol{v}$ as in Eq. (7.15), so Eqs. (7.15), (7.28), and (7.29) constitute $\boldsymbol{R}^H$.

There may be multiple mixed-type rules, each yielding one or more l indicator variables for each rule. The decision vector for the MIP problem may therefore be written as

$$\boldsymbol{z} = (\boldsymbol{d}(\boldsymbol{v}), \boldsymbol{x}, \Delta_1, \ldots, \Delta_m, \ldots, \Delta_{m+n}, \ell_1, \ldots, \ell_K), \tag{7.30}$$

where K is the total number of linear rules occurring in all the mixed-type rules. Finally, the $\boldsymbol{R}^0$ matrix connecting the change indicator variables (Δ) with the actual recorded values consists of the union of the restrictions for categorical variables (Eq. (7.18)) and those for numerical variables (Eq. (7.7)).

Example 7.3.8 *We consider a record* **r** *with the variables* type of business *t, which takes values in* $D_t = \{sp, other\}$, *where "sp" stands for "sole proprietorship",* personnel cost $c \in \mathbb{R}$, *and* number of staff $p \in \mathbb{R}$. *Hence, we have* $\boldsymbol{r} = (t, p, c) \in D_t \times \mathbb{R}^2$. *We impose the following rules on* **r**: $p \geq 0$, $c \geq 0$, $c \geq p$, *and if the* business type *is a sole proprietorship, then the number of staff must equal zero. This may be expressed as* $(t \in \{sp\}) \implies (p = 0)$ *or equivalently* $(t \in \{other\}) \vee (p = 0)$. *For a record* $\boldsymbol{r}^0 = (sp, p^0, c^0)$, *the error localization problem takes the following form:*

$$\underset{(\boldsymbol{r},\boldsymbol{\Delta},\ell)\in D_t\times\mathbb{R}^2\times\{0,1\}^4}{\text{Minimize}} \quad \Delta_t + \Delta_p + \Delta_c$$

$$\text{s.t.} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & -M \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -M & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -M & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -M & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & -M & 0 \end{bmatrix} \begin{bmatrix} d_{sp}(t) \\ d_{other}(t) \\ p \\ c \\ \Delta_t \\ \Delta_p \\ \Delta_c \\ \ell \end{bmatrix} \begin{array}{c} \geq \\ \geq \\ \geq \\ \geq \\ = \\ > \\ = \\ \leq \\ \leq \\ \leq \\ \leq \end{array} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ p^0 \\ -p^0 \\ c^0 \\ -c^0 \end{bmatrix}.$$

The first row in the restriction represents the mixed-type rule, translated as shown in Eq. (7.28). Row 6 connects the indicator variable ℓ with the numerical rule in the consequent of $t \in \{sp\} \Longrightarrow (p = 0)$. *Rows 2, 3, and 4 represent the numerical rules limiting values of p and c. Row 5 forces t to have only one value, and row 7 connects the value of t with that of* Δ_t. *Finally, rows 8–11 connect the numerical variables with the corresponding change indicators.*

7.4 Numerical Stability Issues

An error localization problem, in its original formulation, is an optimization problem over n binary decision variables that indicate which variables in a record should be adapted. Depending on the type of rules, its reformulation as a MIP problem adds at least n variables and $2n$ restrictions. Moreover, the reformulation as a MIP problem introduces a constant M, the value of which has no mathematical significance but for which a value must be chosen in practice. Because of limitations in machine accuracy, which is typically on the order of 10^{-16}, the range of problems that can be solved is limited as well. In particular, MIP problems that involve both very large and very small numbers in the objective function and/or the restriction matrix may yield erroneous solutions or become numerically unfeasible. Indeed, the manual of `lp_solve` (Berkelaar *et al.*, 2010) points out that '*[...] to improve stability, one must try to work with numbers that are somewhat in the same range. Ideally, in the neighborhood of 1*'. The following sections point out a number of sources of numerical instabilities and provide ways to handle them.

7.4.1 A Short Overview of MIP Solving

Consider a set of linear restrictions on numerical data of the form $\boldsymbol{Ax} \leq \boldsymbol{b}$, where we assume $\boldsymbol{b} \geq \boldsymbol{0}$, and the restrictions consist solely of inequalities ($\leq$). In practice, these restrictions will not limit the type of linear rules covered by this discussion, since it can be shown that all linear rules can be brought to this form, possibly by introducing dummy variables [see, e.g., Bradley *et al.* (1977); Schrijver (1998)]. Furthermore, suppose that we have a record $\boldsymbol{x}^0 \geq \boldsymbol{0}$ that does not obey the restrictions. The MIP formulation of the error localization problem can be written as follows:

$$\text{Minimize } f = \boldsymbol{w}^T\boldsymbol{\Delta}$$
$$\text{s.t.} \begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{I} & -\boldsymbol{M} \\ -\boldsymbol{I} & -\boldsymbol{M} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{\Delta} \end{bmatrix} \leq \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{x}^0 \\ -\boldsymbol{x}^0 \\ \boldsymbol{1} \end{bmatrix}, \tag{7.31}$$

and $\boldsymbol{x}, \boldsymbol{\Delta} \geq \boldsymbol{0}$. Also, $\boldsymbol{I}$ denotes the unit matrix, $\boldsymbol{1}$ a vector with all coefficients equal to 1, and $\boldsymbol{M} = \boldsymbol{I}M$. The last row is added to force $\boldsymbol{\Delta} \leq \boldsymbol{1}$. This is necessary because we will initially treat the binary variables Δ_j as if they are real numbers in the range [0, 1].

The `lp_solve` library uses an approach based on the revised Phase I– Phase II simplex algorithm to solve MIP problems. In this approach, every inequality of Eq. (7.31) is transformed to an equality by adding dummy variables: each row $\boldsymbol{a}^T\boldsymbol{x} \leq b$ is replaced by $\boldsymbol{a}^T\boldsymbol{x} + s = b$, with $s \geq 0$. Depending on the sign of the inequality, the extra variable s is called a *slack* or *surplus* variable. In Eq. (7.31), there are four sets of restrictions (rows). We therefore need to add four sets of surplus and slack variables (columns) in order to rewrite the whole system in terms of equalities.

Note that after this transformation, the whole problem including the cost function is written in terms of equalities. It is customary to organize this set of equality objective function in a single *tableau* notation as follows:

$$\left[\begin{array}{c|cccccc|c} 1 & \mathbf{0} & -\boldsymbol{w}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ \hline 0 & \boldsymbol{A} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{b} \\ 0 & \boldsymbol{I} & -\boldsymbol{M} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \boldsymbol{x}^0 \\ 0 & \boldsymbol{I} & \boldsymbol{M} & \mathbf{0} & \mathbf{0} & -\boldsymbol{I} & \mathbf{0} & \boldsymbol{x}^0 \\ 0 & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{I} & \mathbf{1} \end{array}\right]. \tag{7.32}$$

Here, the first row and column represent the cost function. Columns 2 and 3 correspond to the original set of variables in Eq. (7.31), while columns 4-7 correspond to sets of slack and surplus variables. The final column contains the constant vector.

A tableau representation shows all the numbers that are relevant in an LP problem at a glance. By examining how LP solvers typically manipulate these numbers, we gain some insight into how and where numerical stability issues may arise.

Since the tableau represents a set of linear equalities, it may be manipulated as such. In fact, the simplex method is based on performing a number of cleverly chosen Gauss–Jordan elimination steps on the tableau. For a complete discussion, the reader is referred to one of the many textbooks discussing it (e.g., Bradley *et al.* (1977)), but in short, the Phase I–Phase II simplex algorithm consists of the following steps:

Phase I. Repeatedly apply Gauss–Jordan elimination steps (called *pivots*) to derive a decision vector that obeys all restrictions. A vector obeying all restrictions is called a *basic solution.*

Phase II. Repeatedly apply pivots to move from the initial nonoptimal solution to the solution that minimizes the objective function f.

In Phase I, a decision vector $(\boldsymbol{x}, \boldsymbol{\Delta}, \boldsymbol{s})$ (with $\boldsymbol{s}$ the vector of slack and surplus variables) is derived that obeys all restrictions. The precise algorithm need not be described here. It involves adding again extra variables where necessary and then manipulating the system of equalities represented by the tableau so that those extra variables are driven to zero. The binary variables $\boldsymbol{\Delta}$ are first treated as if they are real variables. In Appendix 7.A, it is shown in detail how an initial solution for Eq. (7.32) can be found; here, we just state the result of a Phase I operation:

$$\left[\begin{array}{c|cccccc|l} 1 & \boldsymbol{w}^T\boldsymbol{M}^{-1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & -\boldsymbol{w}^T\boldsymbol{M}^{-1} & \mathbf{0} & \boldsymbol{w}^T\boldsymbol{M}^{-1}\boldsymbol{x}^0 \\ \hline 0 & \boldsymbol{A} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{b} \\ 0 & 2\boldsymbol{I} & \mathbf{0} & \mathbf{0} & \boldsymbol{I} & -\boldsymbol{I} & \mathbf{0} & 2\boldsymbol{x}^0 \\ 0 & \boldsymbol{M}^{-1} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & -\boldsymbol{M}^{-1} & \mathbf{0} & \boldsymbol{M}^{-1}\boldsymbol{x}^0 \\ 0 & -\boldsymbol{M}^{-1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{M}^{-1} & \boldsymbol{I} & \mathbf{1}-\boldsymbol{M}^{-1}\boldsymbol{x}^0 \end{array}\right]. \tag{7.33}$$

This tableau immediately suggests a valid solution: it is easily confirmed by matrix multiplication that the vector $(\boldsymbol{x}, \boldsymbol{\Delta}, \boldsymbol{s}) = (\mathbf{0}, \boldsymbol{x}^0\boldsymbol{M}^{-1}, [\boldsymbol{b}, 2\boldsymbol{x}^{\mathbf{0}}, \mathbf{0}, \mathbf{1} - \boldsymbol{M}^{-1}\boldsymbol{x}^0])$ obeys all restrictions. In the above form of a tableau, where the restriction matrix contains (a column permutation of) the unit matrix, the right-hand side has only nonnegative coefficients, and the cost vector equals zero for the columns above the unit matrix, which is called the *canonical form*.

Now, a *pivot* operation consists of the following steps:

1) Select a positive element R_{ij} from the restriction matrix. This is called the *pivot element*.
2) Multiply the ith row by R_{ij}^{-1}.
3) Subtract the ith row, possibly after rescaling, from all other rows of the tableau such that their jth column equals zero.

The result of a pivot operation is again a tableau in canonical form but with possibly a different value for the cost function. The simplex algorithm proceeds by selecting pivots that decrease the cost function until the minimum is reached or the problem is shown to be unfeasible.

Up until this point, we have treated the binary variables $\boldsymbol{\Delta}$ as if they were real variables, so the tableaux discussed above do not represent solutions to our original problem, which demands that all Δ_j are either 0 or 1. In the `lp_solve` library, this is solved as follows:

1) For each optimized value Δ_j^*, test whether it is 0 or 1. If all Δ_j^* are integers, we have a valid solution of objective value $\boldsymbol{w}^T\boldsymbol{\Delta}$, and we are done.
2) For the first variable Δ_j^* that is not integer, create two submodels: one where the minimum value of Δ_j equals 1, and one where the maximum value of Δ_j equals 0.
3) Optimize the two submodels. If solutions exist, the result will contain an integer Δ_j.
4) For the submodels that have a solution and whose current objective value does not exceed that of an earlier found solution, return to step 1.

The above *branch-and-bound* approach completes this overview. The discussion of pivot and branch-and-bound operations has so far been purely mathematical: no choices have been made regarding issues such as how to decide when the floating-point representation of a value is regarded zero, or how to handle badly scaled problems. Do note, however, that in the course of going from Phase I to Phase II, the LP solver is handling numbers that may range from M^{-1} to M, which typically differ many orders of magnitude.

7.4.2 Scaling Numerical Records

In the MIP formulation of error localization over numerical records under linear restrictions, Eq. (7.7) restricts the search space around the original value x^0 to $|x - x^0| \leq M$. This restriction may prohibit a MIP solver from finding the actual minimal set of values to adapt or even render the MIP problem unsolvable. As an example, consider the following error localization problem on a two-variable record:

$$\begin{cases} x_1 \geq x_2, \\ \boldsymbol{x}^0 = (10^6, 10^9). \end{cases}$$

Obviously, the record can be made to obey the restriction by multiplying x_1^0 by 10^3 or by dividing x_2^0 by the same amount. However, in `errorlocate`, the default value for $M = 10^7 < 10^9 - 10^6$, which renders the corresponding MIP problem unsolvable. Practical example where such errors occur is when a value is recorded in the wrong unit of measure (e.g., in € instead of k€.).

It is therefore advisable to remove such unit-of-measure errors prior to error localization[2] and to express numerical records on a scale such that all $|x^0| \ll M$. Note that under linear restrictions (Eq. (7.5)) one may always apply a scaling factor $k > 0$ to a numerical record $\boldsymbol{x}$ by replacing $\boldsymbol{Ax} \leq \boldsymbol{b}$ with $\boldsymbol{A}(k\boldsymbol{x}) \leq k\boldsymbol{b}$. In the above example, one may replace $\boldsymbol{x}^0$ by $10^{-6}\boldsymbol{x}^0$ for the purpose of error localization. If $\boldsymbol{b} = \boldsymbol{0}$ and the coefficients of $\boldsymbol{x}$ do not vary over many orders of magnitude, such a scaling will suffice to numerically stabilize the MIP problem.

7.4.3 Setting Numerical Threshold Values

On most modern computer systems, real numbers are represented in IEEE (2008) double precision format. In essence, real numbers are represented as rounded-off fractions so that arithmetic operations on such numbers always result in loss of precision and round-off errors. For example, even though mathematically we have $0.7 - 0.5 = 0.2$, in the floating-point representation (denoted $\mathrm{fl}(\cdot)$), we have $\mathrm{fl}(0.7) - \mathrm{fl}(0.5) \neq \mathrm{fl}(0.2)$. In fact, the difference is about 0.56×10^{-16} in this case.

This means that in practice one cannot rely on equality tests to determine whether two floating-point numbers are equal. Rather, one considers two numbers v and w equal when $|\mathrm{fl}(v) - \mathrm{fl}(w)|$ is smaller than a predefined tolerance. For this reason, `lp_solve` comes with a number of predefined tolerances. These tolerances have default values, but these may be altered by the user.

The tolerances implemented by `lp_solve` are summarized in Table 7.1. The value of `epspivot` is used to determine whether an element of the restriction matrix is positive so that it may be used as a pivoting element. Its default value is 2×10^{-7}, but note that after Phase I, our restriction matrix contains elements on the order of $M^{-1} = 10^{-7}$. For this reason, the value of `epspivot` is lowered in `errorlocate` by default, but users may override these settings. For the same reason, the value of `epsint`, which determines when a value for one of the $\boldsymbol{\Delta}_j$ can be considered integer, is lowered in

Table 7.1 Numerical parameters for MIP-based error localization.

	Default value		
Parameter	`lp_solve`	`errorlocate`	**Description**
`M`	—	10^7	Set bounds so $\boldsymbol{x} \in \boldsymbol{x}^0 \pm M$
`eps`	—	10^{-3}	Translate $x < 0$ to $x \leq \varepsilon$
`epspivot`	2×10^{-7}	10^{-15}	Test pivot element $R_{ij} > 0$
`epsint`	10^{-7}	10^{-15}	Test $\Delta_j \in \mathbb{N}$
`epsb`	10^{-10}	10^{-10}	Test $b_i > 0$
`epsd`	10^{-9}	10^{-9}	Test obj. values $|f - f'| > 0$ during simplex
`epsel`	10^{-12}	10^{-12}	Test other numbers $\neq 0$
`mip_gap`	10^{-11}	10^{-11}	Test obj. values $|f - f'| > 0$ during B&B

[2] Methods for detecting such errors exist, see, for example, de Waal *et al.* (2011, Chapter 2). In fact, the principle of minimal change is not applicable here since a better value can be deduced from the cause of the error.

errorlocate as well. The other tolerance settings of lp_solve, epsb (to test if the right-hand side of the restrictions differs from 0), epsd (to test if two values of the objective function differ), epsel (all other values), and mip_gap (to test whether a bound condition has been hit in the branch-and-bound algorithm), have not been altered.

The limited precision inherent to floating-point calculations implies that computations get more inaccurate as the operands differ more in magnitude. For example, on any system that uses double precision arithmetic, the difference $\mathrm{fl}(1) - \mathrm{fl}(10^{-17})$ is indistinguishable from $\mathrm{fl}(1)$. This, then, leads to two contradictory demands on our translation of an error localization problem to a MIP problem. On one hand, one would like to set M as large as possible so that the ranges $x_j^0 \pm M$ contain a valid value of x_j. On the other hand, large values for M imply that MIP problems such as Eq. (7.31) may become numerically unstable.

In practice, the tableau used by lp_solve will not be exactly the same as represented in Eq. (7.33). Over the years, many optimizations and heuristics have been developed to make solving linear programming problems fast and reliable, and several of those optimizations have been implemented in lp_solve. However, the tableau of Eq. (7.33) does fundamentally show how numerical instabilities may occur: the tableau simultaneously contains numbers on the order of M^{-1} and on the order of $\boldsymbol{x}^0$. It is not at all unlikely that the two differ in many orders of magnitude.

The above discussion suggests the following rules of thumb to avoid numerical instabilities in error localization problems:

1) Make sure that elements of $\boldsymbol{x}^0$ are expressed in units such that $\boldsymbol{A}$, $\boldsymbol{b}$, and $\boldsymbol{x}$ are on the order of 1 wherever possible.
2) Choose a value of M appropriate for $\boldsymbol{x}^0$.
3) If the above does not help in stabilizing the problem, try lowering the numerical constants of Table 7.1.

In our experience, the settings denoted in Table 7.1 have performed well in a range of problems where elements of $\boldsymbol{A}$ and $\boldsymbol{b}$ are on the order of 1 and values of $\boldsymbol{x}^0$ are in the range $[1, 10^8]$. However, these settings have been made configurable so that users may choose their own settings as needed.

7.5 Practical Issues

The following sections provide some background that can aid in choosing a set of reliability weights or to make a error localization problem more tractable by showing how certain rules can be simplified.

7.5.1 Setting Reliability Weights

The error localization problem can be reformulated as follows: For a record $\boldsymbol{r} = (r_1, r_2, \ldots, r_n)$ violating one or more validation rules imposed on it, find the set of variables $S \subseteq \{1, 2, \ldots, n\}$ of which the values can be altered such that (1) $\boldsymbol{r}$ satisfies all restrictions and (2) the sum

$$\sum_{j \in S} w_j$$

is minimized. Here, the vector $\boldsymbol{w} = (w_1, w_2, \dots, w_n)$ is a vector of chosen reliability weights. Solutions S that satisfy restriction (1) are called *feasible solutions*, and solutions that satisfy (2) are called *minimal feasible solutions*. We also denote with $|S|$ the size of the solution. The solution for which $|S|$ is minimal is called the *feasible solution of minimal size*.

When all weights are equal, the solution found by the error localization procedure is the solution that alters the least number of variables: in that case the solution of minimal weight is equal to the solution of minimal size. If the weights differ enough across the variables, this is not necessarily the case anymore, the solution of minimal weight may contain more variables than the solution of minimal size. As an example, suppose that there are $n = 3$ variables. We find for a certain error localization problem that there are three feasible solutions: $\{1\}$, $\{2, 3\}$, and the trivial solution $\{1, 2, 3\}$. If the weight vector $\boldsymbol{w} = (1, 1/3, 1/3)$, we find that $\{1\}$ is the solution of minimal size, while $\{1, 2\}$ is the solution of minimal weight.

In practice, weights are often determined by domain experts, based on their experience with repeatedly analyzing a type of dataset. One question arising in this context is whether it is possible to choose a set of weights such that the solution of minimal weight can be guaranteed to be a solution of minimal size. Such a set of weights then represents the guiding principle that states that we wish to change as few values as possible while still allowing to control the localization procedure with information on the relative reliability of variables. Perhaps surprisingly, it is possible to create such a vector. To show this we will derive the following result (this is a slightly stricter version of the result in van der Loo (2015a)).

Proposition 7.5.9 *Given a weight vector $\boldsymbol{w} = (w_1, w_2, \dots, w_n)$ with positive coefficients, we write $w_j = 1 + \delta_j$ with $0 \leq \delta \leq \delta^{max}$. If $\delta^{max} < 1/(n-1)$, then the feasible solution of minimal weight is equal to the feasible solution of minimal size.*

Proof: Suppose that S and T are different feasible solutions with $|S| < |T|$. The weights, associated with this solution, can be written as

$$w(S) = |S| + \sum_{j \in S} \delta_j,$$

and likewise for $w(T)$. We wish to ensure that $w(S) < w(T)$. Since variables occurring in $S \cap T$ do not add to the difference in weight, we assume without loss of generality that S and T have no variables in common. In the worst case, we have $\delta_j = \delta^{\max}$ for all $j \in S$ and $\delta_j = 0$ for all $j \in T$. The demand that $w(S) < w(T)$ can then be written as

$$(1 + \delta^{\max})|S| < |T|.$$

Since by assumption we have $|S| < |T|$, we know that T contains at least one more variable than S. So, we have in the worst case that $|T| = |S| + 1$. Replacing the right-hand side with this stronger demand, we get:

$$(1 + \delta^{\max})|S| < |S| + 1 \text{ , or}$$

$$\delta^{\max} < \frac{1}{|S|}.$$

By assumption $|S| < |T|$, which means that $|S|$ is maximally equal to $n - 1$. Choosing $\delta^{\max} < \frac{1}{n-1}$ therefore guarantees $w(S) < w(T)$ for any size of $|S|$. □

We can apply this result to construct a rescaling for any weight vector so that the feasible solution of minimal weight becomes a feasible solution of minimal size.

$$w'_j = 1 + \delta'_j = 1 + \frac{w_j - w^{\min}}{w^{\max} - w^{\min}} \times \frac{1}{n}. \tag{7.34}$$

Here, we defined δ'_j so that $0 \leq \delta'_j \leq 1/n$. This of course ensures that $\delta'_j < 1/(n-1)$.

To illustrate this scaling, return again to the example with $n = 3$, $\boldsymbol{w} = c(1, 1/3, 1/3)$, and feasible solutions $\{1\}$ $\{2, 3\}$ and $\{1, 2, 3\}$. After scaling, we get $\boldsymbol{w}' = (4/3, 1, 1)$, so $w'(\{1\}) = 4/3$, $w'(\{2, 3\}) = 2$, and $w'(\{1, 2, 3\}) = 10/3$.

7.5.2 Simplifying Conditional Validation Rules

Validation rules that combine linear (in)equality restrictions can put a severe performance burden on the error localization algorithm. Depending on the algorithm, the computational time needed for error localization may be as much as double with each extra conditional rule. In the case of MIP solving, a conditional rule means the introduction of dummy variables that increase the size of the problem. There are some cases, however, where conditional restrictions on combinations of linear inequalities can be simplified and rewritten into a set of pure inequalities. Here, we discuss two types of (partial) rule sets where simplification is possible.

Case 1

The first type is a ruleset of the form

$$E = \begin{cases} x \geq 0, \\ y \geq 0, \\ \texttt{if } x > 0 \texttt{ then } y > 0. \end{cases} \tag{7.35}$$

As a practical example, one can think of x as the *number of employees* and y as *wages paid*. Both cannot be negative, and if there are any employees, some wages must have been paid. Figure 7.1(a) shows in gray the valid region corresponding to this ruleset in

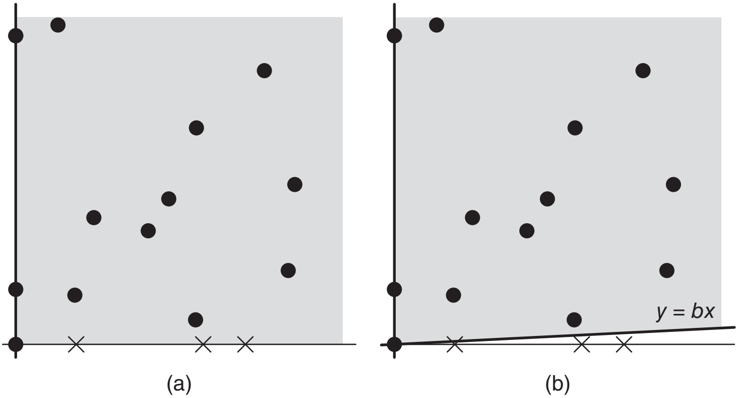


Figure 7.1 (a) In gray, the area of valid combinations (x, y) according to the rule set of Eq. (7.36), black dots represent valid combinations, and crosses represent invalid combinations. The y-axis is part of the valid area. (b) The valid area is defined by $x \geq 0$ and $y \geq bx$, with an appropriate choice for b.

the $x \times y$-plane. The black dots correspond to valid combinations of x and y, while the crosses correspond to invalid combinations. Now as depicted in 7.1(b), if we replace the conditional rule with $y > bx$, with some appropriately chosen b, the valid region shrinks somewhat, while all valid observations remain valid, and all invalid observations remain invalid. So, the question is what choice of b would be appropriate, given some observed pairs (x, y). This is stated in the following result:

Proposition 7.5.10 *For any finite set S of observations (x, y) subject to the ruleset E of Eq. (7.36), there is a $b > 0$ such that we can replace rule set E with a rule set*

$$F = \begin{cases} x \geq 0, \\ y \geq bx. \end{cases} \tag{7.36}$$

Proof: Note that the second rule implies $y \geq 0$ (since $b > 0$). Without loss of generality, we assume that $x >\geq 0$ and $y \geq 0$ for all $(x, y) \in S$. The second rule in F implies that when $x > 0$, then $y > 0$ for all $b > 0$. So, records that satisfy the second rule in F also satisfy the third rule in E. We need to show that the converse can also be constructed by choosing an appropriate value $b > 0$. We write $\min_+(\boldsymbol{x})$ and $\max_+(\boldsymbol{y})$ for the smallest and largest positive values of the observed x values and similarly for y. Now take

$$b = \frac{\min_+(\boldsymbol{x})}{\max_+(\boldsymbol{y})}, \tag{7.37}$$

we see that for each $(x, y) \in S$ that satisfies the third rule of E that

$$bx \leq \frac{\min_+(\boldsymbol{y})}{\max_+(\boldsymbol{x})} \underset{+}{\max(\boldsymbol{x})} \leq y, \tag{7.38}$$

which was what we needed to prove. □

In the above calculation, the smallest possible value for b was chosen so that the replacing rule still works. In practice, one can choose for$b$ some reasonable minimum. For example, suppose again that x stands for number of employees and y for wages paid. The smallest nonzero value for x would then be 1, and the value b is then the smallest wage that can be paid to an employee.

Case 2

Now, consider a second type of ruleset, given by

$$E' = \begin{cases} x \geq 0, \\ y \geq 0, \\ \texttt{if } x > 0 \texttt{ then } y > 0, \\ \texttt{if } y > 0 \texttt{ then } x > 0. \end{cases} \tag{7.39}$$

Again, we can think of x as *number of employees* and of y as *wages paid*. The ruleset then states that wages are paid if and only if the number of employees is positive. Figure 7.2(a) shows in gray the associated valid region in the xy-plane. Black dots again depict valid observations, while crosses depict invalid observations. In Figure 7.2(b), we see the valid regions bordered by the lines $y = cx$ and $y = bx$. The question is, again what values for b and c to choose?

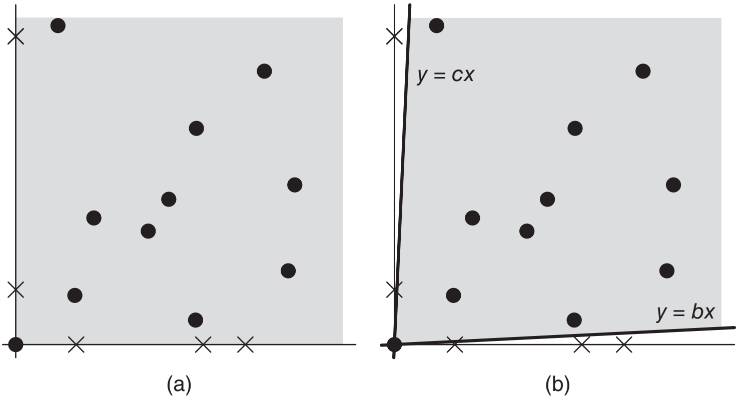


Figure 7.2 (a) In gray, the area of valid combinations (x, y) according to the rule set of Eq. (7.39). Black dots represent valid combinations, and crosses represent invalid combinations. The y-axis is part of the valid area. (b) The valid area is defined by $x \geq 0$, $y \geq bx$, and $y \leq cx$ with appropriate choices for b and c.

Proposition 7.5.11 *For any finite set S of observations (x, y) subject to the ruleset E of Eq. (7.36), there is a $b > 0$ and a $c \geq b > 0$ such that we can replace rule set E' with a rule set*

$$F' = \begin{cases} x \geq 0, \\ y \geq bx, \\ y \leq cx. \end{cases} \tag{7.40}$$

Proof: Replacing the third rule of E' follows from Proposition 7.5.10. We only consider records $(x, y) \in S$ that satisfy the first three rules in E'. From the fourth rule in F', we see that if $y > 0$, then $cx > 0$, which implies $x > 0$. So, records satisfying F' satisfy E'. We now need to find a c such that the converse is also true. Choose

$$c = \frac{\max_+(\boldsymbol{x})}{\min_+(\boldsymbol{y})}. \tag{7.41}$$

Every record $(x, y) \in S$ that satisfies the fourth rule in E' satisfies

$$y \leq \frac{\max_+(\boldsymbol{y})}{\min_+(\boldsymbol{x})} \min_+(\boldsymbol{x}) \leq cx.$$

Finally, we need to show that $c \geq b$. Using Eqs. (7.37) and (7.41), we get

$$c = \frac{\max_+(\boldsymbol{x})}{\min_+(\boldsymbol{y})} \geq \frac{\min_+(\boldsymbol{y})}{\max_+(\boldsymbol{x})} = b.$$

The equality $c = b$ only holds when $\min_+(\boldsymbol{y}) = \max_+(\boldsymbol{y})$ and $\min_+(\boldsymbol{x}) = \max_+(\boldsymbol{x})$, that is, when all records are identical. □

As an example, consider a dataset on hospitals, where x is the number of beds and y the number of patients occupying a bed. These numbers cannot be negative. We also assume that if a hospital has beds, then there must be a patient (hospitals are never empty), and if there is at least one patient, there must be a nonzero number of beds. This amounts to the ruleset

$$\begin{cases} \text{patients} \geq 0, \\ \text{beds} \geq 0, \\ \texttt{if}\ \text{patients} > 0\ \texttt{then}\ \text{beds} > 0, \\ \texttt{if}\ \text{beds} > 0\ \texttt{then}\ \text{patients} > 0. \end{cases}$$

Using the above proposition, we can replace this ruleset with the simpler

$$\begin{cases} \text{patients} \geq 0, \\ \text{beds} \geq \text{factor}_1 \times \text{patients}, \\ \text{beds} \leq \text{factor}_2 \times \text{patients}. \end{cases}$$

Here, factor_1 is the smallest number of beds per patient, which we can choose equal to 1. factor_2 is the largest number of beds per patient or the inverse of the smallest occupancy rate that can reasonably be expected for a hospital.

Generalizations

Finally, we state a few simple generalizations of the above simplifications. The proofs of these generalizations follow the same reasoning as the proofs for Propositions 7.5.10 and 7.5.11 and will not be stated here.

Proposition 7.5.12 (Generalization of 7.5.10) *The rule set*

$$\begin{cases} x \geq 0, \\ y \geq 0, \\ \texttt{if}\ x > A\ \texttt{then}\ y > 0, \end{cases}$$

with $A \geq$ a constant, can be replaced by

$$\begin{cases} x \geq 0, \\ y \geq 0, \\ y \geq b(x - A), \end{cases}$$

where $0 < b \leq \min_+(\boldsymbol{y})/\max_+(\boldsymbol{x} - A)$.

Proposition 7.5.13 (Generalization of 7.5.11) *The rule set*

$$\begin{cases} x \geq 0, \\ y \geq 0, \\ \texttt{if}\ x > A\ \texttt{then}\ y > 0, \\ \texttt{if}\ y > B\ \texttt{then}\ x > 0 \end{cases}$$

with $A \geq 0$ and $B \geq 0$ constants can be replaced by

$$\begin{cases} x \geq 0, \\ y \geq 0, \\ y \geq b(x - A), \\ y \leq cx + B \end{cases}$$

with $0 < b \leq \max_+(\boldsymbol{x} - A)/\min_+(\boldsymbol{y} - B) \leq c$.

7.6 Conclusion

When a record is invalidated by a set of record scoped rules, it can be a problem to identify which fields are likely to be responsible for the error. In this chapter, we have seen that these fields can be indentified using the Fellegi–Holt formalism. This error localization problem for linear, categorical, and mixed-type restrictions can be formulated in terms of a MIP-problem. Mixed-type restrictions can be seen as a generalization of both record-wise linear and categorical restrictions. MIP-problems can be solved.

Although MIP-problems can be solved by readily available software packages, there may be a trade-off in numerical stability with respect to a branch-and-bound approach. This holds especially when typical default settings of such software are left unchanged and data records and/or linear coefficients of the restriction sets cover several orders of magnitude. In this paper, we locate the origin of these instabilities and provide some pointers to avoiding such problems.

The `lp_solve` package has now been introduced as a MIP solver back end to our `errorlocate` R package for error localization and rule management. Our benchmarks indicate that in generic cases where no prior knowledge is available about which values in a record may be erroneous (as may be expressed by lower reliability weights), the MIP method is much faster than the previously implemented branch-and-bound-based algorithms. On the other hand, the branch-and-bound-based approach returns extra information, most notably the number of equivalent solutions. The latter can be used as an indicator for quality of automatic data editing.

Appendix 7.A: Derivation of Eq. (7.33)

In the Phase I–Phase II simplex method, Phase I is aimed to derive a valid solution, which is then iteratively updated to an optimal solution in Phase II. Here, we derive a Phase I solution, specific for error localization problems.

Recall the tableau of Eq. (7.32); for clarity, the top row indicates to what variables the columns of the tableau pertain to.

$$\left[\begin{array}{c|cccccc|c} f & \boldsymbol{x}^T & \boldsymbol{\Delta} & s_x & s_+ & s_- & s_\Delta & \\ 1 & \mathbf{0} & -\boldsymbol{w}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\ \hline 0 & \boldsymbol{A} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{b} \\ 0 & \boldsymbol{I} & -\boldsymbol{M} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \boldsymbol{x}^0 \\ 0 & \boldsymbol{I} & \boldsymbol{M} & \mathbf{0} & \mathbf{0} & -\boldsymbol{I} & \mathbf{0} & \boldsymbol{x}^0 \\ 0 & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{I} & \mathbf{1} \end{array}\right].$$

Here, the $\boldsymbol{s}_i$ are slack or surplus variables, aimed to write the original inequality restrictions as equalities. The $\boldsymbol{s}_x$ are used to rewrite restrictions on observed variables, the $\boldsymbol{s}_\pm$ to write the upper and lower limits on $\boldsymbol{x}$ as equalities, and $\boldsymbol{s}_\Delta$ to write the upper limits on $\boldsymbol{\Delta}$ as equalities.

Observe that the above tableau *almost* suggests a trivial solution. If we choose $\boldsymbol{s}_x = \boldsymbol{b}$, $\boldsymbol{s}_\pm = \pm\boldsymbol{x}^0$, and $\boldsymbol{s}_\Delta = \mathbf{1}$, we may set $(\boldsymbol{x}, \boldsymbol{\Delta}) = \mathbf{0}$. However, recall that we demand all variables to be nonnegative, so $\boldsymbol{s}_-$ may not be equal to $-\boldsymbol{x}^0$. To resolve this problem, we introduce a set of *artificial* variables $\boldsymbol{a}_-$ and extend the restrictions involving $\boldsymbol{s}_-$ as follows:

$$\boldsymbol{x} + M\mathbf{1} - \boldsymbol{s}_- + \boldsymbol{a}_- = \boldsymbol{x}_0.$$

This yields the following tableau:

$$\left[\begin{array}{c|ccccccc|c}
f & \boldsymbol{x}^T & \boldsymbol{\Delta} & \boldsymbol{s}_x & \boldsymbol{s}_+ & \boldsymbol{s}_- & \boldsymbol{s}_\Delta & \boldsymbol{a}_- & \\
1 & \mathbf{0} & -\boldsymbol{w}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\
\hline
0 & \boldsymbol{A} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{b} \\
0 & \boldsymbol{I} & -\boldsymbol{M} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{x}^0 \\
0 & \boldsymbol{I} & \boldsymbol{M} & \mathbf{0} & \mathbf{0} & -\boldsymbol{I} & \mathbf{0} & \boldsymbol{I} & \boldsymbol{x}^0 \\
0 & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{1}
\end{array}\right].$$

The essential point to note is that the tableau now contains a unit matrix in columns 4, 5, 7, and 8, so choosing $\boldsymbol{s}_x = \boldsymbol{b}$, $\boldsymbol{s}_+ = \boldsymbol{x}^0$, $\boldsymbol{s}_\Delta = \mathbf{1}$, $\boldsymbol{a}_- = \boldsymbol{x}^0$, and $(\boldsymbol{x}, \boldsymbol{\Delta}, \boldsymbol{s}_-) = \mathbf{0}$ is a solution obeying all restrictions. The artificial variables have no relation to the original problem, so we want them to be zero in the final solution. Since the tableau represents a set of linear equalities, we are allowed to multiply rows with a constant and add and subtract rows. If this is done in such a way that we are again left with a unit matrix in part of the columns, a new valid solution is generated. Here, we add the fourth row to row 3 and subtract $\boldsymbol{M}^{-1}$ times the fourth row from the last row. Row 4 is multiplied with $\boldsymbol{M}^{-1}$. This gives

$$\left[\begin{array}{c|ccccccc|c}
f & \boldsymbol{x}^T & \boldsymbol{\Delta} & \boldsymbol{s}_x & \boldsymbol{s}_+ & \boldsymbol{s}_- & \boldsymbol{s}_\Delta & \boldsymbol{a}_- & \\
1 & \mathbf{0} & -\boldsymbol{w}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \\
\hline
0 & \boldsymbol{A} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{b} \\
0 & 2\boldsymbol{I} & \mathbf{0} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \boldsymbol{I} & 2\boldsymbol{x}^0 \\
0 & \boldsymbol{M}^{-1} & \boldsymbol{I} & \mathbf{0} & \mathbf{0} & -\boldsymbol{M}^{-1} & \mathbf{0} & \boldsymbol{M}^{-1} & \boldsymbol{M}^{-1}\boldsymbol{x}^0 \\
0 & -\boldsymbol{M}^{-1} & \mathbf{0} & \mathbf{1} - \boldsymbol{M}^{-1}\boldsymbol{x}^0 & & & & &
\end{array}\right].$$

This almost gives a new valid solution, except that the first row, representing the objective function, has not vanished at the third column. However, we may re-express the objective function in terms of the other variables, namely, using row 4, we have

$$\boldsymbol{w}^T\boldsymbol{\Delta} = \boldsymbol{w}^T\boldsymbol{M}^{-1}\boldsymbol{x}^0 + \boldsymbol{w}^T\boldsymbol{M}^{-1}(\boldsymbol{s}_- - \boldsymbol{x} - \boldsymbol{a}_-).$$

Substituting this equation in the top row of the tableau shows that $\boldsymbol{\Delta} = \boldsymbol{M}^{-1}\boldsymbol{x}^0$, $\boldsymbol{s}_x = \boldsymbol{b}$, $\boldsymbol{s}_+ = 2\boldsymbol{x}^0$, $\boldsymbol{s}_\Delta = \mathbf{1} - \boldsymbol{M}^{-1}\boldsymbol{x}^0$, and $(\boldsymbol{x}, \boldsymbol{s}_-, \boldsymbol{a}_-) = \mathbf{0}$ represent a valid solution. Since the artificial variables have vanishing values, we may now delete the corresponding column and arrive at the tableau of Eq. (7.33).

8

Rule Set Maintenance and Simplification

8.1 Quality of Validation Rules

Since data validation is an intrinsic part of data cleaning, it is worthwhile to treat their fundamental building blocks, the data validation rules, as separate objects of study. In particular, one can consider a set of validation rules and wonder whether they are both effective and efficient with respect to the goal of data cleaning: to create a dataset that is fit for a particular analytic purpose. Regardless of this purpose, one would at least like a set of validation rules to be internally consistent, free of redundancies, and understandable. Having a grip on these properties is especially important in production systems of which the rule sets are updated and extended over time, possibly by multiple persons. In our experience, rule sets tend to grow organically over time, and it is difficult to weed out parts that make a rule set less efficient or effective than it could be.

8.1.1 Completeness

The effectiveness of a set of validation rules is largely determined by its completeness. On one hand, an incomplete set will allow combinations of values in a dataset that are impossible for reasons of logic or simple physics. An overly complete set, on the other hand, may erroneously label actually valid combinations as invalid.

Since completeness concerns the extent to which relevant domain knowledge has been condensed into validation rules, it is hard to assess it automatically. A simple measure that gives some insight into completeness is a check on which variables in the dataset are covered by the rules. This is supported by the `variables` function in the `validate` package. It may help to check whether all important variables have some checks on them, but it is still a very shallow metric of quality.

A better way to assess completeness is then to think of validation rules as a form of source code. Methodology for assessing source code and practices for maintaining its quality are well developed, and it makes sense to reuse those ideas in the context of validation rules. Important practices include documenting code inline and maintaining its development history using a version control system. These practices can easily be copied for maintaining validation rules, and the presence of documentation, especially, forms a prerequisite for assessing completeness.

The basis of finding out whether a set of rules is (overly) complete with respect to a body of domain knowledge is by confronting it with knowledgeable peers. There are many ways to organize this, some of which will be mentioned here. One way is an independent peer review of the rule set where independent experts go through the set line by line and check whether each rule is justified and also discuss whether something is missing. A second method is to develop a rule set in a pair programming setting, where two developers work together on a rule set. Finally, one can organize a code review session, projecting the rules on a screen and presenting and explaining them one by one for a group of peers.

Finally, it is important to realize that the effectiveness of rules may change because of contextual changes. For example, a new law may render certain age checks unjustified, or a new data source may introduce new variables with extra information, adding new restrictions. In short, since the world and our understanding of it is in constant flux, it is advisable to review a repeatedly used rule set with some frequency.

8.1.2 Superfluous Rules and Infeasibility

A rule set can contain rules or parts of rules that are redundant, or in other ways unnecessary or unwanted. As an example of redundancy, consider a rule set where both `x > 0` and `x > 1` are present in the set, the former rule is redundant with respect to the latter. Redundancy is not the same as being overcomplete, since the latter refers to completeness with respect to covering domain knowledge. Since the number of rules that are present in a rule set can strongly hamper the performance of the system that consumes them, it is still desirable to avoid them where possible. More subtle cases where rules are (partially) superfluous will be discussed further on in this chapter.

A rule set is infeasible when there is no dataset that can possibly satisfy all demands simultaneously. For example, a rule set containing both `x > 0` and `x < -1`. Needless to say, an infeasible rule set is useless for any purpose.

Rule sets can contain redundancies and infeasibilities in complicated and implicit ways. However, in contrast to completeness, mathematical tools to weed out redundancy and infeasibility are available, at least for certain types of rule sets. The remainder of this chapter is devoted to such methods.

8.2 Rules in the Language of Logic

If we disregard the notion of missing values, a validation rule can be interpreted as a logical proposition: a statement that is either true or false. This means that familiar methods from logic can be used to combine and manipulate rule sets. In the following, we shall denote a validation rule with r and its value after evaluation on a dataset $\boldsymbol{x}$ with $r(\boldsymbol{x})$ (true or false).

The first 'manipulation' is that of logical conjunction ($\wedge$). Given a set of rules $S = \{r_1, r_2, \ldots, r_n\}$, we expect a dataset $\boldsymbol{x}$ to satisfy all rules in S. Whether $\boldsymbol{x}$ is valid according to S is therefore the conjunction of each rule r_i applied to $\boldsymbol{x}$, in notation:

$$S(\boldsymbol{x}) = r_1(\boldsymbol{x}) \wedge \cdots \wedge r_n(\boldsymbol{x}) = \bigwedge_i^n r_i(\boldsymbol{x}). \tag{8.1}$$

To come up with algorithms that weed out redundancies and infeasibilities, we need to limit the types of rules under discussion. Here, each rule $r_i(\boldsymbol{x})$ is either a 'fundamental' or atomic clause that cannot be further dissected using pure logic manipulations, or it is a combination of such clauses that are connected by logical 'or' (disjunction). In notation, we have

$$r_i(\boldsymbol{x}) = C_i(\boldsymbol{x}) \text{ or } r_i(\boldsymbol{x}) = \bigvee_j C_i^j(\boldsymbol{x}). \tag{8.2}$$

Here, C_i denotes an atomic clause. To be able to translate algorithms to mixed-integer programming problems that can be solved in practice, we further limit our discussion to atomic clauses that can be evaluated on single records $\boldsymbol{x}$, where the x_j may be numeric or categorical. An atomic clause is then either a linear equality or inequality on numerical values in $\boldsymbol{x}$ or a set membership statement for combinations of categorical values in $\boldsymbol{x}$.

That is, each atomic clause C can be written in one of the following forms:

$$C_i^j(\boldsymbol{x}) = \begin{cases} \boldsymbol{a}^T\boldsymbol{x} \leq b, \\ \boldsymbol{a}^T\boldsymbol{x} = b, \\ x_j \in F_{ij} \text{ with } F_{ij} \subseteq D_j, \\ x_j \notin F_{ij} \text{ with } F_{ij} \subseteq D_j. \end{cases} \tag{8.3}$$

Here, F_{ij} denotes a subset of the allowed categories D_j for the jth value of $\boldsymbol{x}$. Consider, for example, the rule

```
IF  married == TRUE THEN age >= 15
```

We first need to realize that this rule can be rewritten (see also below) as

```
NOT married OR age >= 15
```

In the notation of Eq. (8.3), we can write

$$(\text{married} \notin \{\texttt{TRUE}\}) \vee (\text{age} \geq 15).$$

We shall refer to rules in the forms of Eq. (8.3) as the *canonical form.*

8.2.1 Using Logic to Rewrite Rules

In rewriting and simplifying rules into a canonical form, the following classical logical equalities are of use.

A conditional statement/implication equals an OR statement

$$A \implies B = \neg A \vee B. \tag{8.4}$$

In R code:

```
if (A) B == !A | B
```

A double negation is a confirmation:

$$\neg(\neg A) = A. \tag{8.5}$$

In R code:

```
!(!A) == A
```

A negation of a logical OR statement is equal to the conjunction of negations:

$$\neg(A \vee B) = \neg A \wedge \neg B. \tag{8.6}$$

In R code:

```
!(A | B) == !A & !B
```

A negation of a logical AND statement is equal to the negation of conjunctions:

$$\neg(A \wedge B) = \neg A \vee \neg B. \tag{8.7}$$

In R code:

```
!(A & B) == !A | !B
```

Exercises for Section 8.2

Exercise 8.2.1 *Rewrite the following R statement as a disjunction*

```
if (x > 0 & y > 0) z >0
```

8.3 Rule Set Issues

A closer look at the canonical form of (8.1)–(8.3) shows that a rule set S can have one or more of the following issues, each amounting in some need of simplification.

Infeasible Rule Set. $S(\boldsymbol{x})$ may be false for all $\boldsymbol{x}$.

Partial Infeasibility. $\boldsymbol{x}$ may be unintentionally too constrained (see examples below).

Fixed Value. $S(\boldsymbol{x})$ may restrict a x_i to one value.

Redundant Rule. $r_i(\boldsymbol{x}) = \mathsf{true}$ may be implied by $\bigwedge_{i \neq j} r_i$.

Non-Relaxing Clause. a $C_i^j = \mathsf{false}$ may be implied by $S(\boldsymbol{x})$.

Non-Constraining Clauses. a $C_i^j = \mathsf{true}$ may be implied by $S(\boldsymbol{x})$.

Solving one or more of these issues results in a simplified rule set S. In the following sections, we describe each issue in more detail.

8.3.1 Infeasible Rule Set

A necessary requirement for a rule set is that it must be feasible.

Definition 8.3.1 *A rule set S is feasible if it is possible to have a record $\boldsymbol{x} = (x_1, \ldots, x_n)$ for which $S(\boldsymbol{x}) = \mathsf{true}$, that is, $\boldsymbol{x}$ complies to each rule r_i.*

If rule set S is infeasible, then it is not possible to validate data with it, because $S(\boldsymbol{x})$ always is false. An infeasible rule set contains contradictory rules. This may seem unlikely, but with organic growth of a rule set it often happens. Note that each rule set can be made infeasible by adding just one contradiction. This means that the more rules a rule set has, the higher the chance that it contains a contradiction. If validation rules are not managed explicitly, the contradictions may be implicit and incorrectly invalidate and adjust valid data.

Example 8.3.2 *A minimal infeasible rule set is:*

```
r1: age > 25
r2: age == 18
```

Either `r1` *or* `r2` *is correct, but not both.*

If a rule set is infeasible, the action to be taken is to find out which rules are contradictory: what is the smallest subset of rules we can remove to make the rule set feasible? Note that often there is a choice in which rules should be removed. In the example above, either `r1` or `r2` must be removed. It is helpful to detect which rules conflict with the smallest removable subset.

Partial Infeasibility

In some cases, rules may unintentionally render parts of the value domain of $\boldsymbol{x}$ infeasible. Bruni and Bianchi (2012) refer to this as partial inconsistency.

Example 8.3.3 *Unintended exclusion: the following rules were independently added to the rule set:*

```
r1: if(gender == "male") income > 2000
r2: if(gender == "male") income < 1000
```

Rule `r1` *and* `r2` *seem contradictory, but technically they are not: the rule set is feasible because it is possible to have records with* `gender != "male"`*. However, the intention of both rules is contradictory because it was not meant to encode* `gender != "male"`*.*

Combinations of numeric rules can also restrict numeric variables unintentionally:

Example 8.3.4 *Numeric partial infeasibility*

```
r1: tax >= 0.5 * income
r2: tax <= 1000
```

This rule set is feasible but excludes all records where `income` *is higher than 2000, which is probably unintended. The intention of the rule is that* `tax` *depends on* `income` *but maximally is 1000.*

A formulation of this intention results in the following rule set:

```
r1: if (tax < 1000) tax >= 0.5 * income
r2: tax <= 1000
```

which does not have the excluding behavior of the original rule set.

It is difficult to find multivariate partial infeasibility algorithmically. It is, however, possible to detect the univariate allowed ranges of a rule set, helping the analyst to check if variables are overly constrained by the current rule set.

8.3.2 Fixed Value

A rule set S may constrain the value of a variable x_i to a single value for all valid $\boldsymbol{x}$. In that case a rule that fixates the variable should be added to S, and x_i should be substituted with its value in all rules in which it occurs.

Example 8.3.5 *Trivial fixed value*

```
r1: x >= 0
r2: x <= 0
```

The only solution to this rule set is if `x == 0`*. This rule set can therefore be replaced with*

```
r1: 0 >= 0
r2: 0 <= 0
r3: x == 0
```

which can be further simplified by removing redundant rules `r1` *and* `r2`*.*

A more typical situation is where a combination of multiple rules generates a fixed value.

Example 8.3.6 `x3` *is fixed in the following rule set:*

```
r1: x1 + x2 + x3 == 0
r2: x1 + x2 >= 0
r3: x3 >= 0
```

This rule set implies that $x3$ is 0. The rule set can therefore be simplified by adding this constraint and substituting `x3=0` *in the other rules, which results in the following rule set:*

```
r1: x1 + x2 == 0
r2: x1 + x2 >= 0
r3: 0 >= 0
r4: x3 == 0
```

which can be further simplified by removing redundant rules `r2` *and* `r3`

If a rule set generates a fixed value, a variable is turned into a constant. If this is undesired, the fixed value is a special case of partial infeasibility. Sometimes the dataset to be analyzed is a subset of the data in which one or more variables are pinned down to a value that is known to be correct. For example, an analysis may restrict itself to all female respondents, and it is known that the gender variable is correct. In that case, even though the rule set does not generate a fixed value, it is still beneficial to substitute the pinned down variables because it simplifies the rule set, making correction procedures more computationally efficient and reasoning about the rules easier.

As demonstrated in the examples, substitution often generates more simplifications by making rules redundant. Substitution should therefore be followed by removing redundant rules.

8.3.3 Redundant Rule

Definition 8.3.7 *A redundant rule r in rule set S is a rule that can be removed from S without changing the validation results for each possible record $\boldsymbol{x}$.*

A rule is redundant if rule r is implied by the other rules from S and therefore is true if the other rules are true.

Example 8.3.8 *Redundant rule implied by one rule*

```
r1: x >= 1
r2: x >= 0
```

Rule `r2` is implied by `r1`: each record that is validated by `r1` is automatically valid for `r2`. The rule set can be simplified by removing redundant rule `r2`.

Example 8.3.9 *Redundant rule implied by two rules*

```
r1: x >= 1
r2: y >= 1
r3: x + y >= 0
```

Rule `r3` is implied by `r1` and `r2`. This rule set can be simplified by removing `r3`.

8.3.4 Nonrelaxing Clause

Recall that conditional rules can be written as disjunctive clauses:

```
if ( x > 0 ) y > 0
```

is equal to

```
x <= 0 | y > 0
```

Definition 8.3.10 *A nonrelaxing clause C_i^j is a component of a conditional rule r_i that, in combination with rule set S, always is* false.

Since a nonrelaxing clause (Dillig *et al.*, 2010) always is false, it can be removed without consequences from a disjunctive rule r_i.

Example 8.3.11 *Nonrelaxing clause*

```
r1: x <= 0 | y > 0
r2: x > 0
```

The first clause of `r1` always is false *and therefore can be removed. This results in the following rule set:*

```
r1: y > 0
r2: x > 0
```

8.3.5 Nonconstraining Clause

It is possible that a clause of a conditional rule r_i in combination with other rules from S always is true. In that case the conditional rule can be replaced with the clause. This is a nonconstraining clause (Dillig *et al.*, 2010).

Definition 8.3.12 *A nonconstraining clause C_i^j is a component of a conditional rule r_i that, in combination with rule set S, always is* true.

This seems a redundant rule but it is not. Contrary to its name, the other components of r_i together with the other rules in S constrain C_i^j to be true. Removing the entire conditional rule would remove this information, and therefore rule r_i should be replaced with its component C_i^j.

Example 8.3.13 *Excluding premises*

```
r1: if (x > 0) y > 0
r2: if (x < 1) y > 1
```

which is equivalent to:

```
r1: x <= 0 | y > 0
r2: x >= 1 | y > 1
```

`x <= 0` and `x >= 1` are excluding components, so either `y > 0` or `y > 1` must be true, which implies that `y > 0` is true.

Replacing `r1` with this clause results in the following rule set:

```
r1: y > 0
r2: x >= 1 | y > 1
```

8.4 Detection and Simplification Procedure

To keep rule sets maintainable, understandable, and to improve performance of systems that consume them, it is desirable to resolve the issues of the previous section. The general procedure that can be summarized to detect and simplify a rule set is described in Procedure 8.4.1. This procedure is based on Daalmans (2015) and extended with a procedure to resolve conflicting rules. The overall procedure starts with checking for feasibility, because that is a requirement for validation and for the further simplification: simplifying an infeasible rule set generates nonsense. The procedure ends with removing redundant rules because the other simplification steps may change rules into redundant rules, for example, by substitution. For detection of infeasibilities and redundancies, we will rely on mixed-integer programming (MIP) as a basic building block.

Procedure 8.4.1 Rule set simplification

Feasible?:

1 Check if the rule set is feasible. If not, remove the smallest subset of rules such that the remaining rule set is feasible.
2 Check for partial infeasibility and adjust rules if necessary.

Simplify rules:

1 Find and substitute fixed values.
2 Remove nonrelaxing clauses.
3 Remove nonconstraining clauses.
4 Remove redundant rules.

8.4.1 Mixed-Integer Programming

R packages `validate` and `errorlocate` allow for rule manipulation but do not offer simplification methods for rules. Simplifying rules can be done using the R package validatetools, which implements the procedures described in this chapter. In the following sections, we assume that the rules are of the type allowed by `errorlocate`, so they can be manipulated by validatetools.

A MIP problem is an optimization problem that can be written in the form:

$$\begin{aligned} \text{Minimize}\, f(\boldsymbol{z}) &= \boldsymbol{c}^T\boldsymbol{z}; \\ \text{s.t.}\ \boldsymbol{Rz} &\leq \boldsymbol{d}, \end{aligned} \tag{8.8}$$

where $\boldsymbol{c}$ is a constant vector and $\boldsymbol{z}$ a vector consisting of real and integer coefficients. In Section 7.3.1 is shown how rules concerning numerical and/or categorical variables can be reformulated in terms of a mixed-integer problem for the purpose of error localization.

8.4.2 Detecting Feasibility

Checking for feasibility is standard functionality for a mixed-integer program solver. The function to be optimized is $\boldsymbol{c} = 0$, with $\boldsymbol{Rx} \leq \boldsymbol{d}$ the MIP translated rule set.

$$\begin{aligned} \text{Minimize}\, f(\boldsymbol{x}) &= 0; \\ \text{s.t.}\ \boldsymbol{Rx} &\leq \boldsymbol{d}^{\cdot} \end{aligned} \tag{8.9}$$

Choosing $\boldsymbol{c} = 0$ means that every, if any, $\boldsymbol{x}$ that conforms to $\boldsymbol{Rx} \leq \boldsymbol{d}$ is a solution. The rule set can be given to the MIP solver, which will report whether this problem is infeasible. If this is the case, the rule set is infeasible, at least to within the bounds of the solver's numerical accuracy.

8.4.3 Finding Rules Causing Infeasibility

If the rule set is infeasible, the question is to find out what the minimal subset of rules needs to be removed to make the rule set feasible. The MIP formulation is well known and has among others been discussed by Kim and Ahn (1999) and Mousseau *et al.* (2003):

$$\begin{aligned} \text{Minimize}\, f(\boldsymbol{x}, \boldsymbol{\ell}) &= \sum_i \ell_i; \\ \text{s.t.}\ \boldsymbol{Rx} &\leq \boldsymbol{d} + M\boldsymbol{\ell} \end{aligned} \tag{8.10}$$

with M a suitable large value and ℓ a binary vector encoding if a rule should be excluded from the set S to make it feasible. Note that this formulation considers each rule r_i in S to be a soft constraint, which may be violated. The objective function of this problem is to minimize the number of rules to remove from rule set S. The solution to this problem may be degenerate: meaning that there is more than one solution. Furthermore, it is good to detect which rules conflict with the rules that are to be removed: this provides helpful information for the analyst: it may help in choosing in which set of rules is the "incorrect" set and should be removed.

8.4.4 Detecting Conflicting Rules

To detect conflicting rules, we generalize Eq. (8.10) by introducing weights for the rules in the objective function. We minimize the following MIP problem:

$$\begin{aligned} \text{Minimize}\, f(\boldsymbol{x}, \boldsymbol{\ell}) &= \sum_i w_i J_i; \\ \text{s.t.}\ \boldsymbol{Rx} &\leq \boldsymbol{d} + M\boldsymbol{J}. \end{aligned} \tag{8.11}$$

If we give the rules causing the infeasibility, where $\ell_i = 1$ is a weight of $w \geq N$, with N the number of rules, and the other rules ($\ell_i = 0$) a weight of 1, the MIP solver will find the rules ($\boldsymbol{J}$) that are in conflict with the rules that were marked to cause the infeasibility ($\boldsymbol{\ell}$).

8.4.5 Detect Partial Infeasibility

A recipe for detecting univariate partial infeasibility is to find the minimum and maximum value for each variable x_i. The MIP formulation is:

$$\begin{aligned}
&\text{Minimize } f(\boldsymbol{x}) = x_i;\\
&\qquad \text{s.t. } \boldsymbol{Rx} \le \boldsymbol{d}\\
&\text{and}\\
&\text{Maximize } f(\boldsymbol{x}) = x_i;\\
&\qquad \text{s.t. } \boldsymbol{Rx} \le \boldsymbol{d}.
\end{aligned} \tag{8.12}$$

If the minimum and maximum values are in conflict with expectations on the data, this means that the rule set contains a partial infeasibility. This is a manual decision. The procedure finds the minimum and maximum values of each x_i; however, it does not detect gaps in this range. This procedure is therefore not a general procedure for detecting univariate partial infeasibility.

This detection also includes detecting the possible values a categorical variable v_i can take: in the MIP formulation, each category is encoded in a binary dummy variable. If a category has maximum of 0, it is always invalid. If its minimum is 1, it is a fixed value.

As practical matter, we note that it is a good idea to include the minimum and maximum values of each variable as boundary conditions in the formulation of the MIP problem, because boundary conditions are treated more efficiently by MIP solvers.

8.4.6 Detect Fixed Values

Fixed values can be substituted in the rule set, which simplifies it, sometimes considerably. Detecting a fixed value is a special case of the partial infeasibility detection:

$$\begin{aligned}
&\text{Minimize } f(\boldsymbol{x}) = x_i;\\
&\qquad \text{s.t. } \boldsymbol{Rx} \le \boldsymbol{d}\\
&\text{and}\\
&\text{Maximize } f(\boldsymbol{x}) = x_i;\\
&\qquad \text{s.t. } \boldsymbol{Rx} \le \boldsymbol{d}.
\end{aligned} \tag{8.13}$$

If the minimum and maximum solutions are equal, we have found a fixed value, which then can be added to the rule set and be substituted. Note that it may be necessary to remove redundant rules after substitution.

8.4.7 Detect Nonrelaxing Clauses

A nonrelaxing clause C_i^j is always false for all valid $\boldsymbol{x}$. If we add C_i^j to S, and S is infeasible, then the clause is a nonrelaxing clause. This observation is used in Procedure 8.4.2. Following this procedure, we check for each C_i^j that is part of a nonatomic rule r_i the feasibility of the following MIP problem:

$$\begin{aligned}
&\text{Minimize } f(\boldsymbol{x}) = 0;\\
&\qquad \text{s.t. } \boldsymbol{R'x} \le \boldsymbol{d}
\end{aligned} \tag{8.14}$$

with $\boldsymbol{R'} = \boldsymbol{R} \wedge C_i^j$. If this problem is infeasible, we remove clause C_i^j from rule r_i.

Procedure 8.4.2 Remove nonrelaxing clause

```
For each conditional rule $r_i = \bigvee_j C_i^j$ with $r_i \in S$:

For each component $C_i^j$
if $S \wedge C_i^j$ is infeasible, remove $C_i^j$ from $r_i$.
```

8.4.8 Detect Nonconstraining Clauses

A nonconstraining clause C_i^j is always true for all valid $\boldsymbol{x}$. This implies that the negation of this clause must be false. If we add the negation of C_i^j, that is, $\neg C_i^j$, to S, and S becomes infeasible, then C_i^j is a nonconstraining clause. This observation is used in Procedure 8.4.3. Following the procedure, we check for each C_i^j that is part of a nonatomic rule r_i the feasibility of the following MIP problem:

$$\begin{aligned} \text{Minimize}\, f(\boldsymbol{x}) &= 0; \\ \text{s.t. } \boldsymbol{R}'\boldsymbol{x} &\leq \boldsymbol{d} \end{aligned} \tag{8.15}$$

with $\boldsymbol{R}' = \boldsymbol{R} \wedge \neg C_i^j$. If this problem is infeasible, we replace rule r_i with clause C_i^j.

Procedure 8.4.3 Removing nonconstraining clauses

```
For each conditional rule $r_i = \bigvee_j C_i^j$ with $r_i \in S$:

1 For each component $C_i^j$
2 if $S \wedge \neg C_i^j$ is infeasible, replace $r_i$ with $C_i^j$
```

8.4.9 Detect Redundant Rules

A redundant rule r_k is always true if $\wedge_{i \neq k} r_i$ is true. If we replace rule r_k with its negation $\neg r_k$ and S becomes infeasible, then rule r_k is redundant. This observation is used in Procedure 8.4.4. Following this procedure, we check for each r_k the feasibility of the following MIP problem:

$$\begin{aligned} \text{Minimize}\, f(\boldsymbol{x}) &= 0; \\ \text{s.t. } \boldsymbol{R}'\boldsymbol{x} &\leq \boldsymbol{d} \end{aligned} \tag{8.16}$$

with $\boldsymbol{R}' = \boldsymbol{R} \setminus r_k \wedge \neg r_k$. If this problem is infeasible, we remove rule r_k from S.

Procedure 8.4.4 Remove redundant rules

```
For each rule $r_k \in S$:

1 Replace $r_k$ with its negation: $S' = S \setminus r_k \wedge \neg r_k$.
2 If $S'$ is infeasible, $r_k$ is redundant and can be removed.
```

Often, it helps to find out which rules cause rule r_k to be redundant. We can find this out using the following procedure:

$$\begin{aligned} \text{Minimize } f(\boldsymbol{x}, \boldsymbol{\ell}) &= \sum_i w_i \ell_i; \\ \text{s.t. } \boldsymbol{R}'\boldsymbol{x} &\leq \boldsymbol{d} + M\boldsymbol{\ell} \end{aligned} \tag{8.17}$$

with $\boldsymbol{R}' = \boldsymbol{R} \setminus r_k \wedge \neg r_k$, $w_k \geq N$, and $w_{j \neq k} = 1$. $\boldsymbol{\ell}$ indicates the rules that are conflicting with the $\neg r_k$, which are the rules that imply r_k: this is the rule subset for which $\ell_j = 1$.

8.5 Conclusion

In production environments that implement reproducible data-cleaning steps, validation rules about the data evolve. Rule housekeeping is therefore a necessary chore. This chapter describes common problems with evolving rule sets and provides recipes for solving them using MIP techniques.

R package validatetools provides an implementation of all techniques discussed, and works nicely with validation rule sets that are defined with `validate` and that conform to the rule restrictions of `errorlocate`.

9

Methods Based on Models for Domain Knowledge

9.1 Correction with Data Modifying Rules

Domain experts who are familiar with a recurring statistical analyses acquire knowledge about errors that are typical for that process. This knowledge is often put to use by writing data processing scripts, for example, in SQL, R, or Python that modify the data to accommodate for such errors. The statements in these scripts can be fairly simple, and in fact they are often a sequence of rules that can be stated as

`if` some condition is met `then` alter one or more values. (9.1)

The aim of each such statement is to resolve an error (detected in the condition) for which an obvious solution or other amendment can be found. Some examples of data modifying rules that we have encountered in practice include the following:

- If *total revenues* per *total employees (in full-time equivalent)* is less than 1000, all variables from the income slate must be divided by 1000.
- If *other staff costs* is less than zero, replace with its absolute value.
- If *general expenditures* exceeds one half of *total expenditures*, then *total other expenditures* is set to missing.

In the last case, a value is set to missing so that a better value can later be estimated by an imputation method.

This method of data correction is a common practice and may have a significant effect on the final results. In a study of Pannekoek *et al.* (2014), the effect of various data cleaning steps on two different multistep data cleaning processes was investigated. It was found that of all cells that were altered during the process, a majority was touched by data modifying rules. It was also shown that the change in mean value for some key variables may change significantly by application of such rules, for example, because they may correct for unit of measure errors.

The main advantage of data correction through such scripts is that it allows for a quick way to translate knowledge to (well-documented) code. Possible disadvantages include lack of reproducibility when code is not under a decent version control system or when manipulations are done by hand. Also, each statement solves a single problem, not taking into account other such statements or validation rules. Indeed, it is our experience that in some cases these scripts do more damage to a data set, in terms of the number of violated validation rules, than they do good. One would therefore like to keep track of such modifications, and at least have some knowledge about whether they interfere with each other or not.

One solution is to define simple knowledge rules not hard-coded in the processing software but to provide them as parameters to a program that applies the rules to data and monitors their effects. This way, the impact of each rule can be kept track of and evaluated. The implementation of such an approach is necessarily a trade-off between flexibility and maintainability. In the most flexible case, a user can insert arbitrary code to alter the data and the system only keeps track of the changes. This means that maintaining such a rule set is as complex as maintaining any software system. In particular, it will be very difficult to discover contradictions or redundancies in such code, and one needs to resort to software testing techniques to guarantee any level of quality. On the more restricted side, one can restrict the type of rules that may be specified, allowing for some (automated) ways of interpreting their properties before confronting them with the data.

Two interesting properties of such rules present themselves. First, note that the condition in rules of the type (9.1) detects an invalid situation. The first requirement we have on a modifying rule is therefore that it actually solves that particular invalidity. In other words, if we apply the same rule twice to a data set, no modification is done at the second round. Second, it would be nice if the rules do not interfere with each other. That is, if one such knowledge rule is applied, we do not wish it to modify the data in such a way that a second rule is necessary to repair the damage done. In general, we shall demand that the order of execution of such rules is not important. Both properties together greatly enhance maintainability and understandability of the rule sets.

In the following section, we will give a formal definition of modifying rules in terms of 'modifying functions'. This allows us to make the two demands on modifying functions more precise and to study the conditions under which they hold. We shall see that we can obtain necessary and sufficient condition for the first demand and only a sufficient condition for the second. After discussing these general conditions, they will be made explicit for an important class of rules on numerical data. In Section 9.2, we will show how modifying rules can be applied to data from R. Readers who are less interested in the theoretical aspects can skip Section 9.1 on first reading.

9.1.1 Modifying Functions

In the following discussion, it will be assumed that the data to be treated is a rectangular data set consisting of m records and n variables, which are not necessarily stored in the correct type. The measurement domain of each variable is denoted D, so a record of n variables is an element of D^n (this is a special case of the definition of the measurement domain in Section 6.3.1).

We interpret a modifying rule as a function $f : D^n \to D^n$. Both the input and the output are in the measurement domain, since we do not demand that a single modifying rule fixes every issue a record might have (although it is also not excluded). We will use the term *modifying function* (to be made precise at the end of this section) to indicate a function based on a data modifying rule.

Before giving the formal definitions, let us look at an example. Consider the following rule:

$$\begin{array}{l}\texttt{if}\\ \textit{general expenditures} \text{ exceeds one half of } \textit{total expenditures}\\ \texttt{then}\\ \textit{total other expenditures} \text{ is set to missing.}\end{array} \qquad (9.2)$$

For compactness we will use the shorthands *e* for *general expenditures, t* for *total expenditures,* and *o* for other expenditures. The measurement domains for each variable is given by $\mathbb{R} \cup \mathsf{NA}$, so we have $D^n = \{\mathbb{R} \cup \mathsf{NA}\}^3$. A formal expression of the rule in terms of a function is simply the following:

$$f(e,t,o) = \texttt{if } e \in (\tfrac{t}{2},\infty) \wedge o \neq \mathsf{NA} \texttt{ then } (e,t,\mathsf{NA}) \texttt{ else } (e,t,o). \qquad (9.3)$$

Here, the notation $(\frac{t}{2},\infty)$ indicates the open range $\{x \in \mathbb{R} : \frac{t}{2} < x < \infty\}$. We also include explicitly the demand $o \neq \mathsf{NA}$ since we do not want to (re)set a value unnecessarily.

The form of function (9.3) is easy to generalize in the following formal definition.

Definition 9.1.1 (Modifying function) *Given a measurement domain D^n and a subset $T \subseteq D^n$. A* modifying function *f on D^n is a function that can be written in the form*

$$f(x) = \texttt{if } x \in T \texttt{ then } m(x) \texttt{ else } x,$$

where m is a function on D and for all $x \in T$ we have $m(x) \neq x$.

This definition includes the case where $T = D^n$, meaning that x is transformed to x' in any case. If T is a proper subset of D^n ($T \subset D^n$), then we can interpret T as an invalid region, and the rule is an attempt to move a record x out of this invalid region.

Idempotency

Consider again the modifying function of Eq. (9.3). It is easy to see that if we replace a record (e,t,o) with $f(e,t,o)$ a second application of f will not change the result. To see this, we need to check two cases: the cases where the initial record satisfies the condition and the cases where the condition is not satisfied. Starting with the latter case, suppose that for our original record we have $e \notin (\frac{t}{2},\infty)$. In that case, we have $f(e,t,o) = (e,t,o)$ and therefore $f(f(e,t,o)) = f(e,t,o) = (e,t,o)$. Now, if we have $e \in (\frac{t}{2},\infty)$ for the original record, we get (e,t,NA). Since we now have $o = \mathsf{NA}$, the result of $f(e,t,\mathsf{NA}) = (e,t,\mathsf{NA})$.

The point is that once a record is moved out of an invalid region, we do not want the function to move it again. The property that iterating a function twice does not alter the result is referred to as the *idempotent* property of a function, which we will now define formally.

Definition 9.1.2 (Idempotency) *Given a measurement domain D^n. A function $f : D^n \rightarrow D^n$ is called* idempotent *on a region $M \subseteq D^n$ if for every $\boldsymbol{x} \in M$ we have $f(f(\boldsymbol{x})) = f(\boldsymbol{x})$. A function that is idempotent on D^n is called* idempotent.

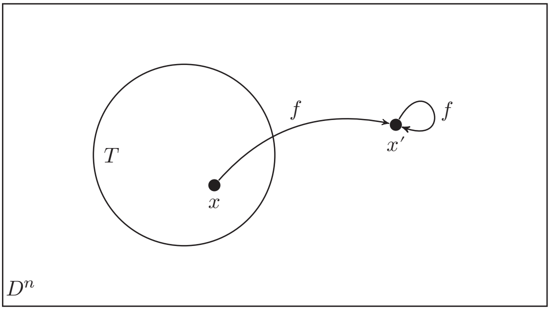


Figure 9.1 Idempotency of a modifying function. The full rectangle represents the measurement domain D^n. The portion in the rectangle outside of T is valid according to the condition in f. The region T is considered invalid. The function f maps an invalid record to a valid one and a valid record to itself.

The idempotent property is not only desirable from the theoretical or subject-matter points of view. The main practical benefit is that the function, or its underlying rule, can be interpreted just by understanding the definition of the condition and the function m without regarding possible iterations of the rule. It is therefore useful to have some general conditions under which a modifying function is idempotent. Intuitively, a modifying function should be idempotent when m takes a record x out of the invalid region, as depicted in Figure 9.1. The following proposition claims that this is indeed a necessary and sufficient condition.

Proposition 9.1.3 *A modifying function f is idempotent if and only if for every $x \in T$ we have $m(x) \in \overline{T}$.*

To show that f is indeed idempotent on whole D under these conditions we prove it separately for the complement of T in D (denoted $\overline{T}$) and for T.

Proof: For $x \in \overline{T}, f$ is the identity function so f is idempotent on $\overline{T}$. Given an $x \in T$. If $m(x) = x' \in \overline{T}$ then by Definition 9.1.1 we have $f(f(x)) = f(x') = x' = f(x)$, showing that m mapping x out of T is a sufficient condition. To show that it is also necessary, assume that $m(x) \in T$. Then again by definition we have $f(f(x)) = m(m(x))$. This is not equal to $m(x)$ (and thus not equal to $f(x)$) since definition of f requires that m is not idempotent on T. □

It is not difficult to define modifying functions that are not idempotent. Exercise 9.1.2 is aimed to give an example of how such situations may occur.

Commutativity

Consider an example rule that can be used to detect and repair unit-of-measure errors where units are off by a factor of 1000. We use the same variables as in Eq. (9.3).

```
if
    general expenditures exceeds 300 × total expenditures
then                                                        (9.4)
    divide general expenditures by 1000
```

The formal definition can be written as follows:

$$g(e,t,o) = \texttt{if } e \in (300t, \infty) \texttt{ then } (e/1000, t, o) \texttt{ else } (e,t,o). \tag{9.5}$$

The result of applying both f and g to a record (e, t, o) may well be order-dependent. Consider a record for which $e \in (300t, \infty)$. Then for sure it must be in $(\frac{t}{2}, \infty)$. Now, if we apply first g and then f we get $f(g(e, t, o)) = f(e/1000, t, 0))$. If $e/1000 \notin (\frac{t}{2}, \infty)$ then $f(e/1000, t, o)) = (e/1000, t, o)$. Now, suppose we first execute f and then g. We have $g(f(e, t, o)) = g(e, t, \mathsf{NA}) = (e/1000, t, \mathsf{NA})$.

So here we have a case where the order of execution of a set of modifying functions $\{f, g\}$ may be important for the result. For larger sets of modifying rules, such interdependencies make the action of any single rule on a data set hard to judge. In the worst case, one function may undo the action of another. When designing a set of rules, it is therefore desirable to detect these cases so the 'risk of collision' can be assessed or to avoided altogether.

The technical term for independence of order of execution is called *commutativity* of two functions, which are now ready to define precisely.

Definition 9.1.4 **(Commutativity)** *Given a measurement domain D^n. Two functions $f, g : D^n \to D^n$* commute on a region $M \subseteq D^n$ *when for each $x \in M$ we have $f(g(x)) = g(f(x))$. When two functions commute on D^n we say that they* commute.

Two functions trivially commute when they act on different variables. If two modifying functions f and g act on different variables in the condition as well as in the consequent, the action of the first function cannot alter the outcome of the second. Apart from acting on different variables, we can derive some very general sufficient (but not necessary) conditions under which two modifying functions commute. These are stated in the following proposition.

Proposition 9.1.5 *Given two idempotent modifying functions f and g with associated invalid regions T and U respectively. The following conditions guarantee that f and g commute:*

1) $f(x) \in (\overline{T} \cap \overline{U})$ *for all* $x \in T$,
2) $g(x) \in (\overline{T} \cap \overline{U})$ *for all* $x \in U$,
3) $f(x) = g(x)$ *for all* $x \in T \cap U$.

Before stating the proof, it is elucidating to consider the situation depicted in Figure 9.2. Here, f and g are modifying functions that we define to be idempotent on all of D. On the right, we see that f and g commute on the point x: the dashed lines correspond to $g(f(x)) = x'$ and the solid arrows correspond to $f(g(x)) = x'$. On the left, we see that f and g do not commute on the point y, because f takes y into the invalid region of g (labeled U), after which g takes y' out of U. Again, following the solid arrows corresponds to $f(g(y))$ and following the dashed lines corresponds to $g(f(y))$.

The proof of the proposition now consists of carefully considering the four regions in the diagram: D^n less the valid regions, region T less U, region U less T, and the intersection of T and U.

Proof: First note that we can write as a disjunct union

$$D^n = (\overline{T} \cap \overline{U}) \cup (T \cap \overline{U}) \cup (\overline{T} \cap U) \cup (T \cap U),$$

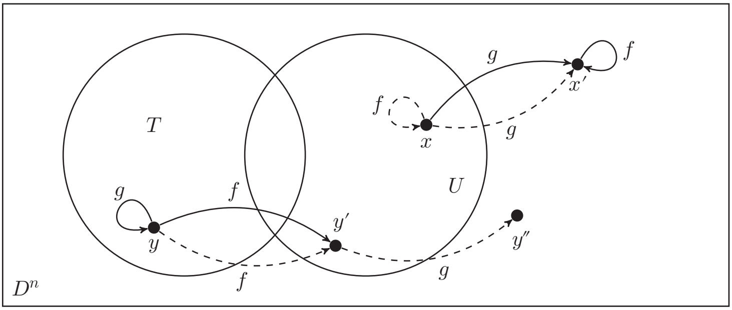


Figure 9.2 Commutativity of idempotent modifying functions. The full rectangle including subsets *T* and *U* corresponds to the measurement domain *D*. The idempotent modifying functions *f* and *g* commute on *x* but not on *y*.

where complements are taken with respect to D^n. First consider the region $\overline{T} \cap \overline{U}$. Since according to the definition of modifying functions (Definition 9.1.1) both f equals identity on $\overline{T}$ and g equals identity on $\overline{U}$, they commute on $\overline{T} \cap \overline{U}$ under no further conditions.

Now consider the region $T \cap U$. Since f is idempotent, it maps any $x \in T \cap U$ to a region outside of T. Suppose that it maps all points in $T \cap U$ to $\overline{T} \cap \overline{U}$. In that case, by idempotency of g we have $g(f(x)) = f(x)$ for all $x \in T \cap U$. Now suppose that g also maps all points in $T \cap U$ to $\overline{T} \cap \overline{U}$. By idempotency of f we have $f(g(x)) = g(x)$. So, when both f and g map all points in $T \cap U$ to points in $\overline{T} \cap \overline{U}$, they commute only when $f(x) = g(x)$. This proofs the sufficiency of condition 3 for the region $T \cap U$.

Now consider the region $T \cap \overline{U}$. Suppose that f maps every point in $T \cap \overline{U}$ to $\overline{T} \cap \overline{U}$. In that case, we have $g(f(x)) = f(x)$. By idempotency of g we have $f(g(x)) = f(x)$ so f and g commute. This shows the sufficiency of the first condition on the region $T \cap \overline{U}$. The sufficiency of the second condition on the region $\overline{T} \cap U$ follows from symmetry by replacing in the previous argument T with U and f with g.

Since we have now shown sufficiency of the conditions in the theorem for the four different disjunct regions that constitute D, we are done. □

Procedure 9.1.1 Commutativity of modifying functions

Input Two modifying function f and g.
Output A guarantee that they commute or an indication for further inspection.

1 If f and g pertain to different variables they commute.

2 If they pertain (partially) to the same variables, test for idempotency. There are three options.

- **2.a** If they are idempotent and the conditions of Proposition 9.1.5 hold, they commute.
- **2.b** If they are idempotent and the conditions of Proposition 9.1.5 do not hold, go to **3**.
- **2.c** If they are not idempotent, go to **3**.

3 Inspect the explicit form of the condition and the modifying functions to determine commutativity.

The above outline of a procedure combines the trivial notion that two functions commute when they act on different variables with the general conditions derived above. This procedure does not guarantee that mutually interfering rules can always be found automatically. It does provide a strategy that can be partially automated to indicate possible conflicts. In order to be able to explicitly test the conditions for idempotency or commutativity given in Propositions 9.1.3 and 9.1.5, we need to assume more about the type of rules. In the following section, these conditions are made explicit for an important class of direct rules.

9.1.2 A Class of Modifying Functions on Numerical Data

Many modification rules on numerical data that are used in practice can be written in the form

$$\begin{aligned} &f : \mathbb{R}^n \to \mathbb{R}^n,\\ &f(\boldsymbol{x}) = \texttt{if } \boldsymbol{Ax} < \boldsymbol{b} \texttt{ then } \boldsymbol{Cx} + \boldsymbol{d} \texttt{ else } \boldsymbol{x}. \end{aligned} \tag{9.6}$$

Here, the real matrices $\boldsymbol{A}$ and $\boldsymbol{C}$ are of dimensions $m \times n$ and $n \times n$, respectively, and we assume that the system $\boldsymbol{Ax} < \boldsymbol{b}$ is feasible. A transformation that sends a vector $\boldsymbol{x}$ to $\boldsymbol{Cx} + \boldsymbol{d}$ is called an *affine map*. Hence, we will call a modifying function as in Eq. (9.6) an *affine modifier with linear inequality conditions*, or AMLIC for short.

It was noted by Scholtus (2015) that affine maps cover several common operations that are performed either manually by analysts or via automated processing of modifying rules. In particular, we will show that such mappings can be used to replace values by a constant, multiply values by a constant, swap values in a record, and compute a value from a linear combination of other values in the record.

As an example, we will study three variables: *turnover*, *other revenue*, and *total revenue*. A record containing a value for each of these variables can be represented as a vector $\boldsymbol{x} = (x_1, x_2, x_3)$, where x_1 is the value of *turnover*, x_2 the value of *other revenue*, and x_3 the value of *total revenue*.

Example 9.1.6 (Replace a value by a constant) *The following modifying rule replaces a negative value.*

> `if` other revenue *is negative,* `then` *replace it by 1.*

The modifying function $\mathbf{C}\boldsymbol{x} + \mathbf{d}$ *now has the following form:*

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ 1 \\ x_3 \end{bmatrix}.$$

Note that the matrix multiplication sets x_2 to zero while the replacing value is stored in $\boldsymbol{d}$.

Example 9.1.7 (Multiply by a constant) *Unit of measure errors must be repaired by multiplication with a constant. The following rule is an attempt to such a repair.*

> *If* other revenue *is larger than 300 times* total revenue, *divide* other revenue *by 1000.*

The modifying function is specified by

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 10^{-3} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ 10^{-3}x_2 \\ x_3 \end{bmatrix}.$$

Here, the correction factor is provided in **C** *and there is no contribution from* ***d*****.**

Example 9.1.8 (Swapping values) *It may occur that values have been stored or submitted in the wrong field of a data file. If this occurs frequently, and in a recognizable way, a direct rule may be a quick solution to solve it.*

> *If* other revenue *is larger than* total revenue, *swap their values.*

The modifying corresponding modifying function is defined as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_3 \\ x_2 \end{bmatrix},$$

where **C** *is a permutation matrix that swaps the last two variables.*

Example 9.1.9 (Replace with a linear combination) *Suppose we have validation rules stating*

> turnover + other revenue = total revenue
> turnover ≥ 0,
> other revenue ≥ 0,
> total revenue ≥ 0.

We can attempt to solve imbalances by attributing differences to the variable other revenue, *using the following rule.*

> *If all variables are nonnegative and* total revenue *minus* turnover *is less than* other revenue, *replace* other revenue *with the difference between* total revenue *and* turnover.

The modifying function of this rule is described as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_3 - x_1 \\ x_3 \end{bmatrix}.$$

There are other examples, which will be discussed in Exercise 9.1.5.

Given that several such common operations can be expressed in the form of Eq. (9.6), it is interesting to see if general and explicit conditions of idempotency and commutativity can be derived. As we will be shown next, this is indeed the case.

Idempotency

According to Proposition 9.1.3, we need to check if every vector $\boldsymbol{x}$ in $T = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{Ax} < \boldsymbol{b}\}$ is mapped to a vector that satisfies the opposite. That is, we need to ensure that any vector $\boldsymbol{x}' = \boldsymbol{Cx} + \boldsymbol{d}$ $(\boldsymbol{x} \in T)$ does *not* satisfy $\boldsymbol{Ax}' < \boldsymbol{b}$, that is, we demand that

$$\boldsymbol{ACx} + \boldsymbol{Ad} \geq \boldsymbol{b}$$

for all $\boldsymbol{x}$ satisfying $\boldsymbol{Ax} < \boldsymbol{b}$. These demands are summarized by stating that the system

$$\begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{AC} \end{bmatrix} \boldsymbol{x} < \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{b} - \boldsymbol{Ad} \end{bmatrix} \tag{9.7}$$

has no solutions. The consistency of such systems can be determined, for example, through Fourier–Motzkin elimination.

Commutativity

Given two modifying functions f and g of the class defined by Eq. (9.6). Also define T and U to be their respective invalid regions, defined by $T = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{Ax} < \boldsymbol{b}\}$ and $U = \{\boldsymbol{x} \in \mathbb{R}^n : \tilde{\boldsymbol{A}}\boldsymbol{x} < \tilde{\boldsymbol{b}}\}$. The modifying function of f is defined by $\boldsymbol{x} \mapsto \boldsymbol{Cx} + \boldsymbol{d}$ and the modifying function of g is defined by $\boldsymbol{x} \mapsto \tilde{\boldsymbol{C}}\boldsymbol{x} + \tilde{\boldsymbol{d}}$. Before stating Procedure 9.1.1 explicitly for this case, we derive sufficient conditions for the two functions to be commutative, in the case that they are idempotent.

If f and g are idempotent, we need to check the three conditions of Proposition 9.1.5. Conditions 1 and 2 are related to regions T and U, respectively, and they state that any vector in those regions must be mapped outside of $T \cup U$. Using a similar reasoning as for Eq. (9.7), we arrive at the conclusion that this is equivalent to stating that the systems of inequalities given by

$$\begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{AC} \\ \tilde{\boldsymbol{A}}\boldsymbol{C} \end{bmatrix} \boldsymbol{x} < \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{b} - \boldsymbol{Ad} \\ \tilde{\boldsymbol{b}} - \tilde{\boldsymbol{A}}\boldsymbol{d} \end{bmatrix} \text{ and } \begin{bmatrix} \tilde{\boldsymbol{A}} \\ \tilde{\boldsymbol{A}}\tilde{\boldsymbol{C}} \\ \boldsymbol{A}\tilde{\boldsymbol{C}} \end{bmatrix} \boldsymbol{x} < \begin{bmatrix} \tilde{\boldsymbol{b}} \\ \tilde{\boldsymbol{b}} - \tilde{\boldsymbol{A}}\tilde{\boldsymbol{d}} \\ \boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{d}} \end{bmatrix} \tag{9.8}$$

have empty solution spaces.

The third condition states that for any vector in $T \cap U$, the modifying functions must be equal so $\boldsymbol{Cx} + \boldsymbol{d} = \tilde{\boldsymbol{C}}\boldsymbol{x} + \tilde{\boldsymbol{d}}$ for all $\boldsymbol{x} \in T \cap U$. Let us denote with W the region where this equality holds, so $W = \{\boldsymbol{x} \in \mathbb{R}^n : (\boldsymbol{C} - \tilde{\boldsymbol{C}})\boldsymbol{x} = \tilde{\boldsymbol{d}} - \boldsymbol{d}\}$. The condition implies that every $\boldsymbol{x} \in T \cap U$ is also in W, or $T \cap U \subseteq W$. To express this in terms of systems of linear equations, we can restate this by demanding that $T \cap U \cap \overline{W} = \emptyset$. Now, $\overline{W}$ is a set of vectors that is defined by *not* satisfying a set of linear equalities. A vector is a member of this set when it violates at least one of those restrictions. Now, let us denote with $\boldsymbol{w}_i^T$ the ith row of the matrix $\boldsymbol{C} - \tilde{\boldsymbol{C}}$. The ith restriction defining W is given by

$$\boldsymbol{w}_i^T\boldsymbol{x} = \tilde{d}_i - d_i.$$

For a vector to violate this restriction, it should satisfy either

$$\boldsymbol{w}_i^T\boldsymbol{x} < \tilde{d}_i - d_i,$$

or

$$\boldsymbol{w}_i^T\boldsymbol{x} > \tilde{d}_i - d_i.$$

We are now able to state the third condition in terms of demands on sets of linear restrictions. The set $T \cap U \cap \overline{W}$ is empty if for at least one of the $i = 1, 2, \ldots, n$ one of the two sets

$$\begin{bmatrix} \boldsymbol{A} \\ \tilde{\boldsymbol{A}} \\ \boldsymbol{w}_i^T \end{bmatrix} \boldsymbol{x} < \begin{bmatrix} \boldsymbol{b} \\ \tilde{\boldsymbol{b}} \\ \tilde{d}_i - d_i \end{bmatrix} \text{ or } \begin{bmatrix} \boldsymbol{A} \\ \tilde{\boldsymbol{A}} \\ -\boldsymbol{w}_i^T \end{bmatrix} \boldsymbol{x} < \begin{bmatrix} \boldsymbol{b} \\ \tilde{\boldsymbol{b}} \\ d_i - \tilde{d}_i \end{bmatrix} \tag{9.9}$$

is infeasible.

Now that we have derived explicit conditions for the commutativity of two modifiers we can specialize Procedure 9.1.1 to affine modifiers with linear inequality constraints. The above procedure is stated in a fairly general way, and may in principle involve many feasibility checks on systems of linear inequations. In practice, user-defined rules commonly involve only one or a few linear inequality conditions and only very simple transformations that involve few variables. So in many cases this procedure will be computationally tractable.

Procedure 9.1.2 Commutativity of AMLIC

Input Two affine modifiers with linear inequality conditions.
Output A guarantee that they commute or an indication for further inspection.
1 If the two functions act on two different variables they commute.
2 Determine for both functions whether they are idempotent by checking the feasibility of Equations (9.7).
  2.a If they are idempotent, sufficient conditions for commutativity are
    1. the systems of equations in (9.8) are both infeasible, and
    2. for at least one $i \in \{1, 2, \ldots, n\}$, one of the systems in Equation (9.9) is infeasible.
  2.b If they are idempotent but not commutative, go to **3**.
  2.c If they are not idempotent, go to **3**.
3 Inspect the explicit form of the condition and the modifying function to determine commutativity.

Exercises for Section 9.1

Exercise 9.1.1 *Consider the following modifying function on the domain $D^2 = \mathbb{R}^2$.*

$$f(\boldsymbol{x}) = \texttt{if } \boldsymbol{x} > 1 \texttt{ then } 0 \texttt{ else } \boldsymbol{x}.$$

What is the invalid region T? Use proposition 9.1.3 to check if this function is idempotent.

Exercise 9.1.2 *Show, by giving an example record that the modifying function of Eq.* (9.5) *is not idempotent.*

Exercise 9.1.3 *Consider the following modifying functions:*

$$\begin{aligned} &f : \mathbb{R}^2 \to \mathbb{R}^2 \\ &(x, y) \mapsto \texttt{if } x > 1 \texttt{ then } (0, y) \texttt{ else } (x, y) \\ &g : \mathbb{R}^2 \to \mathbb{R}^2 \\ &(x, y) \mapsto \texttt{if } y > 1 \texttt{ then } (x, 0) \texttt{ else } (x, y). \end{aligned}$$

Do f and g commute? Demonstrate this by drawing an x–y plane, shading the invalid regions for f and g. Depict the action of f and g on points in the plane for each of four relevant regions (these are $\overline{T}\cap\overline{U},\overline{T}\cap U$, $T\cap\overline{U}$, $T\cap U$, where T and U are the invalid regions for f and g).

Exercise 9.1.4 *The same as in Exercise 9.1.3, but now for the following functions:*

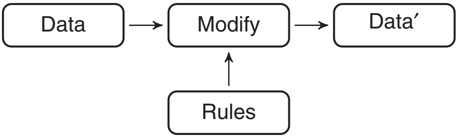
$$\begin{aligned}
&f:\mathbb{R}^2\to\mathbb{R}^2\\
&(x,y)\mapsto \texttt{if}\ x>1\ \texttt{then}\ (x,0)\ \texttt{else}\ (x,y)\\
&g:\mathbb{R}^2\to\mathbb{R}^2\\
&(x,y)\mapsto \texttt{if}\ y>1\ \texttt{then}\ (0,y)\ \texttt{else}\ (x,y).
\end{aligned}$$

Exercise 9.1.5 *Write the modifier of the following rules in matrix notation, as in Examples 9.1.6–9.1.9.*

a) *If* total revenue *is less than zero, replace it with* – total revenue.
b) *If* other revenue *is larger than* total revenue, *transfer 1000 from* total revenue *to* other revenue.

9.2 Rule-Based Correction with `dcmodify`

The package `dcmodify` implements a work flow where data modifying rules are not part of data processing software, but instead are stored and manipulated as parameters to a data cleaning workflow. The situation is depicted schematically as follows:



Here, a set of rules defines how data is modified. The allowed rules are assignments that executed possibly within a (nested) conditional statement. Just like the `validate` package, `dcmodify` employs a subset of the R language for rule definition.

The simplest command-line use is implemented as the `modify_so` function, accepting a `data.frame` and one or more (conditional) modifying rules.

```
library(dcmodify)
dat <- data.frame(x = c(5,17), y = c("a","b"),stringsAsFactors=FALSE)
modify_so(dat, if( x > 6 ) y <- "c")
##    x y
## 1  5 a
## 2 17 c
```

The function is set up so it can be integrated in a stream-like syntax facilitated by the `magrittr` package.

```
library(magrittr)
dat %>% modify_so(
    if (x > 6) y <- "c"
```

```
  , if (y == "c") x <- 11 )
 ##    x y
 ## 1  5 a
 ## 2 11 c
```

We point out two observations. First, the conditional statements are interpreted as if they are executed record by record. Under the hood, optimization takes place to make sure that code is executed in vectorized manner. Second, the rules are executed sequentially. In the example, the first rule changes the value of `y` from `"b"` to `"c"`. Next, the second rule changes the value of `x` from `6` to `11` in the second record because the value of `y` is now `"c"`. (Needless to say, these two rules do not commute in the sense explained in the previous section). This is in fact default behavior that can be toggled by setting the `sequential` option to `TRUE`.

9.2.1 Reading Rules from File

The `dcmodify` package follows the structure of the `validate` package. Like in `validate`, there is an object storing rules, the difference being that rules are stored in an object called `modifier` and not `validator`, reflecting the purpose of the rules. To execute the rules, there is a function called `modify` that 'confronts' a data set with a set of modifying rules.

```
 mod <- modifier(
   if ( x > 6 ) y <- "c"
  , if ( y == "c" ) x <- 11
)
 modify(dat, mod)
 ##    x y
 ## 1  5 a
 ## 2 11 c
```

Modifying rules may be read from and stored to files just like validation rules, and the structure of the files is the same as discussed in Section 6.5.6, except that the rule syntax is different. So rule may be defined in a free-format text file such as the following:

```
# modifiers.txt

if ( x > 6 ){
  y <- "c"
}

if ( y == "c" ){
  x <- 11
}
```

Subsequently, rules may be read with `modifier` using the `.file` argument.

```
mod <- modifier(.file="modifiers.txt")
```

Moreover, `modifier` objects support the same metadata as `validator` objects, so functions such as `origin`, `label`, and `variables` also work on modifier objects.

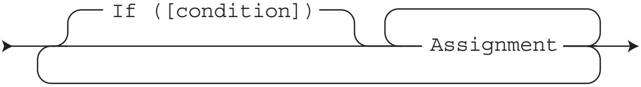
```
variables(mod)
## [1] "x" "y"
```

Metadata can be defined using the `yaml` format described earlier in Section 6.5.6 as well. So, for example, the following file defines a bit of metadata for one of the rules defined earlier:

```
rule:
-
  expr: |
    if ( x > 6 ) y <- "c"
  name: fix_x
  description |
    When ‘x' is too large, the y-category must be 'c'.
```

9.2.2 Modifying Rule Syntax

The general structure of a modifying statement allowed currently by the `dcmodify` package follows the following very simple syntax diagram:



Curly braces (blocks) have been left out for clarity. In short, a modifying statement consists of one or more assignments, possibly executed under a (nested) conditional statement.

The following examples illustrate the type of rules that can be defined with this syntax:

```
# multiple assignment
if ( x > 0 ){
  x <- 0
  y <- 2 * y
}

# nested conditions
if ( x > 0 ){
  x <- 0
  if ( y < 10 ){
    y <- 2*y
  }
}

# reassignment
x <- y - 7
```

Like in the `validate` package, it is possible to define variable macros (using the `:=` operator) or variable groups (with `vargroup`). In other words, the modifier defined by

```
  modifier(
    x_limit := 10*median(x)
  , if ( x > x_limit) x <- NA
 )
```

is equivalent to

```
  modifier( if ( x > 10*median(x) ) x <- NA )
```

9.2.3 Missing Values

It may occur that the condition in a modifying rule cannot be evaluated because one or more of the necessary values are missing. By default, the package then assumes that the result is `FALSE`. For example, compare the output for the two-record data set in the following code:

```
  dat <- data.frame(x=c(NA, 0), y=c(0,0))
  m <- modifier( if ( x == 0 ) y <- 1 )
  modify(dat, m)
  ##     x y
  ## 1 NA 0
  ## 2  0 1
```

In the first record, the condition `x == 0` could not be evaluated so the result `FALSE` was assumed and consequently, `y` is left untouched. The second record could be evaluated normally.

The default assumption that an `NA` in the condition is interpreted as `FALSE` can be altered with the option `na.condition`.

```
  modify(dat, m, na.condition=TRUE)
  ##     x y
  ## 1 NA 1
  ## 2  0 1
```

9.2.4 Sequential and Sequence-Independent Execution

In principle, execution of one rule can influence the condition of another rule (see also the discussion of Section 9.1.1). As an example, consider the following single-row data set and sequence of modifications:

```
  dat <- data.frame(x = 0, y = 0)
  m <- modifier(
    if ( x == 0 ) x <- 1
  , if ( x == 0 ) y <- 1
 )
```

If we apply the modifier to the data, the first modifying rule will set `x` to `1`, since `x` equals zero. The consequent of the second rule is not executed since now `x` has changed.

It may occur that the decision to execute the second statement should not depend on previously executed rules. This can be controlled by setting the option `sequential=FALSE`. As an example, consider the following outputs:

```
  modify(dat, m)
  ##   x y
```

```
## 1 1 0
modify(dat, m, sequential=FALSE)
##   x y
## 1 1 1
```

In the second case, the conditional part of a rule is always evaluated using the original data. So in the second case, the condition `x == 0` always evaluated to `TRUE`, regardless of the changes made by the first rule.

9.2.5 Options Settings Management

The option settings for the `dcmodify` package are handled by the same options manager as for the `validate` package. This means (again) that options can be set on a global level using `validate::voptions`.

```
voptions(na.condition=TRUE)
```

It is also possible to attach options to objects carrying rules, including `modifier` objects.

```
voptions(m, sequential=FALSE)
```

The options hierarchy, where global options (set with `voptions`) are overruled by options attached to an object, which are again overruled by options passed during a function call are the same as for the `validate` package as well.

9.3 Deductive Correction

In some cases, inconsistencies in a data record can be traced back to a single origin, either by (automatic) reasoning based on the validation rules or by allowing for a weak assumption based on subject matter knowledge. In general, deductive correction methods allow one to use data available within a faulty record to find a unique correction that fixes one or more inconsistencies.

For example, suppose we have a numerical record with integer values ($p = 120$, $c = 23, t = 134$), subject to the rule $p + c = t$ (*p*rofit plus *c*ost equals *t*urnover). Without any further information, Fellegi and Holt's principle tells us that one randomly chosen variable should be altered. However, if we study the values a little, we see that the record can be made consistent with the rule if we swap the last two digits in the value for turnover: 134 → 143, assuming that a typo has been made.

Over the years, several algorithms have been developed and implemented that allow various forms of deductive data correction. Such algorithms are often specific to a certain data type, a certain type of validation rules, and certain assumptions about the cause of error. In the following section, we highlight a number of those algorithms and show how they can be applied in R.

9.3.1 Correcting Typing Errors in Numeric Data

Integer data that is entered by humans may contain typing errors that cause violations of linear equality restrictions. If such errors can be detected, they may be repaired by replacing the erroneous value with a valid one. Scholtus (2009) describes a method for detecting and repairing typing errors in numerical data under linear restrictions. The basic idea is both elegant and simple; given a record violating one or more restrictions, one generates a number of candidate solutions by keeping one or more variables fixed and solving for the other ones using the linear restrictions. If the values in the candidate solution are not more than, say, a single typing error away from the original value, it is assumed that a typing error has been made. Finally, from all the candidate solutions, a trade-off between the minimal number of changes with the maximum number of resolved violations determines what changes are actually applied.

Area of Application

This method can be applied to records containing integer data, subject to at least set of linear equality restrictions that can be written as

$$\boldsymbol{Ax} = \boldsymbol{b}, \tag{9.10}$$

where $\boldsymbol{A}$, and $\boldsymbol{b}$ have integer coefficients. Such a subset is necessary to generate candidate solutions. There may be more (in)equality restrictions, but these are only used to check that proposed solutions do not generate extra violations. The demand that $\boldsymbol{x}$ contains only integer values poses a restriction on the constraint matrix. In particular, $\boldsymbol{A}$ must be a totally unimodular matrix for Eq. (9.10) to have any integer solutions. A matrix is *totally unimodular* when every nonsingular square submatrix has determinant ±1, which again implies that each coefficient $A_{ij} \in \{-1, 0, 1\}$. There are several tests for unimodularity available, such as those described by Heller and Tompkins (1956) or Raghavachari (1976), and the `lintools` R package implements a combination of them.

Although a general classification of unimodular matrices seems to be missing, the authors have never encountered a practical set of restrictions that did not satisfy the unimodularity criterion. The reason is that many restriction sets are balance restrictions of the form $\boldsymbol{Ax} = \boldsymbol{0}$ that follow a tree-like structure. In such systems, there is a single overall result, for example, *total profit* = *total revenue* − *total cost*, and each term on the right-hand side is the sum of new subvariables (a list of particular costs, a list of particular revenues, and so on). Each subvariable can be the sum of new variables again. It is not hard to see that for this important class of tree-like balance structures, an integer solution can always be constructed by choosing an integer for the top result variable, and recursively choosing values down the tree, so in these cases the corresponding restriction matrix must be totally unimodular.

The Algorithm

In his original work, Scholtus identifies typographical errors by studying their effect on the decimal expansion of integers. Here, we generalize the method a little by treating an integer as a string and allowing for any typographical error based on an edit-based string metric as described in Section 5.4.1. This has the advantage that sign errors can be detected and repaired as well. Moreover, in the original formulation, the set of equality restrictions was limited to the form $\boldsymbol{Ax} = \boldsymbol{0}$, whereas we treat the case of Eq. (9.10). The approach described as follows was first described in van der Loo *et al.* (2011).

The algorithm consists of two main parts. First, a list of candidate solutions is generated. Next, a subset of candidates that is in some sense optimal is selected from the set of candidates. The following procedure gives an overview of the algorithm.

Procedure 9.3.1 Correct typographical errors in restricted numerical data

```
Input A record $\boldsymbol{x} \in \mathbb{Z}^n$ and a matrix $\boldsymbol{A}$, and vector $\boldsymbol{b}$ expressing a set of linear equality
    restrictions. A distance function $d$ computing an edit-based distance and a maximum
    allowed distance $m$.
0 $L \leftarrow \varnothing$.
1 Determine the index set $J \in \{1, 2, \ldots, n\}$ selecting variables occurring in violated
    restrictions, but not in satisfied restrictions.
2 For each $j \in J$
    2.a Determine the matrix $\boldsymbol{A}^{(j)}$ containing the violated restrictions in which $x_j$ occurs
        and the corresponding vector $\boldsymbol{b}^{(j)}$.
    2.b For each row $i$ of $\boldsymbol{A}^{(j)}$ compute $\hat{x}_j^{(i)} = \left(b_i^{(j)} - \sum_{j' \neq j} A_{ij'}^{(j)} x_{j'}\right) / A_{ij}^{(j)}$
    2.c If the distance $d(\hat{x}_j, x_j) \leq m$ add $\hat{x}_j$ to $L$.
3 Choose the subset $S \subseteq L$ that maximizes the number of satisfied restrictions without
    creating new violations.
Output A set $S$ of optimal solutions.
```

Here, in step **1** attention is restricted to variables that occur only in violated restrictions since altering a variable that also occurs in a satisfied equality restriction would lead to new violations. To explain the second step, first note that given a violated restriction of the form $\sum_j A_{ij} x_j = b_i$, one can always find a solution that resolves the violation choosing a variable and solving it from the restriction. In step **2**, this is done for each variable occurring in each violated restriction. Each such solution is considered a candidate if it differs no more than a certain string distance from the original value. The set of candidates L may therefore in general contain multiple suggestions for each variable. In step **3**, the set of candidate solutions is reduced by selecting the subset of candidates that (1) contains only one suggestion for each chosen variable, (2) maximizes the set of satisfied restrictions, and (3) does not violate any extra restrictions that were not violated before. This maximization be performed by generating all subsets without duplicate suggestions and computing the violations after substituting the suggestions in the original record $\boldsymbol{x}$.

There are some extensions of this algorithm that make it more useful in practice. The first extension is to allow for linear (in)equations. The subset selection method in the final step can take these into account by checking that the chosen solution does not violate any new inequality restrictions. The second extension allows for missing values by eliminating missing values from the set of (in)equations before running the algorithm just described.

Example 9.3.10 (after Scholtus (2009)) *Consider a record* $\boldsymbol{x} = (123, 192, 252)$ *subject to the restriction* $x_1 + x_2 = x_3$. *This restriction is not satisfied as* $123 + 192 = 315 \neq 252$. *We have* $\boldsymbol{A} = [1, 1, -1]$ *and* $\boldsymbol{b} = [0]$. *Here, all variables occur in the violated restriction and the candidates for correcting this record are (see step* **2** *of Procedure 9.3.1)*

$$\hat{x}_1^{(1)} = 0 - \frac{-1 \times 192 + 1 \times 205}{-1} = 60,$$

$$\hat{x}_2^{(1)} = 0 - \frac{-1 \times 123 + 1 \times 252}{-1} = 129,$$

$$\hat{x}_3^{(1)} = 0 - \frac{-1 \times 123 - 1 \times 192}{1} = 315.$$

Treating the numbers as strings, we compute the optimal string alignment distance d_{osa} between the candidates and their original values. This gives

$$d_{\text{osa}}\left(x_1^{(1)}, x_1\right) = d_{\text{osa}}(60, 123) = 3 \text{ (two substitutions, one insertion)},$$

$$d_{\text{osa}}\left((x_2^{(1)}, x_1\right) = d_{\text{osa}}(129, 192) = 1 \text{ (one transposition)},$$

$$d_{\text{osa}}\left((x_3^{(1)}, x_1\right) = d_{\text{osa}}(315, 252) = 3 \text{ (three substitutions)}.$$

Since the second candidate is only a single typographic error away from its original, we replace $\boldsymbol{x}$ with (123, 129, 252).

Correcting Typographical Errors in Numerical Data with R

The package `deductive` implements the algorithm for correcting typographical errors described above. The restrictions may be defined using the `validate` package. Example 9.3.10 can be computed in R as follows:

```
library(validate)
library(deductive)
dat <- data.frame(x1 = 123, x2 = 192, x3 = 252)
v <- validate::validator(x1 + x2 == x3)
deductive::correct_typos(dat, v)
##    x1  x2  x3
## 1 123 129 252
```

The `correct_typos` function accepts a data set and a rule set and returns a data set with corrections where possible. By default the maximum accepted edit distance (measured as optimal string alignment, see Section 5.4.1) equals one. The distance measure may be set by passing `method="[methodId]"` where `[methodId]` is any method accepted by `stringdist::stringdist` (see Section 5.4.2).

Here is an example with a sign error, which may be detected as a typographical error.

```
dat <- data.frame(x1 = -123, x2 = 129, x3 = 252)
deductive::correct_typos(dat,v)
##    x1  x2  x3
## 1 123 129 252
```

The `deductive` package extends Procedure 9.3.1 by also rejecting solutions that would cause new violations of restrictions that are not used in deriving candidate solutions.

```
dat <- data.frame(x1 = -123, x2 = 129, x3 = 252)
v <- validate::validator(x1 + x2 == x3, x1 < 0)
deductive::correct_typos(dat, v)
##     x1  x2  x3
## 1 -123 129 252
```

Here, no correction is applied to `x1` since switching it from negative to positive would break the rule `x1 < 0`.

Exercises for Section 9.3.1

Exercise 9.3.6 *Load the* `retailers` *data set from the* `validate` *package.*

```
data(retailers)
```

Use the validate package to set up a number of validation rules, for example.

$$\begin{aligned} turnover + other.rev &= total.rev \\ turnover &\geq 0 \\ other.rev &\geq 0 \\ total.rev &\geq 0. \end{aligned}$$

Apply `correct_typos` *to the data set and store it in an new variable called* `tcor`*. The old and new data set can be compared using the* `compare` *function of the* `validate` *package.*

```
compare(v, old=retailers, new=tcor)
```

where `v` *is a* `validator` *object. Study the output and read the help in* `?compare`*. Add more (in)equality restrictions to see if you can increase the number of typos found.*

9.3.2 Deductive Imputation Using Linear Restrictions

Correction of numerical records that are subject to a set of linear (in)equations is often a multistep process during which existing values are accepted as correct, missing values imputed or erroneous values removed. Once all erroneous values are removed, a record can be thought of as consisting of two parts: a part with correct values and a part missing. Since the now-present values are deemed correct, they may be substituted in the system of restrictions, yielding a reduced system that applies to the unobserved values. If sufficient values have been substituted, the reduced set of restrictions may restrict one or more unobserved values to a unique value.

One way of flushing out such implied equalities for a variable, say x, is by eliminating one by one all other variables until either no restriction pertaining to x is left, or an obvious unique value for x occurs. Repeating this procedure for each missing value yields every value that is uniquely implied by the reduced system of restrictions. For systems with inequalities, the elimination procedure involves Fourier–Motzkin elimination that quickly becomes intractable as the number of variables grows; the number of implied inequations maximally grows exponentially with the number of eliminated variables.

As an alternative, one may attempt to derive as many implied values as possible using other, more heuristic methods. These methods do not necessarily find every unique imputation, but they have significantly lower complexity than Fourier–Motzkin-based methods. Once the heuristic methods have been applied, a (possibly) smaller set of missing values remain, leaving a simpler problem for the elimination-based methods.

Below, we describe three such methods. The first one is based on computing the Moore–Penrose pseudoinverse for the subset of equality restrictions, the second finds zero imputations in specific cases and the recursively third finds combined inequations that imply a simple equality.

Extracting Implied Values from the Subsystem of Equalities

For this method, we assume we have a numerical record $\boldsymbol{x} \in \mathbb{R}^n$ for which a number of values are missing. The values may have been missing from to begin with or perhaps have been removed after application of a suitable error localization method. We furthermore suppose that $\boldsymbol{x}$ must satisfy a set of linear restrictions denoted

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}. \tag{9.11}$$

It is possible that the record is subject to inequality restrictions as well, but these may be ignored here.

Sorting the values conveniently, we may write $\boldsymbol{x} = (\boldsymbol{x}_o, \boldsymbol{x}_m)$, where o stands for observed and m for missing. Likewise, the columns of the restriction matrix $\boldsymbol{A}$ can be sorted so that we can write

$$[\boldsymbol{A}_o, \boldsymbol{A}_m] \begin{pmatrix} \boldsymbol{x}_o \\ \boldsymbol{x}_m \end{pmatrix} = \boldsymbol{b}.$$

This may be rewritten to obtain a set of demands on the missing values, expressed in the observed values.

$$\boldsymbol{A}_m\boldsymbol{x}_m = \boldsymbol{b} - \boldsymbol{A}_o\boldsymbol{x}_o. \tag{9.12}$$

Note that the right-hand side of this equation is fully known since it is computed from the observed values.

We are now left with three cases. The first option is that the system $\boldsymbol{A}_m\boldsymbol{x}_m = \boldsymbol{b}^*$ has no solutions. In that case, either the system of Eq. (9.11) was inconsistent to begin with, or the values of $\boldsymbol{x}_o$ are such that the system $\boldsymbol{A}_m\boldsymbol{x}_m = \boldsymbol{b}*$ is rendered inconsistent. Note that this is never the case when the original system was consistent and an appropriate error localization algorithm was applied. The second option is that $\boldsymbol{A}_m$ is square and of full rank so it can be inverted. In that case, we can impute the missing values uniquely by setting $\boldsymbol{x}_m = \boldsymbol{A}_m^{-1}\boldsymbol{b}^*$. The third option is that there are an infinite number of solutions. It can be shown that every solution $\hat{\boldsymbol{x}}_m$ can be expressed as

$$\hat{\boldsymbol{x}}_m = \boldsymbol{A}_m^+(\boldsymbol{b} - \boldsymbol{A}_o\boldsymbol{x}_o) + (\boldsymbol{1} - \boldsymbol{A}_m^+\boldsymbol{A}_m)\boldsymbol{w}, \tag{9.13}$$

where $\boldsymbol{A}_m^+$ is the so-called Moore–Penrose pseudoinverse of $\boldsymbol{A}_m$ and $\boldsymbol{w}$ is any vector in $\mathbb{R}^{n_m}$ with n_m the number of missings. We refer the reader to the appendix of this section for a short discussion of the Moore–Penrose matrix and the notation for this solution.

We may use Eq. (9.13) for deductive imputation in the following way. Suppose that the jth row of the matrix $\boldsymbol{1} - \boldsymbol{A}_m^+\boldsymbol{A}_m$ is filled with zeros. Since the Moore–Penrose pseudoinverse is unique, the jth coefficient of $\hat{\boldsymbol{x}}_m$ is then uniquely determined by the jth element of the first term in Eq. (9.13).

$$(\hat{\boldsymbol{x}}_m)_j = (\boldsymbol{A}_m^+(\boldsymbol{b} - \boldsymbol{A}_o\boldsymbol{x}_o))_j. \tag{9.14}$$

Example 9.3.11 *Given a set of balance restrictions on records $\boldsymbol{x} \in \mathbb{R}^{10}$,*

$$\begin{aligned} x_1 + x_2 &= x_3, \\ x_4 + x_5 + x_6 &= x_1, \\ x_7 + x_8 &= x_2, \\ x_9 + x_{10} &= x_3, \end{aligned}$$

and a record with the values

$$\begin{array}{cccccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \\ 100 & NA & NA & 15 & NA & NA & 25 & 35 & NA & 5 \end{array}.$$

Separating the observed and missing parts, the explicit form of Eq. (9.12) *becomes*

$$\underbrace{\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix}}_{\boldsymbol{A}_m} \underbrace{\begin{bmatrix} x_2 \\ x_3 \\ x_5 \\ x_6 \\ x_9 \end{bmatrix}}_{\boldsymbol{x}_m} = \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{\boldsymbol{b}} - \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\boldsymbol{A}_o} \underbrace{\begin{bmatrix} 100 \\ 15 \\ 25 \\ 35 \\ 5 \end{bmatrix}}_{\boldsymbol{x}_o}.$$

The matrix becomes after computing the pseudoinverse, the second term of Eq. (9.13) *reads*

$$\mathbf{1} - \boldsymbol{A}_m^+ \boldsymbol{A}_m = \begin{array}{l} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} \leftarrow x_2 \\ \leftarrow x_3 \\ \\ \\ \leftarrow x_9 \end{array},$$

where we marked the empty rows that correspond to variables x_2, x_3, and x_9. The values of these variables can be computed from the first term of Eq. (9.13).

$$\underbrace{\begin{bmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & -1 & 1 \end{bmatrix}}_{\boldsymbol{A}_m^+} \left(\underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{\boldsymbol{b}} - \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\boldsymbol{A}_o} \underbrace{\begin{bmatrix} 100 \\ 15 \\ 25 \\ 35 \\ 5 \end{bmatrix}}_{\boldsymbol{x}_o} \right) = \begin{bmatrix} 60 \\ 160 \\ 42.5 \\ 42.5 \\ 155 \end{bmatrix} \begin{array}{l} \leftarrow \hat{x}_2 \\ \leftarrow \hat{x}_3 \\ \\ \\ \leftarrow \hat{x}_9 \end{array}.$$

So the imputed record reads

$$\begin{array}{cccccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \\ 100 & 60 & 160 & 15 & NA & NA & 25 & 35 & 155 & 5 \end{array}.$$

Using Nonnegativity Constraints

Under some specific circumstances, nonnegativity constraints combined with a linear equality restriction can lead to unique solutions. For example, suppose we have the following balance rule:

$$x_1 + x_2 + x_3 + x_4 = x_5, \tag{9.15}$$

where all $x_i \geq 0$. Now if x_1, x_2, and x_5 are observed with values such that $x_1 + x_2 = x_5$, and x_3 and x_4 are missing, the unique possible values for x_3 and x_4 equals zero (assuming that the observed values are indeed correct). Such cases occur for instance, where only nonzero values have been submitted to a survey or administrative system.

To generalize this to a general linear equality restriction, we write

$$\boldsymbol{a} \cdot \boldsymbol{x} = \boldsymbol{a}_o \cdot \boldsymbol{x}_o + \boldsymbol{a}_m \cdot \boldsymbol{x}_m = b. \tag{9.16}$$

We may replace $\boldsymbol{x}_m$ with $\hat{\boldsymbol{x}}_m = \boldsymbol{0}$ if the following conditions are fulfilled (de Waal *et al.*, 2011):

- $\boldsymbol{a}_o \cdot \boldsymbol{x}_o = b$.
- $\boldsymbol{x}_m$ is restricted by $\boldsymbol{x}_m \geq \boldsymbol{0}$.
- All coefficients of $\boldsymbol{a}_m$ have the same sign.

Example 9.3.12 *Consider a record $\boldsymbol{x} = (x_1, x_2, x_3, x_4)$, subject to the constraints $x_1 + x_2 + x_3 = x_4$ and $x_i \geq 0$ for $i = 1, 2, 3, 4$. Given a record* (10, *NA*, *NA*, 10)*, the only possible imputation satisfying the restrictions is* (10, 0, 0, 10).

Deducing Values from Combined Inequalities

Given a record $\boldsymbol{x} = (\boldsymbol{x}_o, \boldsymbol{x}_m)$ subject to a set of rules $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$. Substituting the observed values $\boldsymbol{x}_o$ in the set of inequations, we obtain a reduced system

$$\boldsymbol{A}_m\boldsymbol{x}_m \leq \boldsymbol{b} - \boldsymbol{A}_o\boldsymbol{x}_o. \tag{9.17}$$

If the reduced system contains one or more pairs of inequations of the form

$$A_{ij}x_j \leq b_i,$$
$$A_{i'j}x_j \leq b_{i'},$$

where

$$\frac{b_i}{A_{ij}} + \frac{b_{i'}}{A_{i'j}} = 0,$$

then x_j may be substituted by b_i/A_{ij}. Since this method of imputation depends on substituting known values, the procedure is iterated by substituting imputed values into the system of inequations (9.17), detecting new pairs and imputing those until no new pairs are found.

Example 9.3.13 *Consider a two-valued record $\boldsymbol{x} = (x_1, x_2)$ subject to the constraints $0 \leq x_1 \leq x_2$. For a particular record* (*NA*, 0) *we may substitute x_2, which gives $-x_1 \leq 0$ and $x_1 \leq 0$, and hence $x_1 = 0$.*

Combining Methods

The three methods described above are not completely independent. For example, it is possible that after a record is partially imputed using Eq. (9.14), more zeros can be deductively imputed based on the nonnegativity constraints. These methods should therefore be applied in chain and iteratively until no further imputations can be found.

Deductive Imputation in R

The methods for deductive imputation mentioned above have been implemented in the `deductive` package. The function `impute_lr` (`lr` for linear restrictions) iteratively applies the heuristic methods, before computing the variable ranges of missing values by Fourier–Motzkin elimination. Here, we demonstrate Example 9.3.11.

```
library(validate)
library(deductive)
v <- validate::validator(
    x1 + x2      == x3
   , x4 + x5 + x6 == x1
   , x7 + x8      == x2
   , x9 + x10     == x3)
dat <- data.frame(
    x1 = 100, x2=NA_real_, x3=NA_real_, x4 = 15, x5 = NA_real_
  , x6 = NA_real_, x7 = 25, x8 = 35, x9 = NA_real_, x10 = 5)
dat
##    x1 x2 x3 x4 x5 x6 x7 x8 x9 x10
## 1 100 NA NA 15 NA NA 25 35 NA   5
impute_lr(dat,v)
##    x1 x2  x3 x4 x5 x6 x7 x8  x9 x10
## 1 100 60 160 15 NA NA 25 35 155   5
```

The function `impute_lr` accepts a data set and a number of rules in the form of a `validator` object. The result is a data set where fields that have a unique solution are imputed.

The following exercise shows how to apply `impute_lr` to a larger data set and how to interpret the results.

Exercises for Section 9.3.2

Exercise 9.3.7 *Set up a validator for the variables in the* `retailers` *data set (see Exercise 9.3.6 on page 224 for an example). Use* `impute_lr` *to apply deductive imputations, and store the result in a* `data.frame` *called* `imp`*. The function* `cells` *from the* validate *package summarizes the changes.*

```
cells(old = retailers, new = imp)
```

Appendix 9.A: Discussion of Equation (9.13)

Using some properties of the pseudoinverse, we will show that Eq. (9.13) is indeed a solution to the system of Eq. (9.12). Theory of the Moore–Penrose inverse is available in many textbooks, such as the book by Lipshutz and Lipson (2009) (where it is called the *generalized inverse*). The paper by Greville (1959) is a good early reference that discusses applications to linear systems and statistics.

The Moore–Penrose Pseudoinverse

The Moore–Penrose pseudoinverse (or just: pseudoinverse) is a generalization of the standard inverse of a square invertible matrix. Suppose that $\boldsymbol{A}$ is an invertible square matrix with singular value decomposition $\boldsymbol{U}\Lambda\boldsymbol{V}^T$, then its inverse is given by $\boldsymbol{A}^{-1} = \boldsymbol{V}\boldsymbol{\Lambda}^{-1}\boldsymbol{U}^T$ where Λ^{-1} is the matrix with the reciprocal eigenvalues of $\boldsymbol{A}$ on the diagonal. If $\boldsymbol{A}$ is not of full rank (this includes the case where it is not square), we can still decompose it using a singular value decomposition but since one or more eigenvalues are zero, $\boldsymbol{\Lambda}^{-1}$ is not defined. To compute the Moore–Penrose pseudoinverse, $\boldsymbol{\Lambda}^{-1}$ is replaced by $\boldsymbol{\Lambda}^{+}$, which is the matrix that has zero at the diagonal when the singular values are zero and the reciprocal singular value otherwise. We note that it is not necessary to use the svd theorem to define the Moore–Penrose pseudoinverse, but the explicit construction does make it easier to understand the generalization and some of its properties.

In particular, it is easy to demonstrate by svd decomposition that the pseudoinverse of a matrix $\boldsymbol{A}$ has the property

$$\boldsymbol{A}\boldsymbol{A}^{+}\boldsymbol{A} = \boldsymbol{A}. \tag{9.A.1}$$

This equation also shows that the matrix $\boldsymbol{I}_L \equiv \boldsymbol{A}\boldsymbol{A}^{+}$ is a *left identity* of $\boldsymbol{A}$. That is, it has the property that $\boldsymbol{I}_L\boldsymbol{A} = \boldsymbol{A}$. Using associativity of matrix multiplication and Eq. (9.A.1), one sees that $\boldsymbol{I}_L$ is idempotent: $\boldsymbol{I}_L^2 = \boldsymbol{I}_L$.

9.A.2 Equation (9.13) is a Solution

First recall that if $\boldsymbol{x}$ is a solution to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, than for any vector $\boldsymbol{v}$ for which $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{0}$, the vector $\boldsymbol{x} + \boldsymbol{v}$ is also a solution (we say that $\boldsymbol{v} \in \ker(\boldsymbol{A})$). Using Eq. (9.A.1), we see that the second term in Eq. (9.13) is in $\ker(\boldsymbol{A}_m)$:

$$\boldsymbol{A}_m(\boldsymbol{1} - \boldsymbol{A}_m^{+}\boldsymbol{A}_m)\boldsymbol{w} = \boldsymbol{0}.$$

So to show that Eq. (9.13) is a solution to the system of Eq. (9.12), we only need to show that the term

$$\boldsymbol{A}_m^{+}(\boldsymbol{b} - \boldsymbol{A}_o\boldsymbol{x}_o) \equiv \boldsymbol{A}_m^{+}\boldsymbol{b}^{*}$$

satisfies $\boldsymbol{A}_m\boldsymbol{A}_m^{+}\boldsymbol{b}^{*} = \boldsymbol{b}^{*}$ (we introduce $\boldsymbol{b}^{*}$ as a convenient shorthand for $\boldsymbol{b} - \boldsymbol{A}_o\boldsymbol{x}_o$). This can be done with the following calculation:

$$\begin{aligned}\boldsymbol{I}_L(\boldsymbol{A}_m\boldsymbol{A}_m^{+}\boldsymbol{b}^{*} - \boldsymbol{b}^{*}) &= \boldsymbol{I}_L\boldsymbol{A}_m\boldsymbol{A}_m^{+}\boldsymbol{b} - \boldsymbol{I}_L\boldsymbol{b}^{*}\\ &= \boldsymbol{I}_L^2\boldsymbol{b}^{*} - \boldsymbol{I}_L\boldsymbol{b}^{*} = \boldsymbol{0}.\end{aligned}$$

Here, we used that $\boldsymbol{I}_L$ spans a basis for the column space of $\boldsymbol{A}$.

10

Imputation and Adjustment

10.1 Missing Data

Missing values may occur in a dataset either because they were not measured or because they got removed somewhere during data processing, for example, because values were deemed erroneous and deleted. In either case, the term 'imputation' is used to indicate the practice of completing a dataset that contains empty values.

At first sight, imputation is not all that different from any predictive modeling task. One constructs a model of an incomplete dataset, trains it on a subset of reliable data, and predicts unobserved values. There are some elements that set imputation apart from 'general' predictive modeling, however. The first is that imputation tasks are often specifically done with the purpose of inference in mind. That is, one is usually less interested in the specific value for a single record than in the properties of the mean, variance, or covariance structure of the whole dataset or population. Secondly, and this is arguably related to the first issue, there are a number of methods that are commonly used in imputation but rarely in predictive modeling. Examples include nearest-neighbor imputation or expectation–maximization-based techniques.

The literature on imputation methodology is vast and extensive, and excellent text books and review papers are available [See, e.g., Anderson (2002); Andridge and Little (2010); Donders *et al.* (2006); Kalton and Kasprzyk (1986); Schafer and Graham (2002); Zhang (2003)]. The current chapter therefore summarizes some of the most common issues and imputation methodology and focuses on methods available in R.

10.1.1 Missing Data Mechanisms

One commonly distinguishes three missing data mechanisms [attributed to Rubin (1976), but see Little and Rubin (2002) or Schafer and Graham (2002)] that determine the basic probability structure of the missing value locations in a data record. To set up the mechanisms, consider three random variables: the variable of interest Y, an auxiliary variable X, and a missing value indicator R. Here, R is a binary variable that indicates whether a realization of Y is observed or not. One can imagine a dataset with realizations of these variables and model the joint probability $P(X, Y, R)$. Three models are distinguished, each of which we state in two equivalent representations as follows:

$$P_{\mathsf{MCAR}}(X, Y, R) = P(R)P(X, Y) = P(R)P(X)P(Y|X), \tag{10.1}$$

$$P_{\text{MAR}}(X, Y, R) = P(R|X)P(X, Y) = P(X, R)P(Y|X), \tag{10.2}$$

$$P_{\text{NMAR}}(X, Y, R) = P(R|X, Y)P(X, Y) = P(X, R)P(Y|X, R). \tag{10.3}$$

In the first model (MCAR: missing completely at random), it is assumed that the distribution of the missing value indicator is independent of both X and Y. If the dataset is created as a simple random sample, the missing values merely make that sample smaller, possibly with a different fraction for X and Y.

In the second model, labeled MAR (missing at random), it is assumed that the value of Y gives no information about whether it is observed or not. However, the distribution of R depends on the value of X (and Y's distribution could depend on the value of X). Gelman and Hill (2006) distinguish further between dependence of R on observed and unobserved variables X.

The last case, labeled NMAR (not missing at random), is not really a model at all. The two expressions on the right-hand side are simply rewrites of $P(X, Y, R)$ using the expression for conditional probabilities. In this case, the distribution of the missing value indicator may depend on both the values of X and Y.

Although these models give a clear classification of probability models for the missing/observed status of a variable, it is in practice not possible to distinguish between them based on observed data. Suppose that one constructs a dataset with X and Y independent (so $P(X, Y) = P(X)P(Y)$). Now, remove all values of Y above a certain threshold. Clearly, this is the not missing at random case, since the distribution of R depends on the value of Y. However, an analyst who has no access to the missing realizations of Y will not be able to detect the correlation between the values of Y and R. Indeed, it is impossible for the analyst to distinguish the NMAR situation from MCAR. Now suppose that X and Y are correlated so that larger values of Y co-occur with larger values of X. In that case, R will be correlated with X, and the analyst observes a MAR situation.

The approach that is commonly taken is rather practical. To accommodate for MCAR and MAR situations, many popular imputation methods simply attempt to leverage all the auxiliary information available (e.g., MICE, missForest, to be discussed later). To correct for the `NMAR` case, one needs to make assumptions about or somehow model the data collection process. To assess, in approximation, the effect of a possible `NMAR` missing data mechanism on outcomes, one may have to resort to sensitivity analyses using a simulation of the missing value mechanism.

10.1.2 Visualizing and Testing for Patterns in Missing Data Using R

Effective summarization of missing values across a multivariate dataset can provide incentives for investigating the missing value mechanism. Ideally (apart from having complete data), missing values are distributed as MCAR. If there are patterns in missing data pointing to a MAR situation, those patterns require an explanation or, when relevant and possible, the data collection process can be altered to prevent such patterns from occurring.

The `VIM` package (Templ *et al.*, 2012, 2016) offers a number of visualizations and aggregations for pattern discovery in missing data. As an example we will use the `retailers` dataset that comes with the `validate` package. The `VIM` function `aggr` aggregates missing value patterns per variable and per record.

```
data("retailers",package="validate")
VIM::aggr(retailers[3:9], sortComb=TRUE, sortVar=TRUE, only.miss=TRUE)
##
##  Variables sorted by number of missings:
##     Variable      Count
##    other.rev 0.60000000
##  staff.costs 0.16666667
##        staff 0.10000000
##  total.costs 0.08333333
##       profit 0.08333333
##     turnover 0.06666667
##    total.rev 0.03333333
```

The result is an overview of the fraction of missing values per variable. Here, the option to sort variables from high to low fractions of missing values is used (`sortVar=TRUE`). As a side effect, a plot of the aggregates, shown in Figure 10.1, is created. The

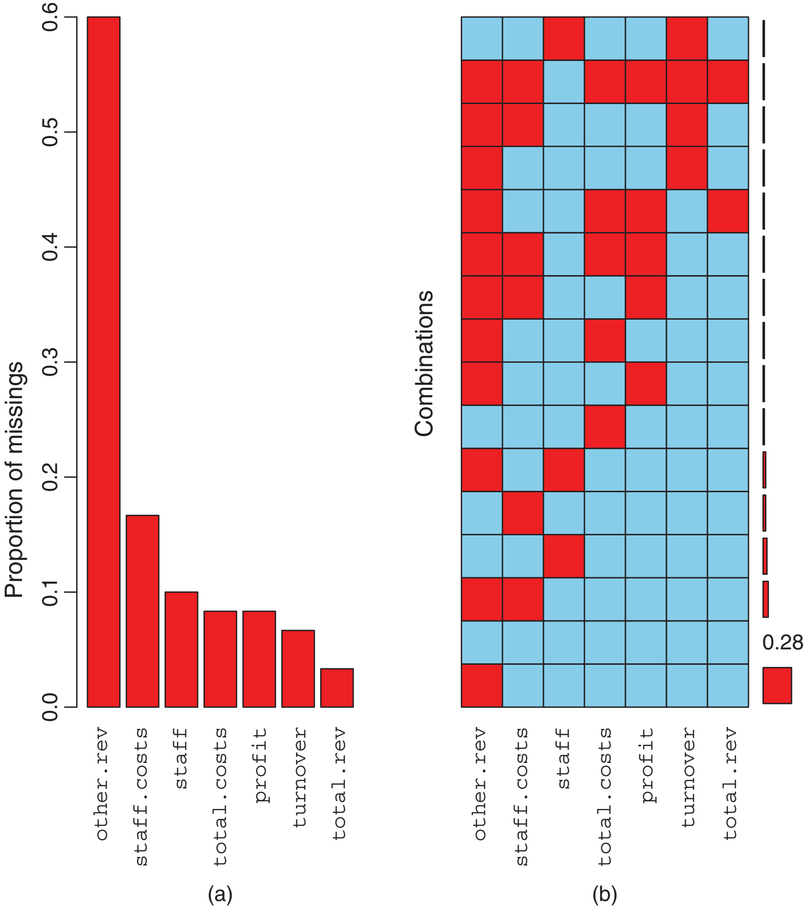


Figure 10.1 Percentages of missing values per variable (a) and occurrence of missing data patterns (b) in the `retailers` dataset, plotted with `VIM::aggr`.

visualization contains a barplot showing fractions of missing values per variable in panel (a) and a rectangular, space-filling plot indicating the occurrence of missing value combinations in panel (b). In the latter plot, each column in a space-filling grid of squares represents a variable, and each row represents a single occurring missing data pattern. A value that is missing is colored gray, and the observed values are colored light gray (by default). On the right there is a vertical bar chart that indicates for each row how often every pattern occurs. In this example, we also sort the graph by variables (decreasing left to right in fraction of missing values) and combination (`sortComb=TRUE`). Also, we specify that the bar chart heights should be relative to the number of patterns that contain at least one missing (`only.miss=TRUE`). The total fraction of complete records represented is printed at the right. The graph shows that the variable *other revenue* is missing most often, while the combination *other revenue* and *staff* is the most often missing combination in this dataset.

To detect whether a variable's missing value mechanism is MAR with respect to a second variable, the missingness indicator of the first variable can be used to split the dataset into two groups. The observed distributions of the second variable for the two groups can then be compared to distinguish between MCAR and MAR. In the case of MCAR, one expects the distributions to be similar. With `VIM::pbox`, one chooses a single numerical variable and compares its distribution with respect to the missingness indicator of all other variables. Here, we compare the distribution of *staff* against the status (missing or present) of other variables.

```
VIM::pbox(retailers[3:9], pos=1)
```

The result is shown in Figure 10.2. The leftmost boxplot shows the distribution of *staff*, with numbers indicating that there are 60 observations of which 6 are missing. The other boxplots, occurring in pairs, compare the distributions of *staff*, split according to the missingness of another variable. For example, the distribution of *staff* in the case where *other.rev* is observed (shown in light gray) appears to differ from the case where *other.rev* is missing. This indicates a possible MAR situation for *other.rev* with respect to *staff*. The widths of the boxplots indicate the number of observations used in producing the boxplot: a (very) thin boxplot indicates that the difference in distributions is supported by little evidence. The number of observations (top) and missing values (bottom) per group are printed below the boxes.

To confirm or reject our suspicion that the locations of the distributions differ significantly, we perform the Student *t*-test. Here, we use log-transformed data since economic data tends to follow highly skewed distribution (typically close to log-normal).

```
t.test(log(staff) ~ is.na(other.rev), data=retailers)
##
##   Welch Two Sample t-test
##
## data:  log(staff) by is.na(other.rev)
## t = 2.7464, df = 46.014, p-value = 0.008572
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.1985149 1.2880867
## sample estimates:
## mean in group FALSE  mean in group TRUE
##            2.329996            1.586695
```

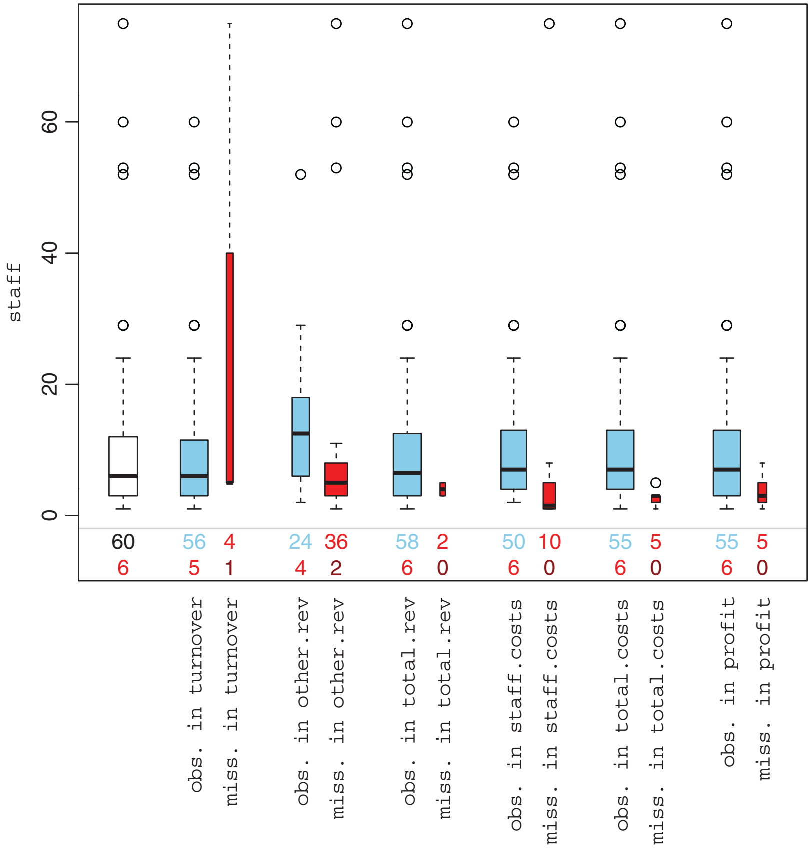


Figure 10.2 Parallel boxplots, comparing the distribution of *staff* conditional on the missingness of other variables.

The low *p* value indicates that the null hypotheses (means are equal across groups) may be rejected with a low probability (< 0.01) of error. The conclusion is that missingness of *other.rev* is to be treated as MAR with respect to *staff*.

One may wonder whether the reverse is also true: is the missingness of *staff* MAR with respect to the values observed in *other.rev*? A quick insight into this question can be obtained by drawing a so-called marginplot (here, using log-transformed variables to accommodate for their skew distributions).

```
dat <- log10(abs(retailers[c(3,5)]))
VIM::marginplot(dat, las=1, pch=16)
```

The marginplot (Figure 10.3) shows a scatterplot of cases where both variables are observed. The margins show, in light gray, boxplots of the variable depicted on the respective axis. These are contrasted with boxplots (in gray) of the same variable, but for the case where the other variable is missing. So from the boxplots in the margins of the *x*-axis, we read off that *other.rev* may be MAR with respect to *staff*. Similarly, from the boxplots in the *y*-axis, we read off that *staff* might be MAR with respect to *other.rev*. The actual values used to produce the dark gray boxplots are also represented in the

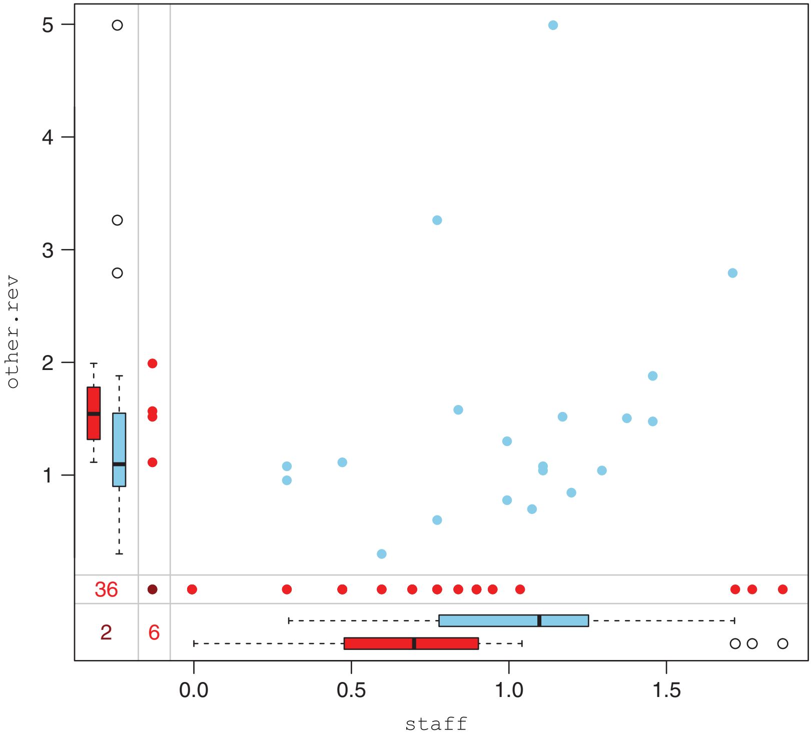


Figure 10.3 Marginplot of *other.rev* against *staff*.

margins as dark gray dots. The number of missing values per variable and the number of co-occurring missing values are denoted in the margins as well.

Exercises for Section 10.1

Exercise 10.1.1 *Perform a t-test to see whether there is enough evidence to support that (the logarithm of)* staff *is MAR with respect to (the logarithm of)* other.rev *in the* `retailers` *dataset.*

10.2 Model-Based Imputation

The literature on imputation methodology is extensive. Besides a broad range of general imputation methods, many methods have been developed with specific applications in mind. Examples include methodology for longitudinal data (Fitzmaurice *et al.*, 2008) or social network data (see Huisman (2009) and references therein). Here, we focus on a number of well-established methods covering a broad range of applications that are readily available in R.

In a predictive model, the target variable Y is described by an estimating function or algorithm f depending on one or more predictors $X = (X_0, X_1, X_2, \ldots, X_{p-1})$ and one or more parameters $\boldsymbol{\beta} = (\beta_0, \beta_1, \dots, \beta_{m-1})$.

$$Y = f(X; \boldsymbol{\beta}) + \varepsilon, \tag{10.4}$$

where ε is the residual of the model, the part of variation in Y that is not described by f. The predictor variables may be real-valued or categorical. In the latter case, a categorical variable taking K values is represented by $K - 1$ binary dummy variables. Likewise, the predicted variable Y can be real-valued or categorical. If Y is a binary variable, the possible values can be coded as 0 and 1. The form of f can be chosen to estimate the probability of Y taking the value 1, so it only takes values in the range $[0, 1]$. If Y can take K different values, we may label them as $1, 2, \dots, K$. One then determines K model functions f_k, each estimating the probability $P(Y = k)$.

The values for $\boldsymbol{\beta}$ are estimated by minimizing a loss function such as the negative log-likelihood over known values of (Y, X). Once the estimates $\hat{\boldsymbol{\beta}}$ are obtained, imputed values $\hat{y}$ for numerical Y are determined as

$$\hat{y} = f(\boldsymbol{x}; \hat{\boldsymbol{\beta}}) + e, \tag{10.5}$$

with $\boldsymbol{x}$ a vector of observed values for X and e a chosen residual value. A common choice is to set $e = 0$, so the imputed value is the best estimate of Y given $\boldsymbol{x}, f$, and the loss function. If one is interested in individual predictions only, setting $e = 0$ is the common choice. When dealing with imputation problems, one is often interested in reconstructing the (co)variance structure of a dataset. So, the other options include sampling e from $N(0, \hat{\sigma}^2)$, where $\hat{\sigma}^2$ is the estimated variance of ε or sampling e (uniformly) from the observed set of residuals. Methods where e is sampled are referred to as stochastic imputation methods. If Y is a categorical variable, the actual predicted value may be the one that is assigned the highest probability, so $\hat{y} = \text{argmax}_{k\in\{1,2,\dots,K\}}\{f_k(\boldsymbol{x}; \hat{\boldsymbol{\beta}})\}$. Alternatively, one can sample a value from $\{1, 2, \dots, K\}$ assuming the $f_k(\boldsymbol{x}; \hat{\boldsymbol{\beta}})$ as probability distribution over the domain of Y.

If Y is a real-valued variable, a common model is the linear model

$$Y = \sum_{j=0}^{p} X_j \beta_j + \varepsilon.$$

Here, β_0 usually (but not necessarily) represents the intercept, so $X_0 = 1$. Given a set of observations $\{(y_i, \boldsymbol{x}_i) : i = 1, 2, \dots, n\}$, the most popular loss function to estimate $\boldsymbol{\beta}$ the sum of squares, so

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}\in\mathbb{R}^{p+1}}{\text{argmin}} \sum_{i=1}^{n} [y_i - f(\boldsymbol{x}_{i.}; \boldsymbol{\beta})]^2, \tag{10.6}$$

where $\boldsymbol{x}_{i.}$ represents the ith row in $\boldsymbol{X}$. The resulting estimator can be interpreted as the conditional expectation of Y given X or $f(\boldsymbol{x}; \hat{\boldsymbol{\beta}}) = \hat{\mathrm{E}}(Y|X = \boldsymbol{x})$. There are several variations on the quadratic loss function including ridge regression (Hoerl and Kennard, 1970), lasso regression (Tibshirani, 1996), and their generalization: elasticnet regression (Zou and Hastie, 2005). Each of these methods forms an attempt to cope with high variability in the training data by adding terms that penalize the size of the $|\beta_j|$ $(j \neq 0)$. Other robust alternatives include the class of M-estimators. There, the loss function is adapted to decrease the contribution of highly influential records in the training set [see, e.g., Huber (2011) or Maronna *et al.* (2006)].

If we denote the matrix $\boldsymbol{X} = [\boldsymbol{x}_{.0}, \boldsymbol{x}_{.1}, \ldots, \boldsymbol{x}_{.p-1}]$, where the $\boldsymbol{x}_{.j}$ are columns of observed values (possibly including the intercept 'variable' $\boldsymbol{x}_0 = \mathbf{1}$), the imputed values can be written as

$$\hat{y} = \boldsymbol{X}\boldsymbol{\beta} + e. \tag{10.7}$$

It was demonstrated by Kalton and Kasprzyk (1986) and more extensively by de Waal *et al.* (2011, Chapter 7) that a surprising number of common imputation methods can be written in this form when the choices for the X_j and e are appropriately adapted. Methods that can be written in this form include (group) mean imputation, linear regression and ratio imputation, nearest-neighbor imputation, the deductive imputation method of Section 9.3.2,and several forms of stochastic imputation. If the (regularized) quadratic loss function is also allowed to vary, even more imputation methods can be written in this form. In particular, if the quadratic loss function is replaced with the least absolute deviation (LAD), we get

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}\in\mathbb{R}^p}{\operatorname{argmin}} \sum_{i=1}^{n} |y_i - f(\boldsymbol{x}_{i.}; \boldsymbol{\beta})|. \tag{10.8}$$

One can show that in this case [See, e.g., Koenker (2005); Chen *et al.* (2008)]

$$f(\boldsymbol{x}; \hat{\boldsymbol{\beta}}) = \widehat{\text{median}}(Y|X = \boldsymbol{x}).$$

Thus, imputing the conditional (group-wise) median can also be summarized under this notation.

Exercises for Section 10.2

Exercise 10.2.2 *In univariate random hot deck imputation, a missing value is replaced by copying the observed value from another, randomly chosen record. Show, by writing down a specific version of Eq. (*10.7*), how this imputation method can be expressed as a linear model with a specific type of residual e.*

10.3 Model-Based Imputation in R

The large amount of literature on imputation methodology is reflected in the large number of R packages implementing them. At the time of writing there are dozens of packages mentioning 'impute' or 'imputation' in their description. Here, we will demonstrate a number of imputation methods using the `simputation` package. The reason for choosing this particular package is it offers a consistent and (to R-users) familiar interface to many imputation models. The package relies mostly on other packages for computing the models and generating predictions. In some cases the backend can be chosen (e.g., one can make `simputation` use `VIM` for certain types of hotdeck imputations).

10.3.1 Specifying Imputation Methods with `simputation`

With the `simputation` package the specification of an imputation method always has the following form:

```
impute_<model-abbreviation>(dat, formula, [model-specific options], ...)
```

where `<model-abbreviation>` is replaced with an abbreviated name for the predictive model to be used (e.g., `lm` for linear models), `dat` is the dataset to be imputed, and `formula` specifies the relation between imputed and predicting variables. Depending on the method there may be some `simputation`-specific options, and all extra arguments (`...`) are passed to the underlying modeling functions.

The `formula` object is an expression of the form

```
imputed_variables ~ predicting_variables [ | grouping_variables ]
```

where `imputed_variables` specifies what variables should be imputed and `predicting_variables` specifies the combination of variables to be used as predictors. The terms enclosed in brackets are optional. The `grouping_variables` term can be used to specify a split-apply-combine strategy for imputation. The dataset is split according to the value combinations of grouping variables, the imputation model is estimated for each subset, values are imputed, and the dataset recombined.

Contrary to most modeling functions in R, the specification of imputed (dependent) variables is flexible and can contain multiple variables. The `simputation` package will simply loop over all variables to be imputed, estimating models as needed. For example, the specification

```
y ~ foo + bar
```

specifies that variable `y` should be imputed, using `foo` and `bar` as predictors. To impute multiple variables, one can just add variables on the left-hand side.

```
y1 + y2 + y3 ~ foo + bar
```

Here, `y1`, `y2`, and `y3` are imputed using `foo` and `bar` as predictors. The dot (`.`) stands for 'every variable not mentioned earlier', so

```
. ~ foo + bar
```

is to be interpreted as impute every variable, using `foo` and `bar` as predictors. It depends on the imputation method whether that means that `foo` and `bar` can also be imputed. The `simputation` package will remove predictors from the list of imputed variables when necessary. Finally, it is also possible to remove variables. The formula

```
. - x ~ foo + bar
```

means impute every variable except `x` using `foo` and `bar` as predictors.

The form that `predicting_variables` can take depends on the chosen imputation model. For example, in linear modeling, the option to model interaction effects (e.g., `foo:bar`) is relevant, while for other models it is not.

10.3.2 Linear Regression-Based Imputation

Linear regression imputation can be applied to impute numerical variables, using numerical and/or categorical variables and possibly their interaction effects as predictors. The model function is given by $f(\boldsymbol{x}, \boldsymbol{\beta}) = \boldsymbol{X}\boldsymbol{\beta}$, where $\boldsymbol{\beta}$ is estimated with Eq. (10.6). In particular, we can partition the vector of Y-values as $\boldsymbol{y} = (\boldsymbol{y}_o, \boldsymbol{y}_m)$, where o indicates where Y is observed and m indicates where Y is missing. Accordingly, the matrix $\boldsymbol{X}$ with predictor values can be partitioned in rows $\boldsymbol{X}_o$ where Y is observed and rows $\boldsymbol{X}_m$

where Y is missing. The value of $\boldsymbol{\beta}$ is then estimated over the observed values of y

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}\in\mathbb{R}^p}{\operatorname{argmin}} \|\boldsymbol{y}_o - \boldsymbol{X}_o\boldsymbol{\beta}\|^2, \tag{10.9}$$

after which the missing values can be estimated as

$$\hat{\boldsymbol{y}}_m = \boldsymbol{X}_m\hat{\boldsymbol{\beta}}.$$

With the `simputation` package, linear model imputation can be performed with the `impute_lm` function. In the following paragraphs a few columns of the `retailers` dataset from the `validate` package will be used.

```
library(simputation)
library(magrittr) # for convenience
data(retailers, package="validate")
retl <- retailers[c(1,3:6,10)]
head(retl, n=3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75       NA        NA      1130  NA
## 2  sc3     9     1607        NA      1607  NA
## 3  sc3    NA     6886       -33      6919  NA
```

We will be interested in imputing the values for `turnover`, `other.rev`, and `total.rev`. The `simputation` package relies on `lm` for estimating linear models, which means that several classes of imputation methods can be specified with ease.

In *mean imputation*, missing values are replaced by the column mean. To impute `turnover`, `other.rev`, and `total.rev` with their respective means, we specify

```
impute_lm(retl, turnover + other.rev + total.rev ~ 1) %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75 20279.48  4218.292      1130  NA
## 2  sc3     9  1607.00  4218.292      1607  NA
## 3  sc3    NA  6886.00   -33.000      6919  NA
```

It is well known that mean imputation leads to a gross underestimation of the variance of estimated means (when computed over the imputed dataset).

A slightly better procedure is to impute the *group mean*. Here, we impute missing variables using `size` (a size classification) as grouping variable.

```
impute_lm(retl, turnover + other.rev + total.rev ~ 1 | size) %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75 1420.375    315.50      1130  NA
## 2  sc3     9 1607.000   6169.25      1607  NA
## 3  sc3    NA 6886.000    -33.00      6919  NA
```

By specifying `size` after the vertical bar, we make sure that `simputation` does the split-apply-combine work over the grouping variable. The same result can be achieved as follows:

```
impute_lm(retl, turnover + other.rev + total.rev ~ size) %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75 1420.375    315.50      1130  NA
## 2  sc3     9 1607.000   6169.25      1607  NA
## 3  sc3    NA 6886.000    -33.00      6919  NA
```

where we let lm estimate a model with size as predictor. The latter method is slightly less robust. When one of the groups contains only missing values for one of the predicted variables, lm will stop, while the split-apply-combine procedure of simputation can handle such cases.

Ratio imputation uses the model $Y = \hat{R}X$, where $\hat{R}$ is the ratio of the mean of $\boldsymbol{y}_o$ and the mean of $\boldsymbol{x}_o$. It is equivalent to a linear model with a single predictor, no abscissa, weighted according to the reciprocal of the predictor. Here, the three variables are imputed with staff (the number of employees) as predictor.

```
impute_lm(retl, turnover + other.rev + total.rev ~ staff - 1
  , weight=1/retl$staff) %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75 26187.55 26426.132      1130  NA
## 2  sc3     9  1607.00  3171.136      1607  NA
## 3  sc3    NA  6886.00   -33.000      6919  NA
```

Ratio imputation is often used as a growth estimate, that is, in cases where a current value as well as a past value is known.

In *linear regression imputation*, one or more predictors may be used to impute a value based on a linear model. Below, the number of staff and turnover reported for value-added tax (vat) are used as predictors.

```
impute_lm(retl, turnover + other.rev + total.rev ~ staff + vat
          )%>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75       NA        NA      1130  NA
## 2  sc3     9     1607        NA      1607  NA
## 3  sc3    NA     6886       -33      6919  NA
```

Observe that in the first three rows, nothing is imputed. The reason is that in those cases, the predictor vat is missing. This illustrates a general property of simputation. The package will leave values untouched when one of the predictors is missing and return the partially imputed dataset.

Each of these models imputes the expected value, given zero or more predictors. They can be made stochastic by adding to each estimated value a random residual e, as denoted in Eq. (10.5). For model-based imputation methods, simputation supports three options: $e = 0$, this is the default; $e \sim N(0, \hat{\sigma}^2)$ with $\hat{\sigma}^2$, the estimated variance of the residuals; and e, sampled from the observed residuals. They can be specified with the add_residual option.

```
# make results reproducible
set.seed(1)
# add normal residual
impute_lm(retl
   , turnover + other.rev + total.rev ~ staff
   , add_residual = "normal") %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75 1100.659  11737.95      1130  NA
## 2  sc3     9 1607.000 -12914.62      1607  NA
## 3  sc3    NA 6886.000    -33.00      6919  NA
# add observed residual
```

```
impute_lm(retl
    , turnover + other.rev + total.rev ~ staff
    , add_residual = "observed") %>% head(3)
##    size staff turnover other.rev total.rev vat
## 1   sc0    75 7439.802  -659.214      1130  NA
## 2   sc3     9 1607.000  1015.605      1607  NA
## 3   sc3    NA 6886.000   -33.000      6919  NA
```

10.3.3 *M*-Estimation

The M-estimation method aims to reduce the influence of outliers on linear model coefficients by replacing the loss function of Eq. (10.9) with a suitable function ρ so that

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}\in\mathbb{R}^p}{\operatorname{argmin}} = \sum_{i=1}^{n_o} \rho(y_{o,i} - \boldsymbol{x}_{o,i\cdot}\boldsymbol{\beta}) = \sum_{i=1}^{n_o} \rho(\varepsilon_i),$$

where n_o is the number of observed records $\boldsymbol{x}_{o,i\cdot}$. To find $\hat{\boldsymbol{\beta}}$, one solves the system of equations for $\boldsymbol{\beta}$

$$\sum_{i=1}^{n_o} \frac{\partial\rho(\varepsilon_i)}{\partial\beta_j} = \sum_{i=1}^{n_o} \frac{d\rho(\varepsilon_i)}{d\varepsilon_i}\frac{\partial\varepsilon_i}{\partial\beta_j} = \sum_{i=1}^{n_o} \psi(\varepsilon_i)\frac{\partial\varepsilon_i}{\partial\beta_j} = \sum_{i=1}^{n_o} \psi(\varepsilon_i)x_{o,ij} = 0, \quad \text{for } j = 1, 2, \dots, p.$$

Here, we defined the so-called *influence function* $\psi(u) = d\rho(u)/du$. It determines the relative influence of each observation to the solution. If we set $\psi(u) = u$, then $\rho(u) = u^2/2$ (up to an unimportant additive constant) and Eq. (10.9) is returned.

The influence function is chosen so that it is less sensitive for increasing values of ε as the standard quadratic loss function. A few popular choices are proposals by Huber *et al.* (1964), Hampel *et al.* (1986), and Tukey's bisquare function, which are also available in R through the `MASS` package.

$$\psi_{\text{Huber}}(u) = u \text{ if } |u| < k, \text{ otherwise } k \cdot \operatorname{sign}(u),$$

$$\psi_{\text{Tukey}}(u) = u[1 - (u/c)^2]^2 \text{ if } u < c, \text{ otherwise } 0,$$

$$\psi_{\text{Hampel}}(u) = \begin{cases} u \text{ when } |u| \leq a, \\ a \cdot \operatorname{sign}(u) \text{ when } a < |u| \leq b, \\ a \cdot \operatorname{sign}(u)(r - |u|)/(1 - b) \text{ when } b < |u| \leq r, \\ 0 \text{ otherwise.} \end{cases}$$

The parameters k, c, a, b, and r are tuning parameters that determine the rate of increase as a function of u and the locations where ρ levels off. Figure 10.4 shows the shape of the ψ and ρ functions of Huber, Tukey, and Hampel. The constants were chosen so that the regression estimators have an efficiency of 95% as described, for example, by Koller and Mächler (2016), that is, $k = 1.345$, $c = 4.685$, $a = 1.5v$, $b = 3.5v$, and $r = 8v$, where $v = 0.902$.

With the `simputation` package, imputation based on M-estimated linear regression parameters can be done with the `impute_rlm` function. It uses the `rlm` function of the `MASS` package for coefficient estimation.

```
impute_rlm(retl, turnover + other.rev + total.rev ~ staff) %>% head(3)
##    size staff turnover other.rev total.rev vat
## 1   sc0    75 12927.56 159.16574      1130  NA
## 2   sc3     9  1607.00  16.29018      1607  NA
## 3   sc3    NA  6886.00 -33.00000      6919  NA
```

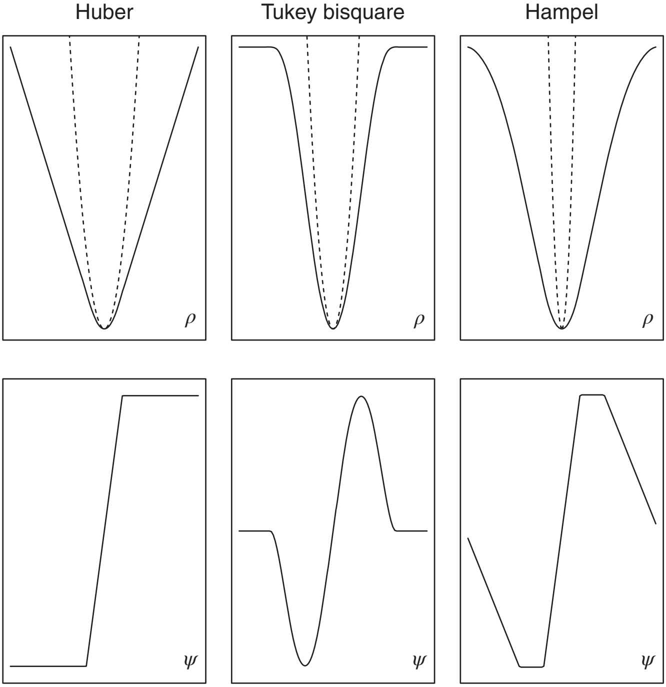


Figure 10.4 Three ρ- and ψ-functions used in *M*-estimation (plotted so that the scales of the *x* axes are comparable). For comparison, the traditional function $\rho(u) = u^2$ is plotted with dotted lines.

The default is to use Huber's ψ function with $c = 1.345$. Extra arguments are passed through to `rlm`. For example, `rlm` has the option to set `method="MM"`. This sets a number of options ensuring that the regression estimator has a high breakdown point (qualitatively, the fraction of outliers that may be present in the data before the estimator gives unacceptable results).

```
impute_rlm(retl, turnover + other.rev + total.rev ~ staff
 , method="MM") %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75 13529.59  75.34900      1130  NA
## 2  sc3     9  1607.00  13.24304      1607  NA
## 3  sc3    NA  6886.00 -33.00000      6919  NA
```

10.3.4 Lasso, Ridge, and Elasticnet Regression

With `impute_en`, elasticnet regression is used to compute imputations. In linear elasticnet regression, the parameters are estimated as

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}\in\mathbb{R}^p}{\operatorname{argmin}} \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 + \lambda \left[\frac{1-\alpha}{2}\|\tilde{\boldsymbol{\beta}}\|^2 + \alpha\|\tilde{\boldsymbol{\beta}}\|_1\right],$$

where $\|\cdot\|$ is the familiar Euclidean norm and $\|\cdot\|_1$ the L_1-norm or the sum over absolute values of the coefficients of its argument. The penalty term is defined in terms of $\tilde{\boldsymbol{\beta}}$, which denotes all coefficients except the intercept (when present). The parameter α allows one to shift smoothly from ridge regression ($\alpha = 0$) to lasso regression ($\alpha = 1$), while λ determines the overall strength of the penalty. The characteristic difference between lasso and ridge regression is that in the case of correlated predictors, lasso regression tends to push one or more coefficients to zero, while ridge regression spreads the value of coefficients over multiple correlated variables.

The simputation package implements elasticnet imputation through the impute_en function, which depends on the glmnet package of Friedman *et al.* (2010).

```
impute_en(retl, turnover + other.rev + total.rev ~ staff + size
   , s=0.005, alpha=0.5) %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75 13360.82 -8354.709      1130  NA
## 2  sc3     9  1607.00 10476.443      1607  NA
## 3  sc3    NA  6886.00   -33.000      6919  NA
```

Here, the predictor size is added since the glmnet package does not accept models with less than two predictors. The parameter s determines the size of the overall penalty factor λ used in predictions, and we (arbitrarily) set alpha=0.5.

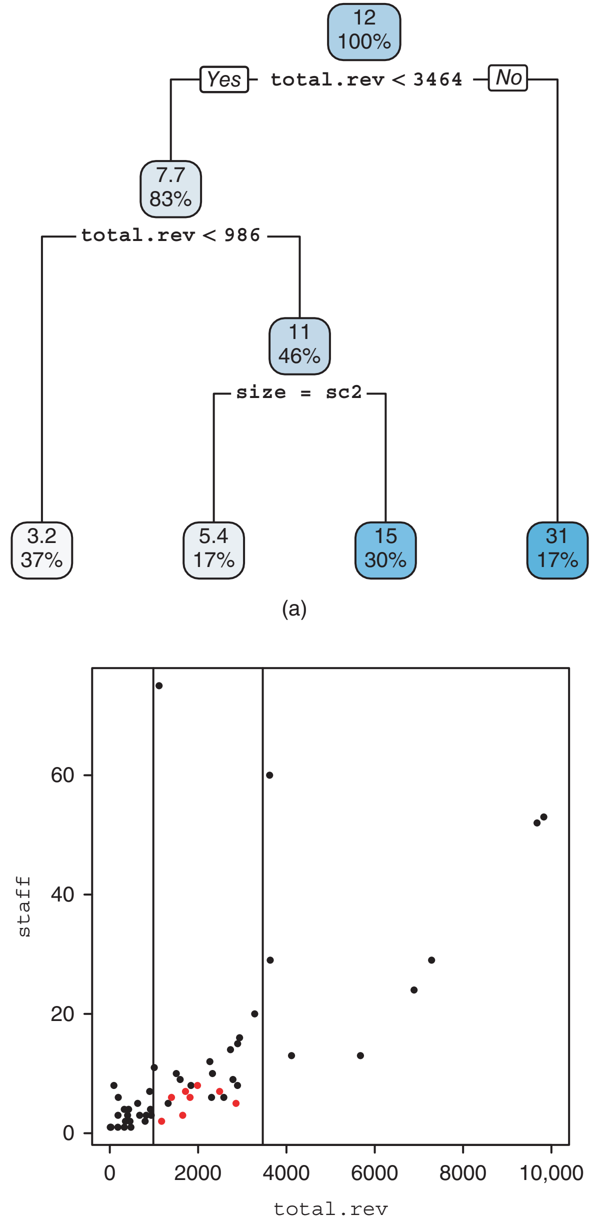
10.3.5 Classification and Regression Trees

Decision tree models can be used in situations where the predicted variable is numeric and its dependence on the predictors is highly nonlinear, or when the predicted variable is categorical with probabilities that have complex dependence on predictor variables. In both the cases, the predictors can be numeric or categorical. Because decision tree models can be used to predict quantitative as well as qualitative variables, they are often referred to as classification and regression trees or CART, for short. The term was introduced by Breiman *et al.* (1984), and it also refers to a specific method for setting up the decision tree (which will be treated below).

Consider again a predicted variable Y (to be imputed) and a set of predictors $X = (X_1, X_2, \ldots, X_p)$. The idea is to partition the set of all possible value combinations $(x_1, x_2, \ldots, x_p)$ into disjunct regions, such that the corresponding value of Y is as homogeneous as possible in each region. For numeric Y, homogeneous usually means 'a small variance'; for categorical data, it means 'a high proportion of a single category'. Given a particular record $(x_1, x_2, \ldots, x_p)$, one follows a binary decision tree $\hat{T}$ to see in what region the record falls. The predicted value for numeric Y is then the mean value within the region (although more robust estimators are sometimes used as well), and the predicted value for categorical Y is the category with the highest prevalence.

Before considering how such a decision tree is constructed, consider the example of Figure 10.5. Here, we used the rpart package to build a predictive model for the *staff* variable in the retailers dataset. Depicted are two representations of the resulting model. In Figure 10.5(a), the decision tree is shown. Each nonterminal node contains two numbers and a decision rule. The root node represents 100% of the records, and the mean value for *staff* over all those records equals 12 (rounded). If we partition the dataset according to the rule total.rev < 3464, we get 83% records for which this rule holds, with a mean number of staff of 7.7 and 17% records for which this total.rev >= 3464, with a mean number of staff of 31. The latter case ends in a terminal node and thus corresponds with a single partition of the feature space. The group of records falling into this category are on the right of the second vertical line in

Figure 10.5 (a) A decision tree for estimating number of staff in the `retailers` dataset, computed with `rpart`. (b) The space partitioning. The vertical bars indicate the limits represented in the top node and the subdivision indicated by its left child node. In the middle region, white dots indicate records where `size=="sc2"`.



(a)

(b)

Figure 10.5(b). The group of records on the left (with `total.rev < 3464`) are subdivided by having smaller or larger `total.rev` of 968. The latter are subdivided once more based on whether the `size` variable equals `"sc2"` (depicted in white in panel (b)). The terminal nodes of the tree contain the actual predictions made for each partition along with the size of the partition.

When comparing this procedure with Eq. (10.5), we see that here, the model function is a procedure $f(\boldsymbol{x}, \hat{T})$, that is parameterized by a decision tree $\hat{T}$. Based on the values of $\boldsymbol{x}$, the tree is traversed until a terminal node is reached and the prediction returned.

The tree itself is built up iteratively. Given a $n \times p$ dataset $\boldsymbol{X}$ and a set of values $\boldsymbol{y}$, the optimal split based on each variable X_j is computed. Of those p splits, the one resulting in the lowest error is chosen. This process is then repeated for each partition recursively. Note that for any dataset, it is in principle possible to create a perfect partition by growing the tree until each leaf has a single record in it (or only records with equal values for Y). In practice, one stops at some minimum number of records. The resulting (still large) tree is then pruned by removing leaf nodes from the bottom up. The final result is determined by a trade-off between error minimization and tree size $|T|$:

$$\hat{T} = \operatorname*{argmin}_{T\in\mathcal{T}} \sum_{i=1}^{n} \text{error}(y_i, f(\boldsymbol{x}_{i\cdot}, T)) + \alpha|T|.$$

Here, $\mathcal{T}$ is the set of subtrees that can be obtained by pruning the initial tree. The function 'error' records the mismatch between prediction and observation, appropriate for the variable Y (e.g., standard deviation for numerical variables end mismatch ratio for categorical variables). The term $\alpha|T|$ penalizes the number of nodes $|T|$. Here, α is referred to as the *cost-complexity parameter*. It is determined automatically by computing $\hat{T}$ for a series of values of α and applying cross-validation to select the best one [see also James *et al.* (2013, Chapter 8) or Hastie *et al.* (2001, Chapter 9)].

With the `simputation` package, CART-based imputation can be performed with the `impute_cart`. The specification of predictor variables tells `impute_cart` what variables can be used in the tree. In many cases, one can choose all variables except the predicted since decision trees have variable selection built-in.

```
impute_cart(retl, staff ~ .) %>% head(3)
##   size    staff turnover other.rev total.rev vat
## 1  sc0 75.00000       NA        NA      1130  NA
## 2  sc3  9.00000     1607        NA      1607  NA
## 3  sc3 30.66667     6886       -33      6919  NA
```

Imputation took place in the third record. The imputation value can be traced by following the decision tree of Figure 10.5. The *total revenue* in the third record equals 6919. Since this is larger than 3464, we end in a leaf node immediately and predict a value of 30.67.

One advantage of CART models over linear models for imputation is their resilience against missing values in predictors. For a linear model, the imputation $f(\boldsymbol{x}, \hat{\boldsymbol{\beta}})$ cannot be estimated when any of the x_i happens to be missing unless it is somehow imputed. In a CART model, the actual decision tree contains more information than shown in Figure 10.5. Anticipating on possible missing predictors, each node stores one or more backup split rules based on other predictors present in the dataset. If during prediction some value x_i is found to be missing, its first so-called surrogate variable is used to decide the split. If the surrogate is also missing, the next one is used, and so on, until an observed surrogate is found, or no surrogates are left. In the latter case, the most populated child node is chosen. The loss of quality of prediction is smaller when surrogates are highly correlated with the primary splitting variables.

10.3.6 Random Forest

Random forest (Breiman, 2001) is an ensemble-based improvement over CART. It can be used to predict both qualitative and quantitative variables. The idea is to take bootstrap samples from the original data, and grow a decision tree for each sample. Moreover, at each split, a subset of the p available predictors (typically about $\sqrt{p}$) is randomly chosen as possible splitting variables. Randomizing the available splitting variables is used to decrease correlation between the trees.

Training a random forest model thus results in a set of B trees $\{\hat{T}_1, \hat{T}_2, \ldots, \hat{T}_B\}$ called a *forest*. If the predicted variable is numerical, the prediction is an aggregate over the individual predictions such as the mean,

$$\hat{y} = \frac{1}{B}\sum_{b=1}^{B} f(\boldsymbol{x}, \hat{T}_b),$$

but in principle, it is possible to use a robust aggregate such as the median as well. For categorical variables, the majority vote over the trees is taken.

With the simputation package, random forest models can be employed for imputation as follows:

```
impute_rf(retl, staff ~ .) %>% head(3)
##    size staff turnover other.rev total.rev vat
## 1   sc0     75       NA        NA      1130  NA
## 2   sc3      9     1607        NA      1607  NA
## 3   sc3     NA     6886       -33      6919  NA
```

Random forest models are somewhat less resilient against missing predictors than the CART models discussed in the previous paragraph. The underlying Fortran code by Breiman and Cutler (2004) is able to use a rough imputation scheme on the training set to generate the forest (this can be set by passing na_action=na.rough fix; this will impute medians for numeric data and modes for categorical data). For prediction, however, not every variable needs to be present: the average can be taken over the subset of trees that do return a value. However, for small datasets such as in this example, it may occur that all trees return NA so no prediction is possible. Here, the problem can be partially resolved by removing a variable with little observations from the list of predictors.

```
impute_rf(retl, staff ~ . - vat) %>% head(3)
##    size    staff turnover other.rev total.rev vat
## 1   sc0 75.00000       NA        NA      1130  NA
## 2   sc3  9.00000     1607        NA      1607  NA
## 3   sc3 20.23483     6886       -33      6919  NA
```

Besides simputation, there are other R packages implementing imputation based on random forests. The missForest package by Stekhoven and Bühlmann (2012) implements an iterative imputation procedure. For initiation, missing values get imputed using a simple rule. Next, a random forest is trained on the completed dataset, yielding updated imputation values. This second step is repeated until a convergence criterion has been satisfied. When installed, the missForest package can be interfaced via simputation using impute_mf.

```
impute_mf(retl, staff ~ .) %>% head(3)
##   missForest iteration 1 in progress...done!
##   missForest iteration 2 in progress...done!
##   size staff turnover other.rev total.rev vat
## 1  sc0 75.00       NA        NA      1130  NA
## 2  sc3  9.00     1607        NA      1607  NA
## 3  sc3 18.15     6886       -33      6919  NA
```

Here, all variables are imputed (iteratively) and used as predictors, but only the variables on the left-hand side for the `formula` are copied to the resulting dataset. With the formula `. ~ .` all variables can be imputed.

10.4 Donor Imputation with R

In donor imputation, a missing value in one record is replaced with an observed value that is copied from another and somehow otherwise similar record. The record from which the value is copied is referred to as the 'donor' record; hence, the name of the method. Donor imputation is also referred to as *hot deck imputation*. The etymology of this term derives from the state of the art in computing when the method was first applied. Researchers would 'hot deck impute' by drawing from a deck of computer punch cards representing records (Andridge and Little, 2010; Cranmer and Gill, 2013).

When compared to model-based imputation, the advantage of donor imputation is that the imputed value is always an actually existing (observed) value. Statistical models always run the risk of predicting a value that is not (physically) possible, especially when extrapolating beyond the observed range of values. The downside of donor imputation is that in spite of its wide application, theoretical underpinning is not as strong as for model-based methods. Moreover, Andridge and Little (2010) conclude in their extensive review that no consensus exists on the best way to apply hot deck imputation methods, and note that 'many multivariate hot deck methods seem relatively *ad hoc*'. Nevertheless, hot deck methods have been commonly applied for a long time in areas related to official statistics [see Ono and Miller (1969); Bailar and Bailar (1979); and Cox (1980) for some early applications and method comparisons] and to a lesser extend in medical or epidemiological settings. Method comparisons are given in, for example, Barzi and Woodward (2004); Engels and Diehr (2003); Perez *et al.* (2002); Reilly and Pepe (1997); Tang *et al.* (2005), and Twisk and de Vente (2002).

Hot deck imputation methods are commonly categorized along two dimensions. The first dimension distinguishes between methods where multiple missing values in a record are imputed from the same donor (multivariate donor imputation) and methods where a separate donor may be appointed for each missing variable. The main advantage of multivariate donor imputation is that one only imputes valid and existing value combinations, so that imputed values cannot introduce inconsistencies. The downside is that the number of possible donors may be greatly reduced as the number of missing values in a record increases. The hot deck donor imputation routines in `simputation` have an option called `pool`, which control this behavior. Its possible values are `"complete"`: Use only complete records as donor pool and perform multivariate imputation. This is the default

`"univariate"`: A new donor is sought for each variable.

`"multivariate"`: For each occurring pattern of missingness find suitable donors and perform multivariate imputation.

The second dimension distinguishes between the various ways donor records are determined, and each of these methods (discussed next) can be executed in a univariate or multivariate fashion.

10.4.1 Random and Sequential Hot Deck Imputation

In *random hot deck imputation*, a donor is sampled from a donor pool. Often, a dataset is separated into *imputation cells* for which one or more auxiliary variables have the same values. With the simputation package, the imputation cells are determined by the right-hand side of the formula object specifying the model (we continue with the retl dataset constructed in the previous paragraph).

```
set.seed(1) # make reproducible
# random hot deck imputation (multivariate; complete cases are donor)
impute_rhd(retl, turnover + other.rev + total.rev ~ size) %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75      359         9      1130  NA
## 2  sc3     9     1607     98350      1607  NA
## 3  sc3    NA     6886       -33      6919  NA
```

In the above example, the dataset is split according to the size class label, and data are imputed in univariate manner. That is, for each variable, a value is sampled from all observed values within the same size class. If multiple categorical variables are used to define imputation cells, the donor pools can quickly decrease in size, leading possibly to many imputations of the same value. Setting pool="univariate" can alleviate this issue to a small extend since per-variable donor pools are generally larger than multivariate donor pools.

```
# random hot deck imputation (univariate)
impute_rhd(retl, turnover + other.rev + total.rev ~ size
 , pool="univariate") %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75      197       622      1130  NA
## 2  sc3     9     1607        33      1607  NA
## 3  sc3    NA     6886       -33      6919  NA
```

By default, records are drawn uniformly from the pool, but one can pass a numeric vector prob assigning a probability to each record in the imputed data. Probabilities will be rescaled as necessary, depending on grouping and donor pool specification.

In *sequential hot deck*, one sorts the dataset using one or more variables, and missing values in a record are taken from the first preceding or ensuing record that has a value. If values are taken from preceding records, the method is referred to as *last observation carried forward* or LOCF in short, if values are taken from ensuing records, the method is referred to as *next observation carried backward* (NOCB). With the simputation package, sequential hot deck is executed with the `impute_shd` function. The 'predictor variables' in the formula are used to sort the data (with every variable after the first used as tie-breaker for the previous one).

```
impute_shd(retl, turnover + other.rev + total.rev ~ staff) %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75     9067       622      1130  NA
## 2  sc3     9     1607        38      1607  NA
## 3  sc3    NA     6886       -33      6919  NA
```

The sort order is always 'increasing', but with the argument `order`, one can choose to use between `"nocb"` (the default) and `"locb"` imputation.

Both random and sequential hot decks are implemented in the `simputation` source code. Especially for large datasets and many groups, it is beneficial to use the faster implementation provided by the `VIM` package. This is possible for both `impute_rhd` and `impute_shd` by setting `backend="VIM"`. Options specific to `simputation` (such as the `order` argument) will be ignored, but one can pass any argument of `VIM::hotdeck` to `impute_rhd` or `impute_shd` for detailed control over the imputation method.

10.4.2 *k* Nearest Neighbors and Predictive Mean Matching

In the k-nearest-neighbor (knn) method, a similarity measure is used to find the knns to a record containing missing values. Next, a donor value is determined. Donor value determination can be done by randomly selecting from the k neighbors or, for example (in the case of categorical data), by choosing the majority value. A particularly popular similarity measure is that of Gower (1971). Given two records $\boldsymbol{r}$ and $\boldsymbol{s}$, each with n variables that may be numeric, categorical, or missing, Gower's similarity measure d_g can be written as

$$d_g(\boldsymbol{r},\boldsymbol{s}) = \frac{\sum_{j=1}^{n} w_j\delta(r_j,s_j)}{\sum_{j=1}^{n} w_j}.$$

The values of w_j and δ depend on the variable type. If the jth variable is numeric, then

$$\delta(r_j,s_j) = 1 - \frac{|r_j - s_j|}{\text{range}(j)} \quad \text{if } r_j \text{ and } s_j \text{ observed, otherwise } 0.$$

Here, range(j) is the observed range of the jth variable. If the jth variable is categorical, then δ is defined as

$$\delta(r_j,s_j) = 1 \quad \text{if } r_j = s_j \text{ and both observed, otherwise } 0.$$

For numerical and categorical variables, the importance weights w_j may be chosen at will, but they are usually set to 1 or 0, where setting $w_j = 0$ amounts to excluding the jth variable from the similarity calculation. For a dichotomous (`logical`, in R) variable, δ is yet defined differently, namely,

$$\delta(r_j,s_j) = r_j \wedge s_j \quad \text{if } r_j \text{ and } s_j \text{ observed, otherwise } 0$$

while the weights are defined as

$$w_j = r_j \vee s_j \text{ if } r_j \text{ and } s_j \text{ observed, otherwise } 0.$$

Here, we identify the logical outputs true with 1 and false with 0. The rationale is that a dichotomous variable only adds to the similarity when both variables are true. If any of the two variables is true they add to the weight in the denominator.

In the simputation package, knn imputation based on Gower's similarity is performed with the impute_knn function. The predictor variables in the formula argument specify which variables are used to determine Gower's similarity. Below, we use all variables by specifying the dot (.). The default value for $k = 5$, but by setting $k = 1$, values are copied directly from the nearest neighbor.

```
impute_knn(retl, turnover + other.rev + total.rev ~ ., k=1) %>% head(3)
##    size staff turnover other.rev total.rev vat
## 1   sc0    75     9067       622      1130  NA
## 2   sc3     9     1607        13      1607  NA
## 3   sc3    NA     6886       -33      6919  NA
```

Like for the random and sequential hot deck imputation procedures, the donor pool can be specified ("complete", "univariate", or "multivariate") and the VIM package can be used as computational backend.

Predictive mean matching (PMM) is a nearest-neighbor imputation method where the donor is determined by comparing predicted donor values with model-based predictions for the recipient's missing values. It can therefore be seen as a method that lies between the purely model-based and purely donor-based imputation methods. On one hand, it partially shares the benefits of both approaches, utilizing the power of predictive modeling while making sure only observed values are imputed. On the other hand, it inherits some of the intricacies of both worlds, such as issues with model selection and the possibility of small donor pools. In practice, PMM has become a popular method, and the popular mice package for multiple imputation (van Buuren and Groothuis-Oudshoorn, 2011) uses it as default imputation method.

In simputation, PMM is achieved with the impute_pmm function. Besides the data to be imputed and a predictive model-specifying formula, it takes one of the impute_ functions as an argument to preimpute the recipients with a chosen model. By default, impute_lm is used so the formula object must specify a linear model.

```
impute_pmm(retl, turnover + other.rev + total.rev ~ staff) %>% head(3)
##    size staff turnover other.rev total.rev vat
## 1   sc0    75     7271        30      1130  NA
## 2   sc3     9     1607      1831      1607  NA
## 3   sc3    NA     6886       -33      6919  NA
```

However, one can switch to a robust linear model based on the *MM*-estimator as follows:

```
impute_pmm(retl, turnover + other.rev + total.rev ~ staff
  , predictor=impute_rlm, method="MM") %>% head(3)
##    size staff turnover other.rev total.rev vat
## 1   sc0    75     9067       622      1130  NA
## 2   sc3     9     1607        13      1607  NA
## 3   sc3    NA     6886       -33      6919  NA
```

10.5 Other Methods in the simputation Package

There are a few other imputation approaches supported by the simputation package that facilitate certain type of imputations.

The first method is a utility function allowing to replace missing values with a constant. For example,

```
impute_const(retl, other.rev ~ 0) %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75       NA         0      1130  NA
## 2  sc3     9     1607         0      1607  NA
## 3  sc3    NA     6886       -33      6919  NA
```

Such an imputation strategy implies strong assumptions that in some cases may nevertheless be reasonable. For example, one could assume that values that are not submitted by a respondent can be interpreted as 'not applicable' or in this case zero. Obviously, one should very carefully test such an assumption since they can introduce severe bias in estimates based on the data.

The second method is referred to by de Waal *et al.* (2011) as *proxy imputation*. Here, the missing value is estimated by copying a value from the same record, but from another variable. For example, we may estimate the variable *total turnover* in the `retailers` dataset by copying the amount of turnover reported to the tax office for `vat`.

```
impute_proxy(retl, total.rev ~ vat) %>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75       NA        NA      1130  NA
## 2  sc3     9     1607        NA      1607  NA
## 3  sc3    NA     6886       -33      6919  NA
```

This method is useful only when both the economic definition of turnover and the legal definition used by the tax authorities coincide or are very close.

The `simputation` package is more flexible than De Waal *et al.*'s original definition and also allows for imputing functions of variables.

```
impute_proxy(retl
  , turnover ~ mean(turnover/total.rev,na.rm=TRUE) * total.rev) %>%
  head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75 1351.159        NA      1130  NA
## 2  sc3     9 1607.000        NA      1607  NA
## 3  sc3    NA 6886.000       -33      6919  NA
```

Here, the right-hand side of the formula may evaluate to a vector of unit length or to a vector with length equal to the number of input rows. Proxy imputation also accepts grouping, so imputing the group mean can be done as follows:

```
impute_proxy(retl, turnover ~ mean(turnover, na.rm=TRUE) | size)
%>% head(3)
##   size staff turnover other.rev total.rev vat
## 1  sc0    75 1420.375        NA      1130  NA
## 2  sc3     9 1607.000        NA      1607  NA
## 3  sc3    NA 6886.000       -33      6919  NA
```

10.6 Imputation Based on the EM Algorithm

The purpose of the Expectation–Maximization algorithm (Dempster *et al.*, 1977) is to estimate the maximum-likelihood estimate for parameters of a multivariate probability distribution in the presence of missing data. As such, it is not an imputation method. Imputations can be generated by computing the expected value for missing items conditional on the observed data or by sampling them from the conditional multivariate distribution.

Contrary to the model-based imputation methods discussed up until now, in the EM algorithm, there is no fixed distinction between predicting and predicted variables—one simply uses everything available to estimate the parameters of some model distribution. This also means that one needs to assume a distributional form (e.g., multivariate normal) for the combined variables in the dataset.

10.6.1 The EM Algorithm

Consider again a set of random variables $X = (X_1, X_2, \ldots, X_m)$ with a joint probability distribution $P(X|\boldsymbol{\theta})$ parameterized by a numeric vector $\boldsymbol{\theta}$. We denote a realization of X as a row vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)^T$ and a set of n realizations as a $n \times m$ matrix $\boldsymbol{X}$. A common way to estimate the parameters of P given a set of observations is to assume a distributional form for P (say, multinormal) and then solve the following maximization problem.

$$\begin{aligned}
\hat{\boldsymbol{\theta}} &= \underset{\boldsymbol{\theta}\in\Omega}{\operatorname{argmax}}\, P(\boldsymbol{\theta}|\boldsymbol{X}) \\
&= \underset{\boldsymbol{\theta}\in\Omega}{\operatorname{argmax}}\, P(\boldsymbol{X}|\boldsymbol{\theta})\frac{P(\boldsymbol{\theta})}{P(\boldsymbol{X})} \\
&= \underset{\boldsymbol{\theta}\in\Omega}{\operatorname{argmax}}\ \ln P(\boldsymbol{X}|\boldsymbol{\theta}) + \ln P(\boldsymbol{\theta}) - \ln P(\boldsymbol{X}),
\end{aligned}$$

where Ω is the space of possible values for $\boldsymbol{\theta}$. Furthermore, we used Bayes' rule in the second line, and in the third line, we used that taking the logarithm does not alter the location of the maximum. If there is no prior knowledge about $\boldsymbol{\theta}$, we may assume that $\boldsymbol{\theta}$ is uniformly distributed over Ω, so $P(\boldsymbol{\theta})$ is constant. Since $P(\boldsymbol{X})$ is also independent of $\boldsymbol{\theta}$, the maximization problem can be simplified to

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}\in\Omega}{\operatorname{argmax}}\ \ln P(\boldsymbol{X}|\boldsymbol{\theta}) \equiv \underset{\boldsymbol{\theta}\in\Omega}{\operatorname{argmax}}\, \ell(\boldsymbol{\theta}|\boldsymbol{X}), \tag{10.10}$$

where we introduced the notation $\ell(\boldsymbol{\theta}|\boldsymbol{X})$, which is referred to as the *maximum-likelihood* function. Correspondingly, Eq. (10.10) is referred to as the *maximum-likelihood estimator*. It returns the value of $\boldsymbol{\theta}$ such that $\boldsymbol{X}$ is the most likely observed data. To actually solve this equation, one substitutes a distribution (e.g., multinormal), equates the right-hand side to zero, and solves for $\boldsymbol{\theta}$.

If only part of the realizations $\boldsymbol{X}$ are actually observed, this maximization problem cannot be solved. To move forward, we partition $\boldsymbol{X}$ into the observed values $\boldsymbol{X}_o$ and missing values $\boldsymbol{X}_m$. Using Bayes' rule twice, we can write

$$\begin{aligned}
\ln P(\boldsymbol{X}|\boldsymbol{\theta}) &= \ln P(\boldsymbol{X}_o, \boldsymbol{X}_m|\boldsymbol{\theta}) \\
&= \ln P(\boldsymbol{X}_m|\boldsymbol{X}_o, \boldsymbol{\theta}) + \ln P(\boldsymbol{X}_o, \boldsymbol{\theta}) - \ln P(\boldsymbol{X}) \\
&= \ln P(\boldsymbol{X}_m|\boldsymbol{X}_o, \boldsymbol{\theta}) + \ln P(\boldsymbol{X}_o|\boldsymbol{\theta}) + 2\ln P(\boldsymbol{\theta}) - \ln P(\boldsymbol{X}).
\end{aligned}$$

Again, dropping terms not depending on $\boldsymbol{\theta}$ (and we already assumed that $P(\boldsymbol{\theta})$ is constant), we find that maximizing the likelihood function of Eq. (10.10) can equivalently be written as the maximization over the likelihood function

$$\ell(\boldsymbol{\theta}|\boldsymbol{X}) = \ell(\boldsymbol{\theta}|\boldsymbol{X}_o) + \ln P(\boldsymbol{X}_m|\boldsymbol{X}_o, \boldsymbol{\theta}). \tag{10.11}$$

Since $\boldsymbol{X}_m$ is unknown, this expression cannot be maximized. The idea of Dempster *et al.* (1977) is to replace the maximum-likelihood function with its expected value with regard to the missing values.

$$\begin{aligned} E[\ell(\boldsymbol{\theta}|\boldsymbol{X})] &= \int_{\Omega_m} d\boldsymbol{X}_m \ell(\boldsymbol{\theta}|\boldsymbol{X}) P(\boldsymbol{X}_m|\boldsymbol{X}_o,\boldsymbol{\theta}) \\ &= \ell(\boldsymbol{\theta}|\boldsymbol{X}_o) + \int_{\Omega_m} d\boldsymbol{X}_m \ln P(\boldsymbol{X}_m|\boldsymbol{X}_o,\boldsymbol{\theta}) P(\boldsymbol{X}_m|\boldsymbol{X}_o,\boldsymbol{\theta}). \end{aligned}$$

Here, integration is over all unobserved variables, where Ω_m indicates the domain of possible values for the variables in $\boldsymbol{X}_m$. In the second line, Eq. (10.11) was substituted. The integral in the second line is the negative entropy of the distribution $P(\boldsymbol{X}_m|\boldsymbol{X}_o,\boldsymbol{\theta})$. It is therefore usually denoted $H(\boldsymbol{\theta})$. In principle, one can numerically maximize the above expression to obtain $\hat{\boldsymbol{\theta}}$. However, if P has a simple form, and we choose some value $\boldsymbol{\theta}'$, the integral

$$H(\boldsymbol{\theta}|\boldsymbol{\theta}') = \int_{\Omega_m} d\boldsymbol{X}_m \ln P(\boldsymbol{X}_m|\boldsymbol{X}_o,\boldsymbol{\theta}) P(\boldsymbol{X}_m|\boldsymbol{X}_o,\boldsymbol{\theta}'),$$

can often be worked out explicitly. Next, an updated value for $\boldsymbol{\theta}'$ can be found by maximizing $\ell(\boldsymbol{\theta}|\boldsymbol{X}_o) + H(\boldsymbol{\theta}|\boldsymbol{\theta}')$ as a function of $\boldsymbol{\theta}$. The Expectation–Maximization algorithm is indeed an iteration over this procedure. The crucial result of Dempster *et al.* (1977) is that the value of $\ell(\boldsymbol{\theta}'|\boldsymbol{X}_o)$ must increase at each iteration and converge to a maximum under mild conditions, including cases where $P(\boldsymbol{X}|\boldsymbol{\theta})$ is a member of the regular exponential family.

Procedure 10.6.1 EM algorithm

```
Input: A data matrix $\boldsymbol{X}$ with observed values $\boldsymbol{X}_o$.
Initialize: Choose a $\boldsymbol{\theta}'$.
  Repeat until convergence:
        $Q(\boldsymbol{\theta}|\boldsymbol{\theta}') \leftarrow \ell(\boldsymbol{\theta}|\boldsymbol{X}_o) + H(\boldsymbol{\theta}|\boldsymbol{\theta}')$        (E-step)
        $\boldsymbol{\theta}' \leftarrow \underset{\boldsymbol{\theta}\in\Omega}{\operatorname{argmax}} Q(\boldsymbol{\theta}|\boldsymbol{\theta}')$        (M-step)
Output: $\hat{\boldsymbol{\theta}} = \boldsymbol{\theta}'$.
```

If P is a member of the regular exponential family, the expressions to be evaluated by the algorithm can be simplified by expressing them in terms of (expected values of) sufficient statistics [e.g., de Waal *et al.* (2011, Chapter 8) or Schafer (1997, Chapter 5)]. The regular exponential family includes a wide range of commonly applied distributions including the (multivariate) normal, Bernoulli, exponential, (negative) binomial, Poisson, and gamma distributions [see, e.g., Brown (1986)]. Indeed, many implementations of the EM algorithm demand that P is from the regular exponential family. The `Amelia` package discussed below is restricted to models based on the multivariate normal distribution.

Advantages of the EM algorithm include that it is a simple and well-understood algorithm that is guaranteed to converge in principle. Given a distribution from the exponential family, the E and M steps can be computed very quickly. Also, since the algorithm provides an estimate for the full multivariate distribution, it has a good chance of correcting for the random (MAR) mechanism. A disadvantage is that computation may take many iterations since the convergence criterion (measured in terms of the difference in $\boldsymbol{\theta}'$ between iterations) decreases approximately linearly with the number of iterations (Schafer, 1997). Convergence can be especially slow when the fraction of missing values is high or when the model distribution is a poor description of the actual data distribution. Furthermore, the EM algorithm does not immediately provide a variance estimate for $\hat{\boldsymbol{\theta}}$.

10.6.2 EM Imputation Assuming the Multivariate Normal Distribution

Probably, one of the most implemented models for EM estimation is the model where numeric variables are distributed according to the multivariate normal distribution. This distribution is parameterized by the mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, with the probability density function given by

$$f(\boldsymbol{x}|\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{\exp\left\{\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right\}}{\sqrt{2\pi\det\boldsymbol{\Sigma}}}.$$

Because of its relatively simple form, the update rules for the E and M steps can be worked out yielding expressions that eventually be evaluated using simple matrix algebra and linear system solving.

Before stating the algorithm, consider a record $\boldsymbol{x} = (\boldsymbol{x}_o, \boldsymbol{x}_m)$ with observed values $\boldsymbol{x}_o$ and missing values $\boldsymbol{x}_m$. Given an estimate for the mean vector and covariance matrix, these can be rearranged accordingly, so

$$\hat{\boldsymbol{\mu}} = \begin{pmatrix}\hat{\boldsymbol{\mu}}_o\\ \hat{\boldsymbol{\mu}}_m\end{pmatrix}, \text{and } \hat{\boldsymbol{\Sigma}} = \begin{pmatrix}\hat{\boldsymbol{\Sigma}}_{oo} & \hat{\boldsymbol{\Sigma}}_{om}\\ \hat{\boldsymbol{\Sigma}}_{mo} & \hat{\boldsymbol{\Sigma}}_{mm}\end{pmatrix}.$$

Here, $\hat{\boldsymbol{\Sigma}}_{oo}$ is the estimated covariance matrix for observed variables, $\hat{\boldsymbol{\Sigma}}_{om} = \hat{\boldsymbol{\Sigma}}_{mo}^T$ the estimated covariance matrix between observed and missing variables, and $\hat{\boldsymbol{\Sigma}}_{mm}$ the estimated covariance matrix for the missing variables. After a fair amount of tedious algebra and integration, one can show that the expected value(s) for the missing part of $\boldsymbol{x}$ conditional on $\hat{\boldsymbol{\mu}}$, $\hat{\boldsymbol{\Sigma}}$, and $\boldsymbol{x}_o$ is given by

$$\hat{\boldsymbol{x}}_m = E(\boldsymbol{x}_m|\boldsymbol{x}_o,\hat{\boldsymbol{\mu}},\hat{\boldsymbol{\Sigma}}) = \hat{\boldsymbol{\mu}}_m + \hat{\boldsymbol{\Sigma}}_{mo}\hat{\boldsymbol{\Sigma}}_{oo}^{-1}(\boldsymbol{x}_o - \hat{\boldsymbol{\mu}}_o) \tag{10.12}$$

This equation can be used to impute a record once the parameters have been estimated (Procedure 10.6.2). The imputed dataset is obtained by applying Eq. (10.12) one last time on every record in the dataset.

```
impute_em(retl, ~ .- size) %>% head(3)
##    size     staff  turnover other.rev total.rev      vat
## 1   sc0 75.00000  893.2151  1168.523      1130 7917.634
## 2   sc3  9.00000 1607.0000  4373.896      1607 1749.444
## 3   sc3 11.84416 6886.0000   -33.000      6919 1991.730
```

Procedure 10.6.2 EM algorithm for multivariate normal distribution

```
Input: A set $\mathcal{X}$ containing $n$ numeric records $\boldsymbol{x}$.
Initialize: Choose an initial $\hat{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\Sigma}}$.
Iterate: Until convergence:
    E-step: Update $\mathcal{X}$ by setting
            $\boldsymbol{x} \leftarrow (\boldsymbol{x}_o, \hat{\boldsymbol{x}}_m)$for all $\boldsymbol{x} \in \mathcal{X}$,
        where $\hat{\boldsymbol{x}}_m$ is computed as defined in Eq. (10.12). Also compute
            $\boldsymbol{S}(\boldsymbol{x}) \leftarrow \hat{\boldsymbol{\Sigma}}_{mm} - \hat{\boldsymbol{\Sigma}}_{mo}\hat{\boldsymbol{\Sigma}}_{oo}^{-1}\hat{\boldsymbol{\Sigma}}_{om}$ for all $\boldsymbol{x} \in \mathcal{X}$.
        Next, compute the expected values of the sufficient statistics $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ as follows:
            $\boldsymbol{t}_1 \leftarrow \sum_{x \in \mathcal{X}} \boldsymbol{x}$
            $\boldsymbol{t}_2 \leftarrow \sum_{x \in \mathcal{X}} \boldsymbol{x} \otimes \boldsymbol{x}^T + \begin{pmatrix} 0_{o \times o} & 0_{o \times m} \\ 0_{m \times o} & \boldsymbol{S}(\boldsymbol{x}) \end{pmatrix}$.
    M-step: Compute the updated parameters
            $\hat{\boldsymbol{\mu}} \leftarrow \frac{1}{n}\boldsymbol{t}_1$
            $\hat{\boldsymbol{\Sigma}} \leftarrow \frac{1}{n}\boldsymbol{t}_2 - \hat{\boldsymbol{\mu}} \otimes \hat{\boldsymbol{\mu}}^T$.
Output: Estimates $\hat{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\Sigma}}$.
```

10.7 Sampling Variance under Imputation

The goal of data analysis is often to infer the value of a population parameter, say θ. If the data is obtained by sampling from the population, an estimate $\hat{\theta}$ is obtained by applying some procedure to the observed dataset. In such cases, one is often interested in how much the estimator $\hat{\theta}$ would vary if the sampling-and-estimation procedure was to be repeated on the same population. The variation over all possible samples is usually measured by the *sampling variance.*

Suppose that the parameter of interest is the population mean μ of a variable Y. Given a sample, obtained using simple random sample without replacement, an unbiased estimate $\hat{\mu}$ can be obtained by computing the sample mean. The sampling variance of $\hat{\mu}$, estimated from the same sample, is given by the well-known expression

$$\frac{1}{n}\left(1 - \frac{n}{N}\right)\frac{1}{n-1}\sum_{j=1}^{n}(y_j - \hat{\mu})^2, \tag{10.13}$$

where n and N are the sample and population size and y_j the observed sample values.

The important thing to realize here is that the expression for sampling variance depends on the sampling scheme (including size) and the expression or procedure used to obtain the estimator. This means that if part of the sampled data must be imputed, this imputation procedure should be considered part of the estimation procedure. This is easy to see from the abovementioned example. Suppose that some fraction of the observations y_j are missing completely at random, and we impute them with the estimated mean (ignoring missing values). This clearly leads to negatively biased variance estimation since the terms in the sum of Eq. (10.13) corresponding to missing y_j are zero by definition.

More generally, suppose that we are interested in some population parameter $\hat{\theta}$. Suppose further that estimation of θ includes some model-based imputation procedure depending on model parameters $\boldsymbol{\beta}$ that are estimated from the same sample as θ. Using the rule of conditional variances (sometimes called Eve's law), we can write

$$\mathrm{V}(\hat{\theta}) = \mathrm{E}_{\beta}\mathrm{V}(\hat{\theta}|\hat{\boldsymbol{\beta}}) + \mathrm{V}_{\beta}\mathrm{E}(\hat{\theta}|\hat{\boldsymbol{\beta}}). \tag{10.14}$$

If the sampling variance of $\hat{\boldsymbol{\beta}}$ is assumed to be zero or very small, the second term vanishes, and we get $\mathrm{V}(\hat{\theta}) = \mathrm{V}(\hat{\theta}|\hat{\boldsymbol{\beta}})$. The first term can therefore be interpreted as the sampling variance of $\hat{\theta}$ that is intrinsic to the sampling scheme, where the procedure to estimate θ includes imputation. The second term corresponds to variance added by uncertainty in the imputation model parameters.

The assumption that $\hat{\boldsymbol{\beta}}$ does not vary is valid in the case of imputations that do not vary with sample composition. This is the case, for example, when imputing with a fixed value or when using a deductive method such as those described in Section 9.3.2.Recall that in deductive methods one derives imputed values from conditions on the data and observed values that are deemed valid. It is also valid when the model used to impute the data was fitted on a dataset that is different from the imputed dataset (but note that this may bias the estimator since one assumes that model parameters are constant across datasets).

In many common cases the assumption of zero variation in imputation model parameters will be invalid, and hence the second term in Eq. (10.14) will be positive. In particular, any procedure where a dataset is imputed using a predictive model, which is subsequently ignored in variance estimation, will underestimate the sampling variance.

Over the past decades, two contrasting views of variance estimation over partially imputed datasets have developed. The first, historically, is due to Rubin who published and refined methodology for multiple imputation over several publications [see Rubin (1978); Rubin (1996); and the basic reference Rubin (1987)]. Rubin (1978) proposes to impute each missing data multiple times so as to 'reflect variation within a model as well as [...] due to a variety of reasonable models'. An analyst who is assumed to be not involved in the imputation process can get an idea of the variance due to imputation by computing parameters of interest over several copies of the imputed dataset.

The second view was proposed first by Rao and Shao (1992) in the context of hot deck imputation, but the idea applies more generally. In their view, one considers a population parameter θ, which is in the complete data case estimated by $\hat{\theta}$. As usual, the estimated quantity has an associated (estimated) sampling variance, denoted $\hat{\mathrm{V}}(\hat{\theta})$ (recall that the sampling variance is caused by variation of $\hat{\theta}$ over all possible same-sized samples from a population). In the case of missing data, the estimator $\hat{\theta}$ is replaced with $\hat{\theta}^*$, which is similar to $\hat{\theta}$ except that it is applied to the imputed dataset. Based on a resampling formalism (the jackknife), they are able to define and analytically approximate an estimator for the sampling variance $\hat{\mathrm{V}}(\hat{\theta}^*)$.

An account of both methodologies and arguments for and against these approaches is given in Rao (1996), Rubin (1996), and Fay (1996) and the ensuing comments by Judkins (1996), Binder (1996), and Eltinge (1996). In the following paragraphs, both methods are discussed in some detail.

10.8 Multiple Imputations

Multiple imputation is a computational method that allows for direct estimation of the two variance components of Eq. (10.14). A schematic of the main idea is shown in Figure 10.6. We start on the left-hand side with a (possibly multivariate) dataset $\boldsymbol{Y}$ of which some fraction of values are missing. This dataset is imputed M times, independently using an imputation model where both the model parameters and the imputed values are drawn from a distribution that is conditioned on the model's predictor variables. Each dataset is then used to estimate a population parameter θ using the same procedure one would use on $\boldsymbol{Y}$ if it were complete to begin with. This yields an ensemble of estimates $\hat{\theta}^{(1)}, \hat{\theta}^{(2)}, \dots, \hat{\theta}^{(M)}$. The final estimate is then computed as the average over the ensemble:

$$\overline{\theta} = \frac{1}{M} \sum_{j=1}^{M} \hat{\theta}^{(j)}. \tag{10.15}$$

Similarly, one can estimate the variance for each estimate $\hat{\theta}^{(j)}$ as one would for a complete sample. For example, if θ is the mean of a variable, and $\boldsymbol{Y}$ is obtained from simple random sampling without replacement, one could use Expression (10.13). This exercise then yields an ensemble of estimated variances $\hat{\mathrm{V}}(\hat{\theta}^{(j)}), \hat{\mathrm{V}}(\hat{\theta}^{(2)}), \dots, \hat{\mathrm{V}}(\hat{\theta}^{(M)})$. Averaging over this ensemble estimates the first term in Eq. (10.14), the expected value of the variance of $\hat{\theta}$. The second term of Eq. (10.14)—the variance of the expected value—is estimated as the variance over the estimates $\hat{\theta}^{(1)}, \hat{\theta}^{(2)}, \dots, \hat{\theta}^{(M)}$. Combining the two estimates, we get (Rubin, 1987)

$$\hat{\mathrm{V}}(\overline{\theta}) = \frac{1}{M} \sum_{j=1}^{M} \hat{\mathrm{V}}(\hat{\theta}^{(j)}) + \left(1 - \frac{1}{M}\right) \sum_{j=1}^{M} \frac{(\hat{\theta}^{(j)} - \overline{\theta})^2}{M-1}, \tag{10.16}$$

where the term $(1 - M^{-1})$ is a correction for the finite number of multiple imputations. In the context of multiple imputation, the first term is often referred to as *within-imputation variance* and the second term as the *between-imputation variance.*

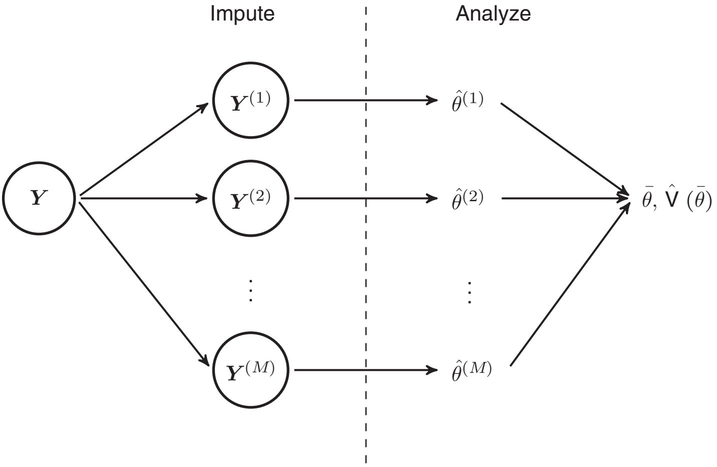


Figure 10.6 Estimation using multiple imputation.

An advantage of the multiple imputation method is that it separates the imputation problem from later analysis and estimation. In fact, one of the aims of its development was to provide a way for data providers to impute a dataset such that analysts can use it for any statistical inference without knowing the details of the imputation method. The trade-off is then that an analyst must repeat the analysis M times and appropriately combine the results.

This luxury of separating imputation from analysis does not come for free, however. As it turns out, it is not possible to choose the imputation method completely independent from the parameters being estimated. In essence, the imputation method must be such that the $\hat{\theta}^{(j)}$ and $\hat{\mathrm{V}}(\hat{\theta}^{(j)})$ (computed from imputed datasets) are unbiased estimates compared to the estimator one would use if $\boldsymbol{Y}$ was observed completely. Furthermore, the between-imputation variance of $\overline{\theta}$ must be an unbiased estimator of the variance caused by uncertainty in the model parameters (Rubin, 1987, 1996). Imputations that satisfy these demands are called *proper imputations* [for a proper technical definition, cf. Rubin (1987)]. The demand for proper imputations therefore implies that imputation is not fully decoupled from analysis and that one needs to carefully design the imputation methods.

A typical improper imputation method is imputation of the mean. Although this does yield an unbiased point estimate of the population mean, estimation of the sampling variance will be negatively biased. However, if the imputed variable is strongly correlated with one of the observed variables, this bias can be reduced with regression. The typical approach to multiple imputation is therefore an attempt to utilize as much as possible all relations between the variables so as to approximate the full multivariate distribution. Two such approaches will be discussed in the following sections: a bootstrapped version of the EM algorithm in Sections 10.8.1 and 10.8.2 and multivariate imputation by chained equations in Sections 10.8.3 and 10.8.4.

A natural question to ask is how large a value of M is necessary. A value often quoted in the literature is $M = 5$ when the fraction of missing information γ is a few percent at most. This recommendation is based on an analysis of the efficiency[1] of an estimator, which Rubin (1987) shows to be approximately equal to $(1 + \gamma/M)^{-1}$ relative to the $\gamma = 0$ case. Graham *et al.* (2007) performed extensive simulations to study the effect of M on the statistical power of an estimator rather than its efficiency. Based on the simulations where the parameter of interest concerned regression coefficients between a constructed pair of random normal variables, they conclude that drop-off in statistical power[2] is much faster than is to be suspected from the drop-off in efficiency as M decreases. They therefore recommend to use much higher numbers of imputations.

Table 10.1 reproduces the recommendations of Graham *et al.* (2007). In the case of $< 1\%$ power loss compared to $M = 100$, one would need to impute 20 times. A comparison with the full information maximum-likelihood model (FIML, which in their simulation is equivalent to the multivariate normal EM algorithm) is also made. The latter model may be considered a benchmark for their simulation when $M \to \infty$. When interpreting Table 10.1, one should keep in mind that it was produced based on artificial multivariate normal data. The number of necessary imputations may vary depending on

[1] The relative efficiency of two estimators is the ratio of their sampling variances.

[2] Statistical power is the probability that a false null hypothesis is rejected. For regression, this is the hypothesis that any or all regression coefficients equal zero.

Table 10.1 Number of multiple imputations *M*, as recommended by Graham *et al.* (2007).

Acceptable power falloff				
	Compared to $M = 100$			Compared to EM
γ	< 5%	< 3%	< 1%	< 1%
0.1	3	5	20	20
0.3	10	20	20	20
0.5	10	20	40	40
0.7	20	40	40	100
0.9	40	40	100	>100

γ is the fraction of missing information, which equals the fraction of missing data when variables are independent.

the shape of the (multivariate) distribution and (possibly nonlinear) relations between the variables. In practice, one will therefore often need to experiment and gain experience with values for M that are appropriate for a particular imputation problem.

10.8.1 Multiple Imputation Based on the EM Algorithm

Honaker *et al.* (2011) propose a bootstrapping scheme to randomize the parameters for a multivariate normal model distribution. The idea, summarized in Procedure 10.8.1, is to create an ensemble of M datasets by resampling from the original dataset with replacement, such that each dataset in the ensemble has the same number of records as the original. Next, the EM algorithm is used on each of the M datasets to find maximum-likelihood estimates of the distribution parameters (mean vector and covariance matrix). Conditional on the validity of the bootstrapping scheme, the resulting ensemble of multivariate normal parameters estimates their sampling distribution. For each dataset in the ensemble, each record can be completed by sampling from the multivariate normal distribution conditional on the observed values in the record.

Procedure 10.8.1 Bootstrapped Expectation–Maximization (EMB)

Input: Numeric dataset $\boldsymbol{Y}$ consisting of n records.
Bootstrap: Create an ensemble of M datasets where each dataset is generated by sampling n records from the input dataset without replacement.
Impute: For each dataset in the ensemble:

1) Use the EM algorithm of Procedure 10.6.2 to estimate the mean vector and the covariance matrix.
2) Complete each record by sampling a vector from the multivariate normal distribution parameterized by the mean and covariance just computed and conditioning on the observed values.

Output: An ensemble of imputed datasets $\{\boldsymbol{Y}^{(1)}, \boldsymbol{Y}^{(2)}, \dots, \boldsymbol{Y}^{(M)}\}$.

Regarding the ‘properness’ of this imputation method for estimating population parameters and their variances from the ensemble, one should realize that the imputation method is based on estimates of the means and covariance matrix. This means that only linear relationships between the variables and their variances can be trusted to be properly represented in the ensemble of imputed datasets. For example, suppose that we have a dataset containing realizations of variables X, Y, and Z, with some of them missing. After creating an ensemble of imputed datasets using the EMB algorithm, we can safely estimate β_0, β_1, and β_2 and their variances in the model $Y = \beta_0 + \beta_1 X + \beta_2 Z$. However, if the model was to be augmented with a term $\beta_{12} YZ$, the value and variance of this interaction effect can in principle not be trusted since the imputation model did not include it.

10.8.2 The `Amelia` Package

The `Amelia` package[3] of the same authors implements the Expectation Maximization with Bootstrapping (EMB) algorithm. The core function is `amelia`, which multiply imputes a multivariate dataset based on Procedure 10.8.1.

Here, we use two variables from the `retailers` dataset to demonstrate `amelia`.

```
data(retailers, package="validate")
dat <- retailers[c("staff","turnover")]
```

Visual inspection (histograms) of both *staff* and *turnover* reveals that they follow very skew distributions, but both distributions are more symmetric when viewed on a log scale. Even though the multivariate normal distribution is not a very good approximation of the transformed data, we will model the data as such. The `amelia` function has the `logs` option, allowing one to indicate which variables should be modeled on a logarithmic scale. Both transformation and back-transformation are then taken care of by `amelia`. To get an indication on the number of imputation needed, compute the fraction of missing data.

```
colSums(is.na(dat))/nrow(dat)
##      staff   turnover
## 0.10000000 0.06666667
```

We see that the maximum fraction of missing values is 0.1. Taking the smallest value for loss of statistical power in Table 10.1, this means we should take at least 20 imputations.

```
out <- Amelia::amelia(dat, m=20, logs=c("staff","turnover"), p2s=0)
```

Here, we also set `p2s=0` (`p2s` = print to screen) to prevent `amelia` from writing output to screen during the EM iterations.

The object `out` is a list (with class attribute `amelia`) containing the copies of the imputed dataset (here, 20 copies), the statistics resulting from each EM optimization, and some information on convergence of each EM run. Individual components can be accessed with the `$` operator. For example, `out$imputations` is a list of imputed datasets.

Figure 10.7 summarizes the object. Here, we plotted the original data and an estimate of its bivariate normal density on a log scale and added the randomly imputed values as

[3] Named after the famous aviator Amelia Earhart who went missing over the Pacific Ocean on June 2, 1937.

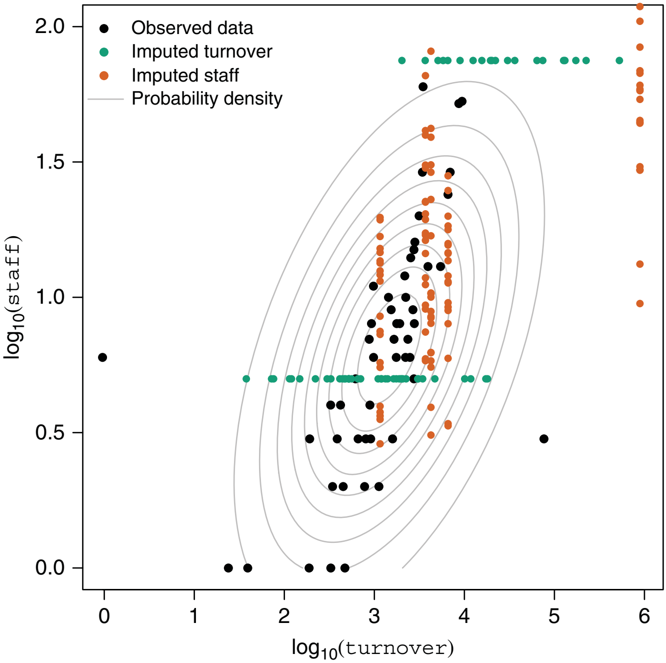


Figure 10.7 Observed and multiple imputed staff numbers and turnovers. The data is taken from the `retailers` dataset of the `validate` package. Imputations are generated with `Amelia::amelia`.

colored points. Observe that the original data (black points) contain some outliers that may influence the estimated probability density. The imputed turnover values (white points) are plotted on a horizontal lines since they are estimated conditional on a fixed observed value of staff (and vice versa for imputed staff numbers). Both in the case of *turnover* and *staff* some extrapolation outside of the observed data ranges was necessary to complete the records.

For cases with multiple variables, a plot such as Figure 10.7 cannot be created. `Amelia` includes a plot function that creates plots for each variable, comparing the distribution of the original data with that of the imputed data. To create such a plot (not shown here) simply pass an `amelia` object to plot.

```
plot(out)
```

With the function `overimpute`, one can create a per-variable comparison between values observed versus values estimated by the model. To create such a plot for the variable `staff` (see Figure 10.8) do the following.

```
overimpute(out, var="staff")
```

The graph shows observed versus predicted values for all observations of staff and their 90% confidence intervals. The color of the intervals codes the fraction of missing values in the record containing the plotted point.

In some cases, the likelihood function may have local maxima or be otherwise 'badly behaved'. Problems may surface, for example, when some variables are strongly colinear. The `disperse` function reruns EM optimization from several random starting points.

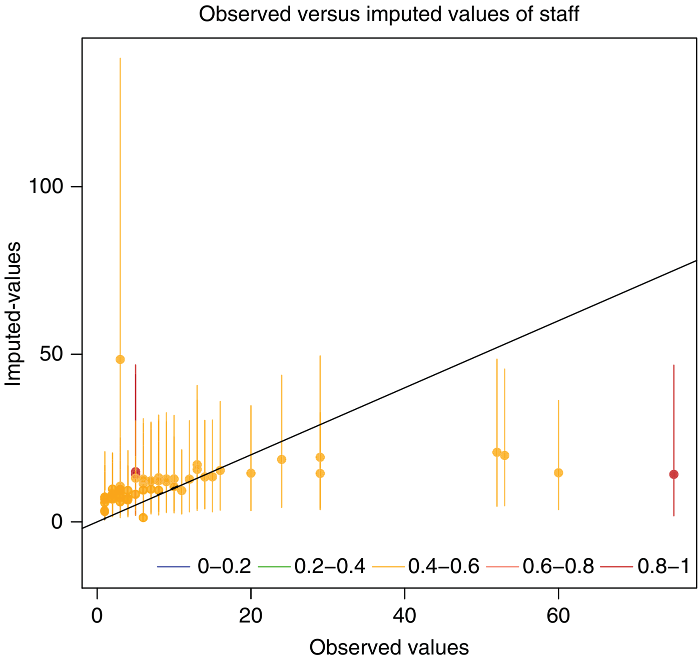


Figure 10.8 Result of a call to `Amelia::overimpute(out, var="staff")` vertical bars indicate 90% confidence intervals. A perfect model would have all points on the diagonal $y = x$ line. The colors indicate the fraction of missing values in the record to which the record pertains.

For stable solutions, one expects that each optimization ends in the same optimum. Using `disperse`, a graph of the optimization paths is created so that one can visually check whether all optimizations yield the same (or very close) result.

Analyzing the multiply imputed dataset is most conveniently done with the `Zelig` package. The `zelig` function accepts an object of class `amelia` and can run many different models. Below, we compute the coefficients of a linear least squares (`"ls"`) regression of *turnover* against `staff`.

```
mod <- Zelig::zelig(turnover ~ staff, model="ls", data=out, cite=FALSE)
```

The argument `cite=FALSE` suppresses printing of citation information when running the model. The output object is of class `zelig` and can be summarized like any model object in R.

```
# summary of model based on original data
summary( lm(turnover ~ staff, data=dat) )
##
## Call:
## lm(formula = turnover ~ staff, data = dat)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##  -3397  -2648  -1930  -1048  76827
##
```

```
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2969.53    2024.67   1.467    0.149
## staff          67.67     121.43   0.557    0.580
##
## Residual standard error: 11200 on 49 degrees of freedom
##   (9 observations deleted due to missingness)
## Multiple R-squared:  0.006298,  Adjusted R-squared:  -0.01398
## F-statistic: 0.3105 on 1 and 49 DF,  p-value: 0.5799
# summary of model based on multiple imputed data
summary(mod)
## Model: Combined Imputations
##
##              Estimate Std.Error z value Pr(>|z|)
## (Intercept)   -11795     26341   -0.45     0.65
## staff           2308      1775    1.30     0.19
##
## For results from individual imputed datasets, use
   summary(x, subset = i:j)
## Statistical Warning: The GIM test suggests this model
   is misspecified
##  (based on comparisons between classical and robust SE's;
   see http://j.mp/GIMtest).
##  We suggest you run diagnostics to ascertain the cause,
   respecify the model
##  and run it again.
##
## Next step: Use 'setx' method
```

Observe that the standard errors computed from the multiply imputed datasets exceed those from the original data. A full description of Zelig's capabilities is beyond the scope of this work, and the reader is kindly referred to Imai *et al.* (2008) for an introduction and overview.

Exercises for Section 10.8.2

Exercise 10.8.3 *The slot* out$imputations *(see page 249) contains the list of randomly imputed datasets.*

*a) Use Eq. (*10.15*) to estimate the mean* staff*.*
*b) Use Eq. (*10.13*) to estimate the variance of the mean* staff *for each imputed dataset. Average over these values to obtain the first term in Eq. (*10.16*).*
*c) Also compute the second term in Eq. (*10.16*) to obtain the full variance estimate.*

10.8.3 Multivariate Imputation with Chained Equations (Mice)

Imputation methods based on the EM algorithm, be it single or multiple imputation, depend on the ability to formulate a multivariate distribution for the treated data from which to sample imputations. As an alternative, one may formulate a separate probability model for each variable to be imputed. Model parameters and imputation values are generated sequentially and randomly conditional on known (possibly previously imputed) variables. In the so-called *fully conditional specification*, one allows every variable except the one that is currently imputed to be part of the predictive model. The sequence of imputations is repeated over until certain distributional properties have converged. Procedure 10.8.2 contains the synopsis of a single such sequence of imputations. Multiple imputation is then achieved by repeating this whole procedure M times to create M imputed datasets.

Procedure 10.8.2 Imputation with chained equations

```
Input: A dataset $\boldsymbol{Y}$ containing realizations of variables $Y_j$ where $j = 1, 2, \ldots k$. $\boldsymbol{Y}$ can be
  written as a combination of an observed part $\boldsymbol{Y}_o$ and missing part $\boldsymbol{Y}_m$. A set of $k$ para-
  metric model specifications with corresponding parameter vectors $\boldsymbol{\beta}_j, j = 1, 2, \ldots, k$.
Initialize: Create initially imputed dataset $\boldsymbol{Y}^* = (\boldsymbol{Y}_o, \boldsymbol{Y}^*_m)$ and initial parameter esti-
  mates $\boldsymbol{\beta}^*_j$.
Itererate: Until convergence do:
  For $j = 1 \ldots k$ do
    Sample $\boldsymbol{\beta}^*_j \sim P(\boldsymbol{\beta}_j | \boldsymbol{Y}_{o,j}, \boldsymbol{Y}^*_{-j}, \boldsymbol{\beta}^*_{-j})$
    Sample $\boldsymbol{Y}^*_{m,j} \sim P(Y_m | \boldsymbol{Y}^*_{-j}, \boldsymbol{\beta}^*_j)$
Output Imputed dataset $\boldsymbol{Y}^*$.
```

It was pointed out by van Buuren and Groothuis-Oudshoorn (2011) that the general approach of iterating over a sequence of probability models has been reinvented several times under different names. Authors reporting such approaches include Kennickell (1991), Brand (1999), Oudshoorn *et al.* (1999), Raghunathan *et al.* (2001), Heckerman *et al.* (2000), Rubin (2003), and Gelman (2004). Here, we follow the terminology of van Buuren and Groothuis-Oudshoorn (2011), who introduced the term *chained equations*. The term refers to the fact that in a sequence where variables are imputed randomly, one by one, and conditional on the previously imputed variables, this conditioning introduces a chain of dependencies on the probability distributions.

The procedures reported by various authors differ both in the chosen sequence of probability models and details of the initialization and iteration over imputations. In a linear model as in Brand (1999); Oudshoorn *et al.* (1999), one assumes

$$Y_j = \boldsymbol{X}_j \boldsymbol{\beta}_j + \epsilon, \quad \text{with } \epsilon \sim N(0, \sigma^2),$$

and $\boldsymbol{X}_j$ a design matrix derived from $\boldsymbol{Y}_{-j}$: all variables of the current dataset, except the imputed variable (this does not mean that all variables are necessarily part of the model: the predictors may differ between imputed variables). If $\boldsymbol{\beta}$ is determined by minimizing ϵ^2 (ordinary least squares), a classic result from regression theory shows that this implies

$$\hat{\boldsymbol{\beta}} - \boldsymbol{\beta} \sim N(0, \sigma^2(\boldsymbol{X}_j^T \boldsymbol{X}_j)^{-1}).$$

which then determines the distribution from which to sample $\boldsymbol{\beta}^*_j$ in Procedure 10.8.2. Kennickell (1991), Gelman (2004), and Raghunathan *et al.* (2001) discuss generalized regression models with explicit transformations of predictor and/or imputed variables, and for each model one needs to determine how to sample the parameters conditionally on the predictor variables. For example, the latter author gives explicit recipes for sampling coefficients and imputations in the case of numeric data (normal linear regression) count data (Poisson regression), binary or general categorical data [(generalized) logistic regression], and mixed data (a two-step procedure). One group differing significantly from (generalized) linear modeling is Heckerman *et al.* (2000), who use probabilistic decision trees.

The general chained equation approach is thus very flexible and even allows conflicting models to be specified (Arnold *et al.*, 1999; Rubin, 2003). For example, given two variables Y_1 and Y_2 and the models $Y_1 = \beta_1 Y_2^2 + \epsilon_1$ and $Y_2 = \beta_2 Y_1^2 + \epsilon_2$. Clearly, this model makes no sense structurally, and it is not possible for both ϵ_1 and ϵ_2 to be normally distributed (ϵ_1 and ϵ_2 are not connected by a linear transformation). A second way of stating this is that the two models, with ϵ_1 and ϵ_2 assumed to be normally distributed, do not permit an implicit joint distribution for Y_1 and Y_2.

10.8.4 Imputation with the `mice` Package

The `mice` package supports a number of model types for multiple imputation with chained equations. The imputation and analyses are set up to closely reflect the states of data in Figure 10.6. The package's core function is called `mice`. In this example we use the same data as for Section 10.8.2.

```
# create a 2-variable dataset
data(retailers, package="validate")
dat <- retailers[c("staff","turnover")]
```

Imputation using the default method (PMM, based on liner modeling) works as follows:

```
library(mice)
out <- mice::mice(dat, m=20, printFlag=FALSE)
```

We set `printFlag=FALSE` to avoid printing iteration info to the screen during computation, and as before (Section 10.8.2) we set the number of imputations `m=20`. The output, stored in `out`, is an object of class `mids`, which stands for 'multiple imputed datasets'. In fact, to save memory, only the imputed values are stored to be used when needed during analyses. The imputed values can be obtained by selecting the appropriate elements of `out$imp`. For example,

```
out$imp$staff[, 1:6]
##      1  2  3  4  5  6
## 3    6  1 13 75 52 60
## 4    5  6  5 24 53 53
## 5    3  3  6 52 13  1
## 14  52 29 53 75  3  3
## 40   1  1 60 52 24 24
## 43   6  1  6 29 52 52
```

selects the first six imputations for each missing value of *staff*. The row numbers indicate the record number in the original dataset.

Analyzing the imputed datasets can be done using `mice`'s version of `with`.

```
fits <- with(out, lm(staff ~ turnover))
```

The object `fits` contains the results of fitting $M = 20$ complete data linear models based on the imputed datasets. One should think of `fits` as resembling the ensemble of

$\hat{\theta}^{(j)}$ values in Figure 10.6. The class of fits is mira, which stands for 'multiply imputed repeated analyses'.

The final step is to combine (pool) the analyses to the final estimates using the pool function.

```
est <- pool(fits)
summary(est)
##                       est           se          t        df     Pr(>|t|)
## (Intercept) 1.251705e+01 2.562367e+00 4.8849567 32.380998 2.701511e-05
## turnover    2.285708e-05 3.985103e-05 0.5735632  7.613212 5.828042e-01
##                     lo 95        hi 95 nmis       fmi    lambda
## (Intercept)  7.300088e+00 1.773401e+01   NA 0.3488676 0.3098555
## turnover    -6.985859e-05 1.155728e-04    4 0.8499097 0.8150586
```

The object est is of class mipo, meaning 'multiply imputed pooled outcomes'. Its printed output resembles the output of an lm object, but note that its contents are different: pool gathers the data in mipo objects in a mira way that makes summarizing the statistics using summary easier. One can therefore not use residuals or predict to obtain residuals or predictions from the final estimated model. On the other hand, the output of summary applied to a mipo object is a simple R matrix, which can easily be processed further for predictive purposes with a bit of programming.

The mice package comes with some diagnostic tools as well. The first we have already seen: by inspecting the imputed values for staff, we can already see that the method used here gives quite a large variance. A visual comparison of the imputed and observed values can be created using stripplot(out) (not shown here—make sure to first load the lattice package). As an illustration, let us compute the ranges of imputed values per record for *staff* and compare it with the range in the original data.

```
range(dat$staff,na.rm=TRUE)
## [1]  1 75
apply(out$imp$staff, 1 , range)
##       3  4  5 14 40 43
## [1,]  1  1  1  1  1  1
## [2,] 75 75 75 75 60 53
```

For each record, the imputations, randomly selected conditional on the other variables in the dataset (*turnover*), the range is as large as the range across all records(!). The problem in this example is that the variables are better modeled on the log–log scale, as shown in Figure 10.7.

Unlike Amelia, mice has no built-in facilities to perform transformations on both predictor and predicted variables (there are facilities to transform the predictor variables using the so-called passive imputation). The solution in our case would be to transform the dataset, create the mids object, and to implement the back-transformation as part of the call to with when creating the mira object. The latter object can then be pooled normally.

Finally, we note that it is possible to obtain the imputed datasets using the complete function. For example, to obtain the complete dataset from the 15th imputation, do the following.

```
imp_15 <- complete(out, 15)
head(imp_15, 3)
```

```
##   staff turnover
## 1    75     3571
## 2     9     1607
## 3     1     6886
```

Unfortunately, the `Zelig` package (see Section 10.8.2) does not work directly with objects of class `mira` since the `zelig` function expects a list of imputed datasets. However, it is not difficult to create an appropriate object from a `mice` output.

```
M <- 20
# Complete the original data, 20 times
completed <- lapply(1:20, function(i) complete(out, i))
# pass the data frames to 'Zelig::mi'
completed_mi <- do.call(Zelig::mi, completed)
# use the created object as data for 'zelig'
Zelig::zelig(staff ~ turnover, data=completed_mi, model='ls',
   cite=FALSE)
## Model: Combined Imputations
##
##              Estimate Std.Error z value Pr(>|z|)
## (Intercept) 1.25e+01  2.56e+00    4.88    1e-06
## turnover    2.29e-05  3.99e-05    0.57     0.57
##
## For results from individual imputed datasets, use
   summary(x, subset = i:j)
## Next step: Use 'setx' method
```

Here, we pass the list of completed data frames to `Zelig::mi` to create an object of class `mi`. This object is accepted by `zelig` to perform multiple analysis.

A full tutorial on `mice` can be found in van Buuren and Groothuis-Oudshoorn (2011), and for a full tutorial on `zelig`, the reader is referred to Imai *et al.* (2008).

10.9 Analytic Approaches to Estimate Variance of Imputation

10.9.1 Imputation as Part of the Estimator

We assume that $\boldsymbol{y} = (y_1, y_2, \dots, y_n)$ is obtained by sampling without replacement from a population of size N, with the aim of estimating the population parameter $E(Y) = \mu$. Using the standard estimator of the mean $\hat{\mu} = \mathbf{1}^T\boldsymbol{y}/n$, the sampling variance can be estimated with Eq. (10.13)

A second way to estimate the sampling variance of an estimator is to compute the *jackknife variance*. This variance is computed by estimating a population parameter n times over the samples created by leaving out one observation at the time. It can be shown that the jackknife variance is given by (Rao, 1996)

$$\hat{\mathrm{V}}_J(\hat{\mu}) = \frac{n-1}{n}\left(1 - \frac{n}{N}\right)\sum_{j=1}^{n}[\hat{\mu}(j) - \hat{\mu}]^2,$$

where $\hat{\mu}(j) = (n\hat{\mu} - y_j)/(n-1)$ is the average over every observed value in $\boldsymbol{y}$ except for the jth element. A nice property of the jackknife estimator is that it extends to general estimators $\hat{\theta} = f(\hat{\mu})$, and in fact, if we choose $\hat{\theta} = \hat{\mu}$, we have $\hat{\mathrm{V}}_J(\hat{\mu}) = \hat{\mathrm{V}}(\hat{\mu})$.

In the case of missing values, it is not unusual to compute estimates based on the imputed dataset (so for the estimator of the mean, we have $\hat{\mu}_* = \mathbf{1}^T \boldsymbol{y}_*$). In the case of a general parameter, the jackknife variance is given by

$$\hat{V}_J(\hat{\theta}_*) = \frac{n-1}{n}\left(1 - \frac{n}{N}\right)\sum_{j=1}^{n} [\hat{\theta}_*(j) - \hat{\theta}_*]^2, \tag{10.17}$$

where $\hat{\theta}_*(j)$ is computed in the same way as $\hat{\theta}(j)$, except that all imputed values are replaced by the values that would be imputed when y_j is left out. Rao and Shao (1992) and Rao (1996) have derived analytical approximations of Eq. (10.17) when $\hat{\theta} = \hat{\mu}$ under random hot deck or regression imputation. They also show that for these cases $\hat{V}_J(\hat{\mu})$ is a consistent estimator. In the case of a general estimator $\hat{\theta}_*$, the jackknife variance is not very difficult to determine computationally by explicitly evaluating the sum in Eq. (10.17), possibly in parallel. Chen and Shao (2001) show that the jackknife estimator can overestimate the variance for nearest-neighbor hot deck imputation and propose improved nonparametric estimates.

10.10 Choosing an Imputation Method

The best method to use for a particular imputation problem generally depends on statistical as well as practical considerations. The latter may include factors such as the level of knowledge and experience of the statistician, the availability of a particular package, restrictions on allowed methodology in regulated environments, performance, and general requirements on the quality of the result. From a statistical point of view, the data type (numerical, categorical, and mixed), the observed data distribution, the missing data mechanism, and the relevance of maintaining distributional properties after imputation all play a role. In the following, some of the main differentiating characteristics of the methods described earlier will be summarized. An overview of the pros and cons of various approaches to imputation can also be found in the paper by Schafer and Graham (2002).

Regarding the chosen methodology, we distinguish the following four main characteristics to be determined:

1) parametric or nonparametric models
2) univariate or multivariate imputation
3) donor imputation or estimated imputation
4) single imputation or multiple imputation.

In Table 10.2, examples of imputation methods corresponding to these choices are given (where possible).

The advantage of parametric models over nonparametric models is their interpretability. The coefficients of, say, a linear model have meaningful units of measure, which means that their values can be assessed based on domain knowledge. In many cases, the statistical properties of the parameters (variance, confidence intervals, and p-values) are readily available. Do note, however, that these inferential properties always depend on the assumption that data is obtained by a proper randomization procedure. In the case of data coming from administrative databases or typical 'big

Table 10.2 A classification of imputation methods with examples.

Type	Example
pues	(robust) Regression imputation
puem	Multiple (robust) regression imputation
puds	(robust) Regression-based predictive mean matching
pudm	Multiple (robust) regression-based knn-predictive mean matching
pmes	Multivariate normal EM-based imputation
pmem	Multivariate normal EM-based multiple imputation (EMB)
pmds	Mice-pmm, avoiding the multiple imputations
pmdm	Mice-pmm
nues	CART-based imputation.
nuem	Multiple CART-based imputation
nuds	CART-based predictive mean matching
nudm	CART-based knn-predictive mean matching
nmes	Iterative random forest (missForest)
nmem	—
nmds	Iterative random forest-based pmm
nmdm	—

The first column indicates whether an imputation method is parametric or nonparametric, univariate or multivariate, estimation or donor-based, or single or multiple imputation. For example, the first row concerns parametric univariate estimation-based single imputation.

data' sources, such conditions are rarely met in practice. Parametric models allow for extrapolation beyond observed values of predictors, which may be an advantage if predictors are of high quality. It can be a nuisance, for example, when predictors contain outliers. The presence of influential outliers can be mitigated by applying robust parameterization, such as M-estimation or elasticnet. Nonparametric models tend to yield lower error of prediction within the range of observed values than parametric models, but the trade-off is that extrapolation beyond the observed predictor range is usually not meaningful. Also, nonparametric models such as random forest can to an extent model interactions between predictors automatically, while for parametric models, incorporating such effects requires manual configuration.

One advantage of multivariate imputation over univariate imputation is that the relations between all variables are modeled all at once. In the parametric case, the most common approach is to assume the multivariate normal distribution. This then models linear correlations between variables (no interactions). The missForest algorithm is a nonparametric counterpart that makes no distributional assumptions and moreover can handle nonnumeric data. Combinations of EM-based and nonparametric modeling can be devised as well. For example, Rahman and Islam (2011) propose to use a tree-based algorithm to split a dataset into highly correlated subsets and subsequently use the EM algorithm to impute each section. Applying multiple univariate imputations is attractive mainly because of its simplicity and the freedom of choosing a model for each variable separately. This may, however, lead implicitly to incompatible assumptions about the nature of the multivariate distribution (see also Section 10.8.3). Regarding the choice between parametric and nonparametric multivariate methods, it is worth noting that (Shah *et al.*, 2014) studied the performance of parametric MICE methodology with that of the missForest algorithm using simulated missing values on a medical dataset. Comparing some standard parameters such as bias and confidence intervals of estimated values, they found no large differences.

The main advantage of donor-based imputation over estimated imputation is that one can be sure that the imputed value is plausible, in the sense that it is something that has been observed. In the case of multivariate donor imputation, where multiple values are copied from a single record, this means that restrictions on the relations between those variables will be satisfied (we assume that only records that do not violate such restrictions will be used as donors). Multivariate restrictions concerning imputed values and values already present in the imputed record are still be violated, unless these are taken into account during donor selection. In comparison, predictive models usually do not take account of any restrictions on the data and therefore may (and often will) impute unacceptable values or value combinations that require further processing. A risk of donor-based imputation is that a certain group of donors is used so often that it significantly influences the (multivariate) distribution of the impute dataset. This risk is often mitigated by restricting the number of times a donor record may be used, although a study by Joenssen and Bankhofer (2012) shows that whether this improves the result really depends on the chosen hot deck method and data type. Given also the lack of foundational theory on hot deck methodology (Andridge and Little, 2010), some simulations are probably always necessary to fine-tune donor-based imputation methods.

10.11 Constraint Value Adjustment

Imputation methods in general cannot take account of logical or mathematical relations imposed on the data, although some approaches exist for specific cases (see de Waal (2017) for a recent overview). A viable and generic approach is therefore to choose a suitable imputation method and to adjust the imputed values afterward so that conditions can be met.

Here, we discuss a method that can be used to adjust numerical data under linear equality and/or inequality restrictions. The method has been discussed recently in relation with constrained imputation by Pannekoek and Zhang (2015). The underlying algorithm has probably been reinvented many times. The earliest reference known to these authors is that of Hildreth (1957). In the following, we first focus on the general method. After this, we point out some subtleties that arise in the context of data cleaning and imputation with examples, and we will finish with a practical example using the `rspa` package.

10.11.1 Formal Description

From a mathematical point of view, the problem is the following. Given a set of equality and inequality constraints represented by a system of equations and in equations:

$$\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}. \tag{10.18}$$

Suppose that we are presented with a vector $\boldsymbol{x}_0$ not satisfying these restrictions. In our applications, these will be values that have been imputed into a numerical record. We wish to move away from $\boldsymbol{x}_0$ minimally, so that we obtain a new vector, say $\boldsymbol{x}'$, such that $\boldsymbol{x}'$ does satisfy the restrictions. Now the term 'minimally' needs to be interpreted by defining a distance function. A generic choice is to use the (weighted) Euclidean distance, which yields the following minimization problem.

$$\boldsymbol{x}' = \underset{\boldsymbol{x}\in\mathbb{R}^n}{\text{argmin}}\ (\boldsymbol{x}_0 - \boldsymbol{x})^T \boldsymbol{W}(\boldsymbol{x}_0 - \boldsymbol{x}) \text{ s.t. } \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b},$$

where $\boldsymbol{W}$ is by definition a diagonal matrix with positive diagonal elements. A special case occurs when all the restrictions are equalities such as those arising from accounting balance restrictions. In that case, one can apply the Lagrange multiplier method. That is, one defines the function

$$L(\boldsymbol{x}, \lambda) = \frac{1}{2}(\boldsymbol{x}_0 - \boldsymbol{x})^T \boldsymbol{W}(\boldsymbol{x}_0 - \boldsymbol{x}) + \lambda^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})$$

where λ is a vector of dual variables (Lagrange multipliers). The solution (if any exists) is obtained by equating derivatives of L with respect to the components of $\boldsymbol{x}$ and λ to zero and solving for $\boldsymbol{x}$. The solution to this problem is

$$\boldsymbol{x} = \boldsymbol{x_0} - \boldsymbol{W}^{-1}\boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{W}^{-1}\boldsymbol{A}^T)^{-1}(\boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{b}).$$

When the set of restrictions also contains inequality restrictions, such as nonnegativity demands on certain variables, the Lagrange multiplier method no longer applies. The algorithm proposed by Hildreth (1957) exploits the fact that although satisfying all restrictions at once is difficult, it is not so difficult to find a solution that satisfies a single restriction. So the idea is to solve for one restriction at a time, iterating (possibly multiple times) over the restrictions until a satisfactory solution is found. A synopsis is given in Procedure 10.11.1, but it is most easily understood using a simple example.

Procedure 10.11.1 Successive projection algorithm

Input: A numerical vector $\boldsymbol{x}_0$, and a set of K (in)equality restrictions represented by $\boldsymbol{A}$, $\boldsymbol{b}$ and a set of K operators in $\{=, \leq\}$. A convergence criterion $\varepsilon > 0$.

Initialize. Create a vector $\boldsymbol{z} = \boldsymbol{0}$, where the dimension of $\boldsymbol{z}$ is equal to the number of restrictions.

Until convergence do, for $k = 1, 2, \ldots, K$:

```
$\lambda \leftarrow \dfrac{b_k - \boldsymbol{a}_k^T\boldsymbol{x}}{\boldsymbol{a}_k^T\boldsymbol{W}\boldsymbol{a}_k}$
$\delta \leftarrow \min(z_k, \lambda_k)$
$\boldsymbol{x} \leftarrow \boldsymbol{x} + \delta\boldsymbol{W}^{-1}\boldsymbol{a}_k$
$z_k \leftarrow z_k - \delta.$
```

Output A vector $\boldsymbol{x} = \boldsymbol{z}^k$ satisfying the constraints to within a distance $\|\boldsymbol{z}\|$.

Let us consider a two-vector $\boldsymbol{x}_0 = (0.8, 0.2)$, subject to the constraints

$$x_1 \geq 1 - x_2, \tag{10.19}$$

$$x_2 \geq x_1. \tag{10.20}$$

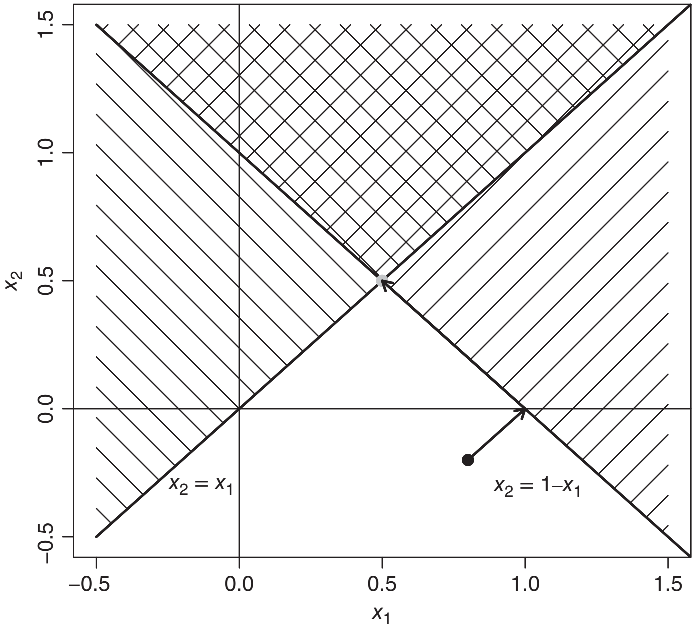


Figure 10.9 Valid regions (shaded) for restrictions (10.20) and (10.20). The black dots represents $\boldsymbol{x}_0 = (0.8, -0.2)$.

In Figure 10.9, the valid regions for each of these restrictions are shown. To interpret this figure, first note that the equalities $x_2 = x_1$ and $x_1 = 1 - x_2$ define lines in the $x_1 \times x_2$ plane that border on the valid regions. The valid regions are half-planes shown as shaded areas. The region that satisfies both constraints is the overlap of both regions, which in Figure 10.9 is doubly shaded.

In matrix notation, the restrictions are defined as

$$\begin{pmatrix} -1 & -1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leq \begin{pmatrix} -1 \\ 0 \end{pmatrix}.$$

Here, we have $\boldsymbol{W} = \mathbb{I}$. We can verify that for $\boldsymbol{x}_0 = (0.8, -0.2)$, the restrictions do not hold. Indeed, we have

$$\begin{pmatrix} -1 & -1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0.8 \\ -0.2 \end{pmatrix} = \begin{pmatrix} 1.0 \\ 0.0 \end{pmatrix} \nleq \begin{pmatrix} -1 \\ 0 \end{pmatrix}.$$

The idea of the Successive Projection Algorithm is to iteratively project $\boldsymbol{x}_0$ onto the borders. In Figure 10.9 this is indicated with arrows. First, $\boldsymbol{x}_0$ is updated by projection on the line $x_2 = x_1 - 1$. The co-ordinates of the new point are computed by following Procedure 10.11.1. We get:

$$\begin{aligned}
\lambda &= \frac{-1 - (-0.6)}{2} = -0.2 \\
\delta &= \min\{0, \lambda\} = -0.2 \\
\boldsymbol{x} &= \begin{pmatrix} 0.8 \\ -0.2 \end{pmatrix} - 0.2 \begin{pmatrix} -1 \\ -1 \end{pmatrix} = \begin{pmatrix} 1.0 \\ 0.0 \end{pmatrix} \\
z_1 &= 0 - (-0.2) = 0.2.
\end{aligned}$$

This step is depicted as the arrow from $\boldsymbol{x}_0$ to the line $x_2 = 1 - x_1$ in Figure 10.9. One can check that the first restriction is now satisfied by filling in the conditions in matrix notation. In the second step, we project onto the line $x_2 = x_1$. This yields the following:

$$\begin{aligned}
\lambda &= \frac{0 - (1, -1) \cdot (1, 0)}{2} = -\frac{1}{2},\\
\delta &= \min\{0, \lambda\} = -\frac{1}{2},\\
\boldsymbol{x} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \frac{1}{2}\begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 1/2 \\ 1/2 \end{pmatrix},\\
z_2 &= 0 - \left(-\frac{1}{2}\right) = \frac{1}{2}.
\end{aligned}$$

This vector satisfies both restrictions, so we are done.

Finally, let us summarize the procedure. In the first step, the signed distance λ between the current value of $\boldsymbol{x}$ and the border of the half-space defined by the kth constraint is computed. We then distinguish between three following cases. In the first case, the current value of $\boldsymbol{x}$ violates the current constraint. In that case $\lambda < 0$ and the updated vector will be the projection onto the half-space defined by the current restriction. In the second case, $\boldsymbol{x}$ already satisfies the current constraint. By comparing the current distance with the accumulated (nonnegative) distance z_k already traveled in the direction of this half-space, it is checked whether we can reduce the distance traveled (hence, the computation of δ) without violating the restriction. This is the case when $\lambda < z_k$, and the result is a projection of $\boldsymbol{x}$ on the border of the half-space from within the valid region. When $\lambda > z_k$, we get $\delta = z_k$, and the accumulated distance traveled will be reduced to zero.

10.11.2 Application to Imputed Data

When applying the successive projection algorithm to imputed values, there are a few subtleties to keep in mind. In general, the pattern of missing values varies from record to record. This means that for each record a new set of constraints is derived by substituting the observed (nonimputed) values into the overall set of constraints. This then yields the constraints that guide adjustment of the imputed values. However, the successive projection algorithm only converges when an actual solution exists. That is, one must ensure that the variables that have been imputed actually permit a solution. This can be done by preceding the imputation procedure with an appropriate error localization step (see Section 7.1).

The second subtlety is related to the chosen weights. Different variables often vary on different scales. Choosing a simple Euclidean distance between the original and adjusted value sets is likely to disturb ratios between the variables. This might render the resulting adjusted values implausible from the domain knowledge perspective (Pannekoek and Zhang, 2015). They show that when the weights are chosen as

$$W_{jj} = \frac{1}{x_j},$$

the adjusted values will in the first order preserve the ratios observed in the initial values.

10.11.3 Adjusting Imputed Values with the rspa Package

The successive projection has been implemented fundamentally by the lintools package. A convenient wrapper that allows for adjusting values stored in a data.frame is available through the rspa package, which we will use in the following example.

We will continue using the retailers dataset from the validate package and also use this package to constrain their values. The simputation package is used to find imputations.

```
library(validate)
library(simputation)
library(errorlocate)
library(rspa)
data(retailers)
```

We define a number of balance restrictions and nonnegativity rules for variables in the retailers dataset (see also Chapter 6).

```
v <- validator(
  staff >= 0
  , turnover >= 0
  , other.rev >= 0
  , turnover + other.rev == total.rev
  , total.rev - total.costs == profit
  , total.costs >= 0
)
```

To ensure that we are able to get values that satisfy all rules, we perform error localization and set all erroneous values to NA with the replace_errors function of the errorlocate package.

```
d1 <- replace_errors(retailers,v)
miss <- is.na(d1)
```

A logical matrix that signals what values are to be imputed (and thus later adjusted) is stored as well. We use the missForest algorithm to every missing value, not taking into account the validation rules.

```
d2 <- impute_mf(d1, . ~ .)
##   missForest iteration 1 in progress...done!
##   missForest iteration 2 in progress...done!
##   missForest iteration 3 in progress...done!
##   missForest iteration 4 in progress...done!
sum(is.na(d2))
## [1] 0
```

Finally, we use the rspa function to adjust the imputed values. Here, we use the simple Euclidean distance, but it is possible to pass a weights argument defining a weight vector for each record.

```
d3 <- match_restrictions(d2, v, adjust=is.na(d1))
```

Let us inspect the results. We allow for an error of 10^{-2} since that is the default convergence parameter defined by `rspa`. We use the `compare` function of the `validate` package to summarize the changes with respect to rule violations in consecutive versions of the data.

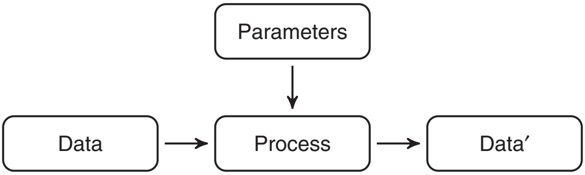
```
# set sensitivity to violations of linear (in)equalities to
# less than 1e-2
voptions(x=v, lin.eq.eps=1e-2, lin.ineq.eps=1e-2)

compare(v, start=retailers,locate=d1, imputed=d2, adjusted=d3)
## Object of class validatorComparison:
##
##                      Version
## Status                start locate imputed adjusted
##   validations           360    360     360      360
##   verifiable            265    238     360      360
##   unverifiable           95    122       0        0
##   still_unverifiable     95     95       0        0
##   new_unverifiable        0     27       0        0
##   satisfied             246    238     298      360
##   still_satisfied       246    238     246      246
##   new_satisfied           0      0      52      114
##   violated               19      0      62        0
##   still_violated         19      0      18        0
##   new_violated            0      0      44        0
```

The compare function shows that after setting the erroneous values to missing, no violations occur anymore, while the number of unverifiable checks increased with 27. These correspond to rules that could not be checked because of extra missing values in the data. After imputation, all 360 checks (60 records times 6 record-wise restrictions) can be verified. However, 62 of them are violated. These violations disappear (to within 10^{-2}) after imputed values are adjusted to match the restrictions.

11

Example: A Small Data-Cleaning System

In this chapter the pieces of software described in the previous chapters are combined into a small example data-cleaning system. Our focus will mainly be on illustrating how to set up an automated data-cleaning script and not so much on the perfect statistical solution for this problem, although some reasonable choices will be made. The idea is to have the main data-cleaning process flow separated as much as possible from the specification of the demands, as sketched in the following diagram.



In this general picture, the 'parameters' can be validation rules, modifying rules, or other parameters that control the process step manipulating the data. By separating the parameters from the main process flow, a degree of configurability can be achieved that does not depend on knowledge of the systems' internals. The parameter sets can be thought of as forming an application program interface (API), and in principle, one could program against it, for example, to build a graphical user interface. Furthermore, by setting up each step so that the main input and output are data stored in a similar type, the different processing steps can be easily connected to form a chain.

In R, this concept is implemented by the `magrittr` chain operator `%>%`. A simple chain, using functions of the `dplyr` package, looks as follows:

```
library(dplyr)
library(magrittr)
data(iris)
iris %>%
  filter(Species=="setosa") %>%
  select(Sepal.Width) %>%
  head(3)
##   Sepal.Width
## 1         3.5
## 2         3.0
## 3         3.2
```

Here, the arguments of filter and select are parameters steering the process of filtering and selecting. Similarly, the argument 3 of head determines how many rows are printed to the output.

For our purpose we want to go one step further and extract parameters steering the process to external configuration files as much as possible. Moreover, to monitor the effect and contribution of each data-cleaning step to the overall quality of the data, we wish to derive some kind of logging information to track changes in the data as it flows through the process.

In the following sections, we use the retailers dataset from the validate package to illustrate these ideas. After setting up the basic infrastructure, we set up a chain of processing steps and point out a few technicalities related to the order of processing. Finally, we add logging functionality and demonstrate how the data is altered throughout the process.

11.1 Setup

The basic goal of cleaning the retailers dataset is to fill in all missing values and make sure that the data satisfy all data validation rules. The workflow is as follows:

1) Read in the data.
2) Read in the validation rules.
3) Clean up the data, reading in parameters as necessary.
4) Write the cleaned up data back to disk.

Figure 11.1 shows the separate file with our validation rules. A first version of our data-cleaning script looks as follows:

```
# load necesarry libraries
library(validate)

# Read data
data(retailers, package="validate")

# Read the rules
rules <- validator(.file="rules.txt")

## Start cleaning!

# implement data cleaning operations

## Done?

# write data
write.csv(retailers, file="retailers_clean.csv",row.names=FALSE)
```

The script just reads data and rules and writes out the data again: the data-cleaning steps are to be implemented.

11.1.1 Deterministic Methods

The first thing we do is to apply some domain knowledge. That is, we apply externally stored data-modifying rules to clean up some obvious errors. Figure 11.2 shows a number of modifying rules that represent some paste experience on working with such

```
# nonnegativity rules
staff >= 0
turnover >= 0
other.rev >= 0
total.costs >= 0

# balance checks
turnover + other.rev == total.rev
turnover - total.costs == profit

# linearized version of
# if ( staff > 0 ) staff.costs > 0
# with minimal staff costs of
# 1k/staff member
staff.costs >= staff
```

Figure 11.1 The validation rules in `rules.txt`.

datasets. Since similar rules have to be applied to several variables, variable groups are used to condense the code. Of course in practice, these rules should be tried and tested to see whether they hold generally enough to apply them to the whole dataset. In this example we shall assume that this has been done. With the `dcmodify` package these rules can be applied to the data.

Besides the modifiers, we can attempt to find typographic errors in the data based on the relations forced by the validation rules. We also load the `deductive` package to apply `correct_typos`. Our script now looks like this:

```
# load necesarry libraries
library(validate)
library(dcmodify)
library(deductive)

# Read data
data(retailers, package="validate")

# Read validation rules
rules <- validator(.file="rules.txt")

## Start cleaning!

# Read modifying rules
mods <- modifier(.file="modify.txt")

# apply modifiers
retailers <- modify(retailers, mods)

# correct typographic errors
retailers <- correct_typos(retailers, rules)

## Done?

# write data
write.csv(retailers, file="retailers_clean.csv",row.names=FALSE)
```

```
# Cost variables are sometimes improperly reported
# as negatives.
#
if ( staff.costs < 0 ) staff.costs <- -staff.costs
if ( total.costs < 0 ) total.costs <- -total.costs

# Amounts of money are often reported in currency units rather
# than thousands.The below function detects unit-of-measure errors
#
is_ume := function(x,y){
  ratio <- x/y
  limit1 <- median(ratio,na.rm=TRUE) + IQR(ratio,na.rm=TRUE)
  limit2 <- 10*median(x,na.rm=TRUE)
  (ratio > limit1) | (is.na(y) & x > limit2)
}

if ( is_ume(turnover, staff) ){
  turnover <- turnover/1000
}

if ( is_ume(other.rev, staff) ){
  other.rev <- other.rev/1000
}

if ( is_ume(total.rev, staff) ){
  total.rev <- total.rev/1000
}

if ( is_ume(staff.costs,staff) ){
  staff.costs <- staff.costs/1000
}

if ( is_ume(total.costs,staff) ){
  total.costs <- total.costs/1000
}

if (is_ume(profit, staff)){
  profit <- profit/1000
}

# total.rev is sometimes submitted in the 'other.rev' field
#
if ( is.na(total.rev) | total.rev == 0 & turnover == other.rev ){
  total.rev <- other.rev
  other.rev <- 0
}
```

Figure 11.2 Modifying rules in `modify.txt`.

11.1.2 Error Localization

After applying our knowledge rules and attempting to use the information in the dataset to detect typos, we are out of our wits and resort to the paradigm of Fellegi and Holt. That is, we will attempt to find for each record the least number of fields that can be altered so that all validation rules can be satisfied. We use the `errorlocate` package to clear these subsets of fields. For the moment, we use no weights to distinguish the reliability of variables.

```
# library calls
library(validate)
library(dcmodify)
library(deductive)
library(errorlocate)

# Read data
data(retailers, package="validate")

# Read validation rules
rules <- validator(.file="rules.txt")

## Start cleaning!

# Read modifying rules
mods <- modifier(.file="modify.txt")

# apply modifiers
retailers <- modify(retailers, mods, sequential=TRUE)

# correct typographic errors
retailers <- correct_typos(retailers, rules)

# remove sufficient fields to fix the data
retailers <- replace_errors(retailers, rules)

## Done?

# write data
write.csv(retailers, file="retailers_clean.csv",row.names=FALSE)
```

11.1.3 Imputation

Our data is now at the point where observed values are deemed correct, and the missing data patterns are such that they can be imputed while satisfying every rule. Our imputation methodology will consist of two parts. First, where possible, unique imputation values will be derived based on the present values and the linear validation rules. Second, the remaining missing values will be imputed using a model-based imputation scheme. To set up model-based imputation, we first have a look at the data we have after running the error localization. Figure 11.3 displays a matrix plot of all variables occurring in the validation rules. From this overview we see that there is a reasonable linear relation between the variables, although some outliers seem still to be present. For the moment,

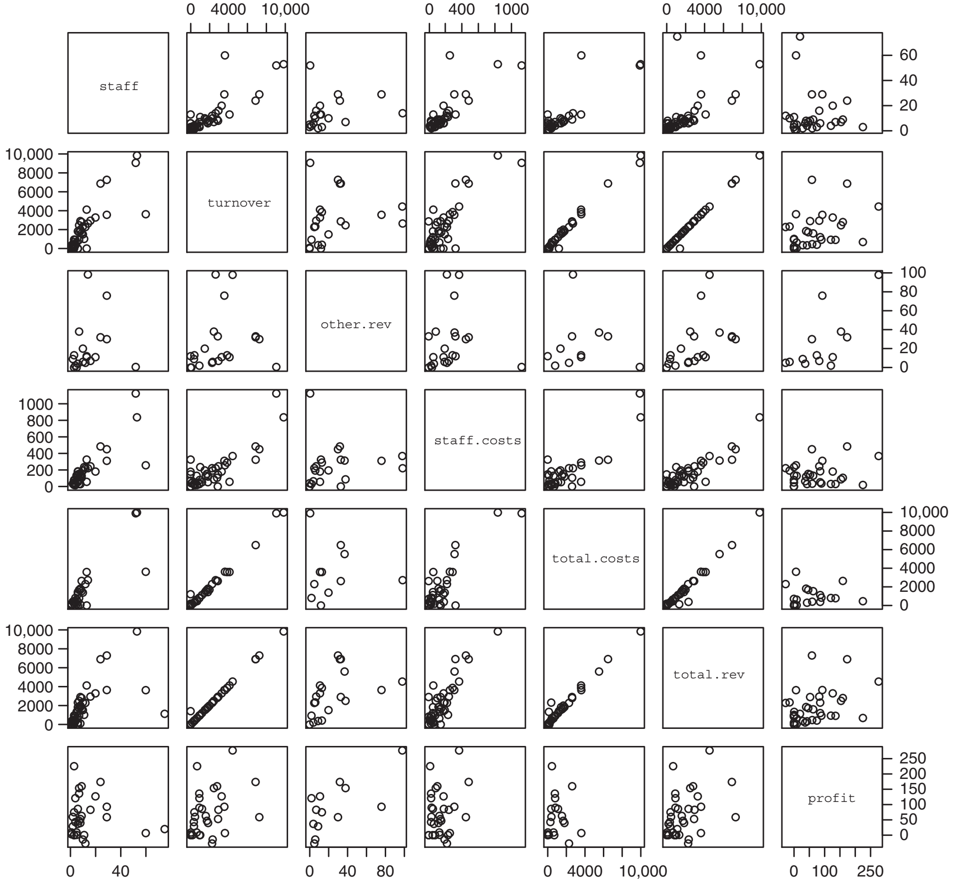


Figure 11.3 The retailers data after error localization.

we leave them in and use a sequence of robust methods to generate imputations. The variables *total revenue* and *staff* have the lowest missing value rates (13% and 15%), so those will be used as initial predictors. To be precise, we follow the following imputation scheme:

1) Use deductive imputation where possible on all variables.
2) Impute all variables except *staff* and *total revenue* by robust regression on predictors *staff* and *total revenue*.
3) Where *staff* is missing, regress only on *total.revenue*.
4) Use the missForest algorithm to impute the remaining missing values (in *total .revenue*).

We expect this will provide a reasonable first go at imputing the dataset.

Using the `simputation` package for model-based imputation, and `deductive` for deductive imputation, our script now looks as follows:

```
# load necesarry libraries
library(validate)
library(dcmodify)
library(deductive)
library(errorlocate)
library(simputation)

# Read data
data(retailers, package="validate")

# Read validation rules
rules <- validator(.file="rules.txt")

## Start cleaning!

# Read modifying rules
mods <- modifier(.file="modify.txt")

# apply modifiers
retailers <- modify(retailers, mods, sequential=TRUE)

# correct typographic errors
retailers <- correct_typos(retailers, rules)

# remove sufficient fields to fix the data
retailers <- replace_errors(retailers, rules)

# deductive imputation
retailers <- impute_lr(retailers, rules)

# impute by robust regression on staff + total.rev
retailers <- impute_rlm(retailers
  , turnover + other.rev + staff.costs +
    total.costs + profit ~ staff + total.rev)

# impute by robust regression on total.rev
retailers <- impute_rlm(retailers
  , turnover + other.rev + staff.costs +
    total.costs + profit ~ total.rev)

# impute using the missForest algorithm
retailers <- impute_mf(retailers, . ~ .)

## Done?

# write data
write.csv(retailers, file="retailers_clean.csv",row.names=FALSE)
```

11.1.4 Adjusting Imputed Data

All imputations were executed without taking the validation rules into account. In this final step, we will adjust imputed values using the successive projection algorithm as implemented by the rspa package. To do so, we need to record which values have been imputed so that `rspa::match_restrictions` knows what values to adjust. Note that the deductively imputed values need not be adjusted since these by definition are the unique solutions derived from observed data and validation rules. This gives the following completed script:

```
# load necesarry libraries
library(validate)
library(dcmodify)
library(deductive)
library(errorlocate)
library(simputation)
library(rspa)

# Read data
data(retailers, package="validate")

# Read validation rules
rules <- validator(.file="rules.txt")

## Start cleaning!

# Read modifying rules
mods <- modifier(.file="modify.txt")

# apply modifiers
retailers <- modify(retailers, mods, sequential=TRUE)

# correct typographic errors
retailers <- correct_typos(retailers, rules)

# remove sufficient fields to fix the data
retailers <- replace_errors(retailers, rules)

# deductive imputation
retailers <- impute_lr(retailers, rules)

# record missing values for match_restrictions
miss <- is.na(retailers)

# impute by robust regression on staff + total.rev
retailers <- impute_rlm(retailers
  , turnover + other.rev + staff.costs +
    total.costs + profit ~ staff + total.rev)

# impute by robust regression on total.rev
retailers <- impute_rlm(retailers
  , turnover + other.rev + staff.costs +
    total.costs + profit ~ total.rev)

# impute using the missForest algorithm
retailers <- impute_mf(retailers, . ~ .)
```

```
# adjust imputed values to match restrictions
retailers <- match_restrictions(retailers, rules, adjust=miss)

## Done!

# write data
write.csv(retailers, file="retailers_clean.csv",row.names=FALSE)
```

We can check whether the resulting data satisfies all rules after running this script. We test to an accuracy of 10^{-2} since that is the default value used in match_restrictions.

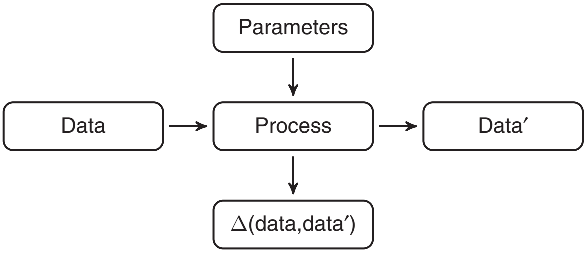
```
library(validate)
dat <- read.csv("R/retailers_clean.csv")
rules <- validator(.file="R/rules.txt")
voptions(rules, lin.eq.eps=0.01, lin.ineq.eps=0.01)

confront(dat, rules)
## Object of class 'validation'
## Call:
##     confront(x = dat, dat = rules)
##
## Confrontations: 8
## With fails    : 0
## Warnings      : 0
## Errors        : 0
sum(is.na(dat))
## [1] 0
```

So, the data is fully imputed and free of errors.

11.2 Monitoring Changes in Data

Ultimately, the data-cleaning steps used in the script rely on statistical assumptions about the data. For example, the function for detecting unit-of-measure errors assumes certain distributional properties, and the imputation models assume correlations between variables. For quality control it is helpful to have an overview of the influence that each step has on the final result. That way the risk associated with a misspecification of a modifying rule, imputation model, or localization weights can be assessed.

Schematically, we would like to extend the generic process step of the schema on page 265 to something that looks like this:



Here, Δ is a function that reports the difference between the input and the output of the process. Depending on the purpose of monitoring, there are several choices for Δ. One simple way of implementing this is to create a dump of the dataset after reading and every processing step afterward. However, this type of logging consumes a lot of storage with lots of redundant information. Also, further processing is necessary afterward to interpret the results. In the following sections we discuss a few methods that summarize changes in data in more interesting ways. We also demonstrate how to do them in R. In Section 11.2.4, we return to our example data-cleaning script and show how logging changes in data can be automated over multiple processing steps with the `lumberjack` package.

11.2.1 Data Diff (Daff)

Daff (Fitzpatrick *et al.*, 2017) is a technical standard for expressing the difference between two tabular datasets in textual format. Its name and purpose is inspired by *diff*: a standard utility in the POSIX family that compares two text files. A call to `diff`, which is available by default on many Unix-like operating systems, may look like this:

```
diff <file1> <file2>
```

The output of this call is a *patch* that can be used to update `<file1>` to `<file2>`. There are other uses: it can also be used to highlight differences when comparing files side by side, for instance.

The daff format describes the difference between tabular data in a way that is in itself tabular. Thus, daff patches are commonly stored as csv files. The format includes standard notation to express row insertion or deletion, column insertion, deletion, or renaming, and modified cell values.

To illustrate the syntax, we will show a few examples using the `daff` R package. For example,

```
data(retailers, package="validate")
library(simputation)
library(daff)

retailers2 <- impute_lm(retailers, turnover ~ total.rev)

# compute diff ('retailers' is the reference set)
d <- diff_data(retailers, retailers2)
# get the diff as a data.frame
d$get_data()
##    @@ ... staff      turnover other.rev total.rev staff.costs
## 1  -> ...    75 NA->1113.3116        NA      1130          NA
## 2     ...     9          1607        NA      1607         131
## 3 ... ...   ...           ...       ...       ...         ...
## 4     ...    NA          3861        13      3874         290
## 5  -> ...    NA NA->5585.3515        37      5602         314
## 6     ...     1            25        NA        25          NA
## 7  -> ...     5 NA->1318.3134        NA      1335         135
## 8     ...     3           404        13       417          NA
## 9 ... ...   ...           ...       ...       ...         ...
```

The first row and column contain metadata. The first row denotes the column names for columns that are of importance in this diff. The first row is the action column, denoting what activity is described in the row. The first row is marked with @@ in the action column: this signals that this row contains the column names. The second row is marked with ->. This signals that a cell value has changed. Here, a missing turnover value was changed to 1113.3116. Three dots (...) signal that rows or columns have been omitted. If the action column is empty, that means that nothing changed. The extra rows and columns are there to provide enough context for the patch program to find where to apply patches.

As a second example, consider renaming a column.

```
retailers3 <- dplyr::rename(retailers2, income = turnover)
d1 <- diff_data(retailers, retailers3)
head(d1$get_data())
##     ! ...            +++      ---
## 1 @@ ... vat    income turnover
## 2  + ...  NA 1113.3116       NA
## 3  + ...  NA      1607     1607
## 4  + ...  NA      6886     6886
## 5  + ...  NA      3861     3861
## 6  + ...  NA 5585.3515       NA
```

We now get an extra row, marked with a !, which indicates that the set of columns (scheme) has changed. The +++ in the fourth column signals the arrival of a new column, and the --- signals the deletion of one. Note also that the values of income and turnover differ in the first row, but this is not recognized as a separate cell change. Notations of row deletions and insertions are notated along similar lines. For a full and up-to-date specification, we refer the reader to the specification website mentioned in the reference.

Using the patch_data function we can now reconstruct the changes made while going from retailers → retailers2.

```
retailers3_reconstructed <- patch_data(retailers, d)
all.equal(retailers3_reconstructed, retailers3)
## [1] "Names: 1 string mismatch"
```

Although the daff format is human-readable, there are some limitations to its interpretability. The patch that denotes the difference may not reflect the actual activity that took place. Indeed, the d1 patch in the above example does not reveal that changes took place in two steps. Since both the content and the name of a column have changed, an unknowing reader of the patch might conclude that a whole column was removed and a new one added (rather than imputation followed by a rename). The daff format can thus be used to accurately store all the changes made on a dataset, while the actual activities that led to those changes would need to be stored as extra information.

11.2.2 Summarizing Cell Changes

Pannekoek *et al.* (2014) propose a way to summarize the status of cells when comparing a dataset before and after a processing step (Figure 11.4). The idea is to count, after processing, the total number of cells in a dataset. This number is then split up into cells with

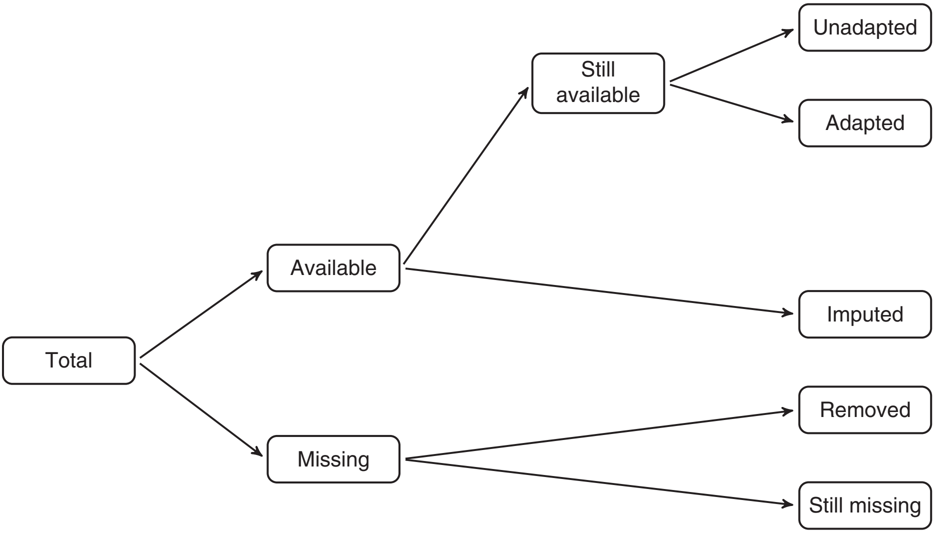


Figure 11.4 A classification of cell status when comparing a dataset after some processing with the dataset before processing. The 'total' is equal to the number of cells in a dataset.

available observations and missing observations. Prior to processing, these missing values either were available, which means that they were taken from the dataset (imputed), or were already missing to begin with. Similarly, the cells with values in them may have been imputed, or they were already filled prior to processing. In the latter case, the values may have changed or not.

The `validate` package exports the `cells` function, which computes each of these counts for two or more datasets.

```
iris1 <- iris
iris1[1:3,1] <- NA
iris1[2:4,2] <- iris1[2:4,2]*2

library(validate)
cells(start = iris, step1 = iris1)
## Object of class cellComparison:
##
##    cells(start = iris, step1 = iris1)
##
##                  start step1
## cells              750   750
## available          750   747
## missing              0     3
## still_available    750   747
## unadapted          750   744
## adapted              0     3
## imputed              0     0
## new_missing          0     3
## still_missing        0     3
```

`cells` accepts any number of data frames, comparing every data frame (except the first) with the first dataset.

```
iris2 <- iris1
iris2[1:3, 3] <- -(1:3)
cells(start = iris, step1 = iris1, step2=iris2)
## Object of class cellComparison:
##
##    cells(start = iris, step1 = iris1, step2 = iris2)
##
##                   start step1 step2
## cells               750   750   750
## available           750   747   747
## missing               0     3     3
## still_available   750   747   747
## unadapted           750   744   741
## adapted               0     3     6
## imputed               0     0     0
## new_missing           0     3     3
## still_missing         0     3     3
```

It is also possible to compare data frames sequentially; for this, use the option `compare="sequential"`.

11.2.3 Summarizing Changes in Conformance to Validation Rules

Like for cell status, one can compare the status of a dataset as measured by validation rule satisfaction after and before a data-cleaning step. Recall that when a validation rule is evaluated for a dataset, there are three possible outcomes: `TRUE` if the rule is satisfied, `FALSE` if the rule is violated, or NA if the rule cannot be evaluated because one or more of the necessary values are missing.

A classification of changes in these outcomes due to a data-cleaning step has been suggested by van den Broek *et al.* (2014) and is shown in Figure 11.5. In this classifications, the total number of outcomes of a validation procedure on the processed dataset

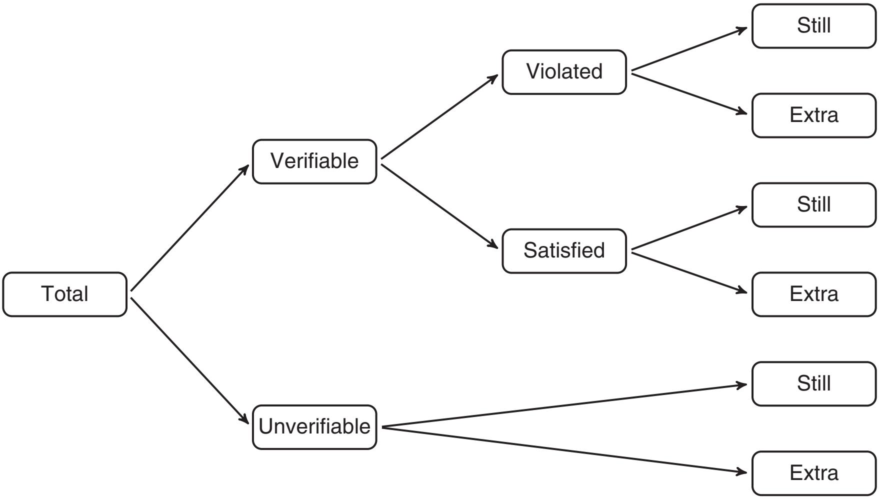


Figure 11.5 A classification of changes in rule violation status before and after a data-cleaning step.

is first split into outcomes that are verifiable and outcomes that are unverifiable (NA). The outcomes that are not NA can be split up further into outcomes that result in FALSE (violated) and in TRUE (satisfied). For each of those we count how many were violated (satisfied) prior to processing and how many switched from one status to the other. Similarly, we count for the currently unverifiable checks, how many were unverifiable to begin with and how many have become unverifiable.

The validate package exports the compare function, which computes these numbers for two or more datasets. Using the same example data as in Section 11.2.2, and introducing a few rules, we get the following.

```
v <- validator(
  Sepal.Length >= 0
 , Petal.Length >= 0
)
compare(v, start=iris, step1=iris1, step2=iris2)
## Object of class validatorComparison:
##
##    knitr::knit("main.Rnw", encoding = "UTF-8")
##
##                      Version
## Status                start step1 step2
##   validations           300   300   300
##   verifiable            300   297   297
##   unverifiable            0     3     3
##   still_unverifiable      0     0     0
##   new_unverifiable        0     3     3
##   satisfied             300   297   294
##   still_satisfied       300   297   294
##   new_satisfied           0     0     0
##   violated                0     0     3
##   still_violated          0     0     0
##   new_violated            0     0     3
```

By default, all data frames are compared to the first data frame, but they can be consecutively compared by adding the argument how="sequential".

11.2.4 Track Changes in Data Automatically with lumberjack

Tracking changes in data is fairly easy when two or more versions of a dataset are available. However, storing and handling multiple versions of a dataset quickly becomes cumbersome when multiple processing steps are involved, especially if one wishes to experiment with different steps, different parameterizations, or different orders of data-cleaning steps.

The lumberjack package offers a way to specify the logging procedure separate from the data processing. The logging action itself needs to take place between two processing steps, which is where the current dataset can be compared with the previous version. To facilitate this, the lumberjack package implements a function composition operator (also called a 'pipe' operator) that executes logging code when needed. The utility functions start_log(), dump_log(), and stop_log() control the logger.

Getting Started

Consider a small example, where we impute some data using simputation.

```
library(validate)
library(simputation)
library(lumberjack)

data(retailers)

# we add a unique row-identifier
retailers$id <- seq_len(nrow(retailers))

# create a logging object.
logger <- cellwise$new(key="id")

out <- retailers %>>%
  start_log(logger) %>>%
  impute_lm(staff ~ turnover) %>>%
  impute_median(staff ~ size) %>>%
  dump_log(file="mylog.csv", stop=TRUE)
```

```
## Dumped a log at mylog.csv
```

Here, we first add a unique identifier to the retailers dataset since the logger we are going to use needs one. Next, a cellwise logger is created. This logger is then passed to start_logger at the beginning of the data-processing pipeline. The %>>% operator works similar to the well-known %>% operator of the magrittr[1] package, except that it also makes sure that the logger gets a chance to measure the difference between the input and the output. After processing, ask the logger to dump its logging info to a csv file and to stop logging. The resulting data is stored in out.

The cellwise logger locates cells that have changed and lists a time stamp, the expression that is responsible for the change, the cell location, and its old and new values. We can inspect its output as follows:

```
log <- read.csv("mylog.csv")
head(log)
##   step                     time                  expression key variable
## 1    1 2017-06-30 16:34:49 CEST impute_lm(staff ~ turnover)  14    staff
## 2    1 2017-06-30 16:34:49 CEST impute_lm(staff ~ turnover)   3    staff
## 3    1 2017-06-30 16:34:49 CEST impute_lm(staff ~ turnover)  40    staff
## 4    1 2017-06-30 16:34:49 CEST impute_lm(staff ~ turnover)  43    staff
## 5    1 2017-06-30 16:34:49 CEST impute_lm(staff ~ turnover)   4    staff
## 6    2 2017-06-30 16:34:49 CEST impute_median(staff ~ size)   5    staff
##   old      new
## 1  NA 96.88595
## 2  NA 10.84707
## 3  NA 10.61990
## 4  NA 10.31884
## 5  NA 10.56555
## 6  NA  1.00000
```

[1] There are differences, for example, %>>% cannot be used to define functions.

So, for example, in row 14, the variable `staff` was imputed by `impute_lm`. Such a log can give quick insights into the effect of each imputation step. For example, which function performed the most imputations?

```
table(log$expression)
##
## impute_lm(staff ~ turnover) impute_median(staff ~ size)
##                           5                           1
```

In this case, five values were imputed in the first linear imputation step, and a single value was imputed through median imputation.

It is not necessary to put all processing steps in a single stream. The abovementioned results could also be achieved with the following code:

```
# re-read data for this example
data(retailers)
retailers$id <- seq_len(nrow(retailers))

logger <- cellwise$new(key="id")
retailers <- start_log(retailers, logger)
retailers <- retailers %>>% impute_lm(staff ~ turnover)
retailers <- retailers %>>% impute_median(staff ~ size)
dump_log(retailers,file="mylog.csv", stop=TRUE)

 ## Dumped a log at mylog.csv
```

Other Loggers

The `lumberjack` package is equipped with a few loggers, but it is set up in such a way that users or package authors can write their own loggers. Some loggers that are worth mentioning are the following:

- The `lbj_daff` logger that comes with the `daff` package. It summarizes changes in cells as described in Section 11.2.1.
- The `lbj_cells` logger that comes with the `validate` package. It summarizes changes in cells as described in Section 11.2.2.
- The `lbj_rules` logger that comes with the `validate` package. It summarizes changes in rule conformance as described in Section 11.2.3.

Loggers may be implemented as R `Reference` classes or as `R6` classes, as long as they follow lumberjack's interface. This means that initializing a logger can happen in two ways. For `R6` classes, it is done with `<logger>$new(<options>)`, while for `Reference` classes, it is done with `<logger>(<options>)`. In the following section, we update our example data-cleaning system and with the `lbj_rules` logger.

Tracking Changes in the Example

The following script is modified to use the `lbj_rules` logger that comes with the `validate` package. For compactness of presentation, all data-cleaning commands have been put into a single data pipeline. Instead of storing missing value locations for later adjustment, we store those locations by tagging the dataset using `tagg_missings`. This function is exported by `rspa`, and the tag is recognized by `match_restrictions`.

```
library(validate)
library(dcmodify)
library(deductive)
library(errorlocate)
library(simputation)
library(rspa)
library(lumberjack)

# Read data
data(retailers, package="validate")

# Read validation rules
rules <- validator(.file="rules.txt")
voptions(rules, lin.eq.eps=0.01,lin.ineq.eps=0.01)
# Initialize logger.
logger <- lbj_rules(rules)

# tag 'retailers' for logging
retailers <- start_log(retailers, logger)

## Start cleaning

# Read modifying rules
mods <- modifier(.file="modify.txt")

# apply modifiers
retailers <- retailers %>>%
  modify(mods, sequential=TRUE) %>>%
  correct_typos(rules) %>>%
  replace_errors(rules) %>>%
  impute_lr(rules) %>>%
  tag_missing() %>>%
  impute_rlm(turnover + other.rev + staff.costs +
      total.costs + profit ~ staff + total.rev) %>>%
  impute_rlm(turnover + other.rev + staff.costs +
      total.costs + profit ~ total.rev) %>>%
  impute_mf(. ~ .) %>>%
  match_restrictions(rules) %>>%
  dump_log(file="cleaninglog.csv",stop=TRUE)

## Done!

# write data
write.csv(retailers, file="retailers_clean.csv", row.names=FALSE)
```

We can read the log as a simple csv file again and study its output.

```
logdata <- read.csv("R/cleaninglog.csv")
```

The table is too large for presentation here, and so we provide a simple plot of two of its columns in Figure 11.6. We see that the number of violations first increases during the data-modifying step. This may indicate that the modifiers have too strict assumptions, for example, about the unit-of-measure errors, or perhaps not enough of them are found causing inconsistencies in balance restrictions. After the modifications, we see that the typo correction decreases the number of violations somewhat. After error localization, all violations disappear, while the number of unverifiable validations surges. This is to

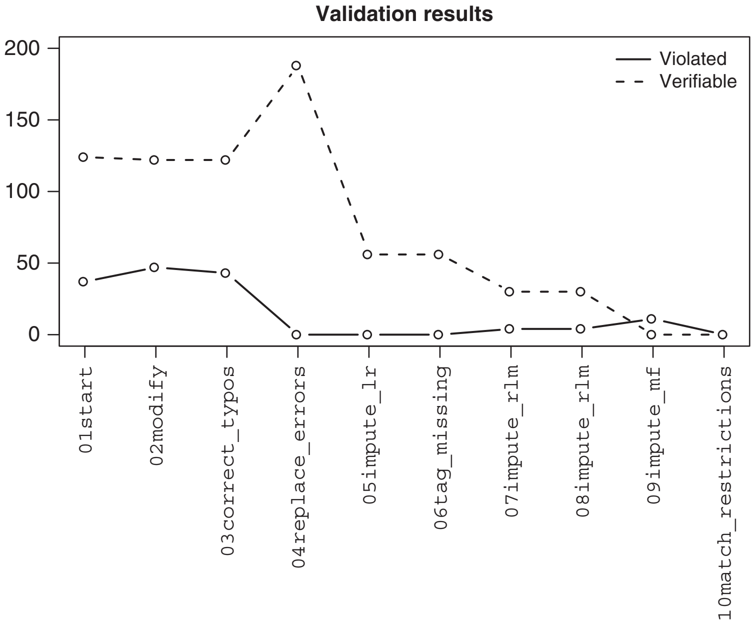


Figure 11.6 Progression of the number of violations and the number of unverifiable validations as a function of the step number in the data-cleaning process.

be expected: recall that during our error localization, step values that are deemed erroneous are removed. Next, the deductive imputation takes place, causing a drop in the unverifiables, while the number of violations stays equal to zero. This is also expected since this imputation step looks for unique values forced by the data and the validation rules. The model-based imputation steps increase the number of violations again since they do not take validation rules into account, while the number of verifiable validations steadily decreases. Finally, after adjustment, all violations disappear.

11.3 Integration and Automation

The example data-cleaning script contains a few parameters that may be convenient to control externally. For example, if the procedure is to be applied frequently to a new dataset, the name of the input and output files may change. Alternatively, the script may need to be integrated into a database or data analyses platform that is not R.

There are many solutions to integrate R scripts into other platforms, both commercial and open source. In this last section, we will point out a method that works on any platform since it is provided by R itself. The idea is to prepare our script for the following workflow:

1) A user or software creates a csv file with data to be cleaned. We cannot make any assumptions about the filename.
2) The file is read by the R script and processed with the steps described in Section 11.1, while a log of changes in rule failure is maintained.

3) After completion, a csv file with processed data and a csv file with the logging info are written to files that can be chosen by the user.

11.3.1 Using RScript

To prepare our script, one should first know that it is possible to execute an R script from the command line using the `Rscript` command. Here, with command line, we mean any command line supported by your operating system, for example, a bash shell for Linux or Apple users and a DOS box or Powershell for Windows users. For example, if we have a file called `myfile.R`

```
print("hello world")
```

then the command

```
$: Rscript myfile.R
[1] "hello world"
```

will start R, run the script, and exit R again.

The text to print can be made configurable by passing extra arguments to `Rscript` and catching them with the `commandArgs` function. So, we change our script as follows:

```
L <- commandArgs(TRUE)
print(paste("hello ",L[[1]]))
```

The argument `TRUE` signals that the first argument (the name of the script file) is of no importance to us. The result of a call to `commandArgs` is a `list`. Here is how to use it (`$` denotes the command prompt).

```
$ Rscript myscript.R "happy user"
[1] "hello  happy user"
```

11.3.2 The docopt Package

As scripts become more involved, and the number of options grows, parsing all options retrieved with `commandArgs` can become cumbersome. The `docopt` package makes defining and processing options much easier while also automatically creating a help index for the script. Below is our little running example, but now using `docopt`.

```
suppressPackageStartupMessages(library(docopt))
"
Usage: myscript.R [-h STRING]
-h      Show this help
STRING  The string to print after 'hello'
" -> doc

opt <- docopt(doc)

print(paste("hello ",opt$STRING))
```

In the first line we quietly load the package. Next, we create a string literal, stored in `doc` that is precisely the help information to be printed when a user requires it. The first

```
library(docopt)
"
Usage: clean.R [-h] INFILE OUTFILE RULEFILE MODFILE
-h       print this message
INFILE   input csv file location
RULEFILE file containing validation rules
MODFILE  file containing modifying rules
OUTFILE output csv file location
LOGFILE output logfile location
" -> doc

opt <- docopt(doc)

# library calls
library(validate)
library(dcmodify)
library(deductive)
library(errorlocate)
library(simputation)
library(rspa)
library(lumberjack)

# Read data
dat <- read.csv(opt$INFILE)

# Read validation rules
rules <- validator(.file=opt$RULES)
voptions(rules, lin.eq.eps=0.01,lin.ineq.eps=0.01)
# Initialize logger.
logger<- lbj_rules(rules)

# tag 'dat' for logging
dat <- start_log(dat, logger)

## Cleaning happens here
# Read modifying rules
mods <- modifier(.file=opt$RULES)
# apply modifiers
out <- dat %>>%
  modify(mods,sequential=TRUE) %>>%
  correct_typos(rules) %>>%
  replace_errors(rules) %>>%
  impute_lr(rules) %>>%
  tag_missing() %>>%
  impute_rlm(turnover + other.rev + staff.costs +
     total.costs + profit ~ staff + total.rev) %>>%
  impute_rlm(turnover + other.rev + staff.costs +
     total.costs + profit ~ total.rev) %>>%
  impute_mf(. ~ .) %>>%
  match_restrictions(rules) %>>%
  dump_log(file=LOGFILE,stop=TRUE)

write.csv(dat,file=OUTFILE, row.names=FALSE)
```

Figure 11.7 The completed data-cleaning script, automated using docopt.

line of the help info specifies how the command can be called: either with -h or with a string (the ‘or’ is implied by putting both arguments between square brackets). The help info also specifies that --help works as well. By calling opt <- docopt(doc) the command-line arguments are parsed and stored in opt. The latter object is a named list.

```
$ Rscript myscript.R -help
Usage: myscript.R [-h STRING]
-h      Show this help
STRING  The string to print after 'hello'


$ Rscript myscript.R "pretty User"
[1] "hello  pretty User"
```

11.3.3 Automated Data Cleaning

In Figure 11.7, the adapted data-cleaning script is shown. The input file, output file, validation rule file, and modifying rule file have all been made changeable optionals. The docopt package is used for argument handling and will provide an informative message if an option is missing, for example. Further automation steps could make the script more robust, for example, by checking whether files exist, making modifying rules optional, and so on. Whether this is desirable is up to the agreement between the provider and the user of the script and therefore is beyond the scope of this book.

References

Adler, D., Gläser, C., Nenadic, O., Oehlschlägel, J., and Zucchini, W. (2014) ff: Memory-Efficient Storage of Large Data on Disk and Fast Access Functions, R Package Version 2.2-13.

Allen, J.D., Anderson, D., Becker, J., Cook, R., Davis, M., Edberg, P., Everson, M., Freytag, A., Iancu, L., Ishida, R., Jenkins, J.H., Lunde, K., McGowan, R., Moore, L., Muller, E., Phillips, A., Pournader, R., Suignard, M., and Whistler, K. (2014) The Unicode Standard – Version 7.9 The Unicode Consortium, http://www.unicode.org/versions/Unicode7.0.0/ (accessed 14 November 2014).

Anderson, P.D. (2002) *Missing Data*, Quantitative Applications in the Social Sciences, vol. **136**, Sage University Publications.

Andridge, R.R. and Little, R.J. (2010) A review of hot deck imputation for survey non-response. *International Statistical Review*, **78** (1), 40–64.

Arlot, S., Celisse, A. *et al.* (2010) A survey of cross-validation procedures for model selection. *Statistics Surveys*, **4**, 40–79.

Armstrong, W. (1974) Dependency structures of data base relationships. *IFIP Congress*, pp. 580–583.

Arnold, B.C., Castillo, E., and Sarabia, J.M. (1999) *Conditional Specification of Statistical Models*, Springer Science & Business Media.

Bache, S.M. and Wickham, H. (2014) magrittr: A Forward-Pipe Operator for R, R Package version 1.5.

Bailar, B.A. and Bailar, J.C. (1979) Comparison of the biases of the “hot-deck” imputation procedure with an “equal-weights” imputation procedure. *Symposium on Incomplete Data: Preliminary Proceedings (Panel on Incomplete Data of the Committee on National Statistics/National Research Council)*, pp. 422–447.

Barkov, A. (2014) Collation-Charts.Org, http://collation-charts.org (accessed 24 November 2014).

Barzi, F. and Woodward, M. (2004) Imputations of missing values in practice: results from imputations of serum cholesterol in 28 cohort studies. *American Journal of Epidemiology*, **160** (1), 34–45.

Beider, A. and Morse, S. (2008) Beider-morse phonetic matching: an alternative to soundex with fewer false hits. *Avotaynu: The International Review of Jewish Genealogy (Summer 2008).*

Berkelaar, M., Eikland, K., and Notebaert, P. (2010) lpSolve, Version 5.5.2.0 Released 8 December 2010.

Binder, D.A. (1996) Comment. *Journal of the American Statistical Association*, **91**, 510–512.

BIPS (2006) The International System of Units (SI) SI Brochure, http://www.bipm.org/en/publications/si-brochure/second.html (accessed 21 November 2014).

Boytsov, L. (2011) Indexing methods for approximate dictionary searching: comparative analyses. *ACM Journal of Experimental Algorithmics*, **16**, 1–86.

Bradley, S., Hax, A., and Magnanti, T. (1977) *Applied Mathematical Programming*, Addison-Wesley.

Brand, J. (1999) Development, implementation and evaluation of multiple imputation strategies for the statistical analysis of incomplete data sets. PhD thesis. Erasmus Universiteit Rotterdam.

Breiman, L. (2001) Random forests. *Machine Learning*, **45** (1), 5–32.

Breiman, L. and Cutler, A. (2004) Random Forests, Fortran Code Version 5.1.

Breiman, L., Friedman, J., Stone, C., and Olshen, R. (1984) *Classification and Regression Trees*, The Wadsworth and Brooks-Cole Statistics-Probability Series, Taylor & Francis.

Brown, L.D. (1986) Fundamentals of statistical exponential families with applications in statistical decision theory. *Lecture Notes-Monograph Series*, **9**, i-iii+v-vii+ix-x+1–279.

Bruni, R. and Bianchi, G. (2012) A formal procedure for finding contradictions into a set of rules. *Applied Mathematical Sciences*, **6** (126), 6253–6271.

Cavnar, W. and Trenkle, J. (1994) N-gram-based text categorization. *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pp. 161–175.

Chen, J. and Shao, J. (2001) Jackknife variance estimation for nearest-neighbor imputation. *Journal of the American Statistical Association*, **96** (453), 260–269.

Chen, K., Ying, Z., Zhang, H., and Zhao, L. (2008) Analysis of least absolute deviation. *Biometrika*, **95** (1), 107–122.

Clement, R. and Sharp, D. (2003) N-gram and Bayesian classification of documents for topic and authorship. *Literary and Linguistic Computing*, **18** (4), 423–447.

Codd, E.F. (1970) A relational model of data for large shared data banks. *Communications of the ACM*, **13** (6), 377–387.

Cohen, W., Ravikumar, P., and Fienberg, S. (2003) A comparison of string metrics for matching names and records. *KDD Workshop on Data Cleaning and Object Consolidation, vol. 3*, pp. 73–78.

Costello, A. (2003) *Punycode: A Bootstring Encoding of Unicode for Internationalized Domain Names in Applications (IDNA) Memo of the Network Working Group*, http://www.ietf.org/rfc/rfc3492.txt (accessed 14 November 2014).

Cox, B.G. (1980) The weighted sequential hot deck imputation procedure. *Proceedings of the American Statistical Association, Section on Survey Research Methods*, pp. 721–726.

Cranmer, S.J. and Gill, J. (2013) We have to be discrete about this: a non-parametric imputation technique for missing categorical data. *British Journal of Political Science*, **43** (02), 425–449.

Csardi, G. and Nepusz, T. (2006) The igraph software package for complex network research. *InterJournal Complex Systems*, **1695** (5), 1–9.

Daalmans, J. (2015) Simplifying Constraints in Data Editing. Technical Report 2015|18, Statistics Netherlands, The Hague/Heerlen.

de Jonge, E. (2016) docopt: Command-Line Interface Specification Language, R Package Version 0.4.5.

de Jonge, E. and van der Loo, M. (2015) editrules: Parsing, Applying, and Manipulating Data Cleaning Rules, R Package Version 2.9.0.

de Jonge, E. and van der Loo, M. (2016) errorlocate: Locate Errors with Validation Rules, R Package Version 0.1.2.

de Jonge, E., Wijffels, J., and van der Laan, J. (2015) ffbase: Basic Statistical Functions for Package ‘ff’, R Package Version 0.12.1.

de Waal, T. (2003) Processing of erroneous and unsafe data. PhD thesis. Erasmus University Rotterdam.

de Waal, T. (2017) Imputation methods satisfying constraints, *Work Session on Statistical Data Editing*, UNECE, http://www1.unece.org/stat/platform/display/WSSDE/Work+Session+on+Statistical+Data+Editing+2017 (accessed 30 August 2017).

de Waal, T. and Quere, R. (2003) A fast and simple algorithm for automatic editing of mixed data. *Journal of Official Statistics*, **19**, 383–402.

de Waal, T., Pannekoek, J., and Scholtus, S. (2011) *Handbook of Statistical Data Editing and Imputation*, Wiley Handbooks in Survey Methodology, John Wiley & Sons, Inc.

Dempster, A.P., Laird, N.M., and Rubin, D.B. (1977) Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, **39** (1), 1–38.

Dillig, I., Dillig, T., and Aiken, A. (2010) in *International Static Analysis Symposium*, vol. **6337** (eds R. Cousot and M. Martel), Springer-Verlag, Berlin, Heidelberg, pp. 236–252.

Donders, A.R.T., van der Heijden, G.J., Stijnen, T., and Moons, K.G. (2006) Review: a gentle introduction to imputation of missing values. *Journal of Clinical Epidemiology*, **59** (10), 1087–1091.

Elff, M. (2015) memisc: Tools for Management of Survey Data, Graphics, Programming, Statistics, and Simulation, R Package Version 0.97.

Eltinge, J.L. (1996) Comment. *Journal of the American Statistical Association*, **91**, 513–515.

Engels, J.M. and Diehr, P. (2003) Imputation of missing longitudinal data: a comparison of methods. *Journal of Clinical Epidemiology*, **56** (10), 968–976.

ESS (2015) Essnet on Validation Deliverable 1, http://www.cros-portal.eu/content/validat-foundation (accessed 16 July 2015).

Fay, R.E. (1996) Alternative paradigms for the analysis of imputed survey data. *Journal of the American Statistical Association*, **91** (434), 490–498.

Fellegi, I.P. and Holt, D. (1976) A systematic approach to automatic edit and imputation. *Journal of the American Statistical Association*, **71**, 17–35.

Fitzmaurice, G., Davidian, M., Verbeke, G., and Molenberghs, G. (2008) *Longitudinal Data Analysis*, CRC Press.

Fitzpatrick, P., de Jonge, E., and Warnes, G.R. (2017) *daff: Diff, Patch and Merge for Data.frames, R Package Version 0.3.0*.

Fowler, M. (2010) *Domain Specific Languages*, 1st edn, Addison-Wesley Professional.

Friedl, J. (2006) *Mastering Regular Expressions*, 3rd edn, O’Reilly.

Friedman, J., Hastie, T., and Tibshirani, R. (2010) Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, **33** (1), 1.

Fürnkranz, J. (1998) A Study Using N-gram Features for Text Categorization. Technical Report OEFAI-TR-98-30, Austrian Research Institute for Artificial Intelligence, Schottengasse 3, A-1010 Wien, Austria.

Gagolewski, M. and Tartanus, B. (2014) R Package Stringi: Character String Processing Facilities.

Gelman, A. (2004) Parameterization and Bayesian modeling. *Journal of the American Statistical Association*, **99** (466), 537–545.

Gelman, A. and Hill, J. (2006) *Data Analysis Using Regression and Multilevel/Hierarchical Models*, Cambridge University Press.

Gelsema, T. (2012) The organization of information in a statistical office. *Journal of Official Statistics*, **28** (3), 413–440.

Gibbons, J. (2013) in *Central European Functional Programming - Summer School on Domain-Specific Languages*, LNCS, vol. **8606**, (eds V. Zsok, Z. Horvath, and L. Csato), Springer, Cham, pp. 1–28.

Golub, G. and van Loan, C. (1996) *Matrix Computations*, 3rd edn, The John Hopkins University Press.

Gower, J.C. (1971) A general coefficient of similarity and some of its properties. *Biometrics*, **27** (4), 857–871.

Graham, J.W., Olchowski, A.E., and Gilreath, T.D. (2007) How many imputations are really needed? Some practical clarifications of multiple imputation theory. *Prevention Science*, **8** (3), 206–213.

Greville, T. (1959) The pseudoinverse of a rectangular or singular matrix and its application to the solution of systems of linear equations. *SIAM Review*, **1** (1), 38–43.

Grolemund, G. and Wickham, H. (2011) Dates and times made easy with lubridate. *Journal of Statistical Software*, **40** (3), 1–25.

Hamming, R. (1950) Error detecting and error correcting codes. *The Bell System Technical Journal*, **29**, 147–160.

Hampel, F.R., Ronchetti, E.M., Rousseeuw, P.J., and Stahel, W.A. (1986) *Robust Statistics: The Approach Based on Influence Functions*, vol. **114**, John Wiley & Sons, Inc., New York.

Hastie, T., Tibshirani, R., and Friedman, J. (2001) *The Elements of Statistical Learning*, Springer Series in Statistics, vol. **1**, Springer-Verlag, Berlin.

Heckerman, D., Chickering, D.M., Meek, C., Rounthwaite, R., and Kadie, C. (2000) Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research*, **1**, 49–75.

Heller, I. and Tompkins, C. (1956) in *Linear Inequalities and Related Systems* (eds H. Kuhn and A. Tucker), Princeton University Press, Princeton, NJ, pp. 247–254.

Hildreth, C. (1957) A quadratic programming procedure. *Naval Research Logistics*, **4** (1), 79–85.

Hoerl, A.E. and Kennard, R.W. (1970) Ridge regression: biased estimation for nonorthogonal problems. *Technometrics*, **12** (1), 55–67.

Holmes, D. and McCabe, M.C. (2002) Improving precision and recall for soundex retrieval. *Proceedings of the International Conference on Information Technology: Coding and Computing, 2002*, IEEE, pp. 22–26.

Honaker, J., King, G., and Blackwell, M. (2011) Amelia II: a program for missing data. *Journal of Statistical Software*, **45** (7), 1–47.

Hopcroft, J., Motwani, R., and Ullman, J. (2007) *Introduction to Automata Theory, Languages, and Computation*, 3rd edn, Addison Wesley.

Hornik, K. (2014) Personal communication.

Hornik, K., Mair, P., Rauch, J., Geiger, W., Buchta, C., and Feinerer, I. (2013) The textcat package for N-gram based text categorization in R. *Journal of Statistical Software*, **52** (6), 1–17.

Huber, P.J. (2011) *Robust Statistics*, Springer-Verlag.

Huber, P.J. *et al.* (1964) Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, **35** (1), 73–101.

Huffman, S. (1995) Acquaintance: Language-Independent Document Categorization by N-Grams. Technical Report, DTIC Document.

Huffman, S. and Damashek, M. (1995) *Acquaintance: A Novel Vector-Space N-Gram Technique for Document Categorization*, NIST Special Publication SP, p. 305.

Huisman, M. (2009) Imputation of missing network data: some simple procedures. *Journal of Social Structure*, **10** (1), 1–29.

Hyndman, R.J. and Athanasopoulos, G. (2014) *Forecasting: Principles and Practice*, OTexts.

IANA (2014) Internet Assigned Numbers Authority Web Site, http://www.iana.org/time-zones (accessed 16 December 2014).

ICU (2014) International Components for Unicode, http://site.icu-project.org/ (accessed 20 November 2014).

IEEE Std. 754-2008. (2008) *Standard for Floating Point Arithmetic*, IEEE.

IERS (2014) International Earth Rotation and Reference Systems Service Bulletin C (leap second announcements), http://www.iers.org/SharedDocs/News/EN/BulletinC.html (accessed 26 November 2014).

Imai, K., King, G., and Lau, O. (2008) Toward a common framework for statistical analysis and development. *Journal of Computational Graphics and Statistics*, **17** (4), 892–913.

ISO 8601:2004(E). (2004) *Data Elements and Interchange Formats – Information Interchange – Representation of Dates and Times International Standard ISO*, 3rd edn, ISO.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013) *An Introduction to Statistical Learning with Applications in R*, Springer Texts in Statistics, Springer-Verlag, New York.

Jaro, M. (1978) *UNIMATCH: A Record Linkage System: User Manual*, United States Bureau of the Census, pp. 103–108.

Jaro, M. (1989) Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *Journal of the American Statistical Association*, **84**, 414–420.

Joenssen, D.W. and Bankhofer, U. (2012) Hot deck methods for imputing missing data, *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, vol. **7376** (eds P. Perner), Springer-Verlag, Berlin, Heidelberg, pp. 63–75.

Judkins, D.J. (1996) Comment. *Journal of the American Statistical Association*, **91**, 507–510.

Kalton, G. and Kasprzyk, D. (1986) The treatment of missing survey data. *Survey Methodology*, **12**, 1–16.

Karatzoglou, A. and Feinerer, I. (2007) Text clustering with string Kernels in R, in *Advances in Data Analysis. Studies in Classification, Data Analysis, and Knowledge Organization* (eds R. Decker and H.J. Lenz), Springer-Verlag, Berlin, Heidelberg.

Kennickell, A.B. (1991) Imputation of the 1989 survey of consumer finances: stochastic relaxation and multiple imputation. *Proceedings of the Survey Research Methods Section of the American Statistical Association*, pp. 1–10.

Kešelj, V., Peng, F., Cercone, N., and Thomas, C. (2003) N-gram-based author profiles for authorship attribution. *Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING, vol. 3*, pp. 255–264.

Kim, S.H. and Ahn, B.S. (1999) Interactive group decision making procedure under incomplete information. *European Journal of Operational Research* **116** (3), 498–507.

Kleene, S. (1951) Representation of events in nerve nets and finite automata. Research memorandum RM-704, U.S. Airforce, 1700 Main street Santa Monica.

Klensin, J. and Padlipski, M. (2008) Unicode Format for Network Interchange. Network Working Group RFC 5198, http://tools.ietf.org/html/rfc5198 (accessed 17 December 2014).

Koenker, R. (2005) *Quantile Regression*, Economic Society Monographs, vol. **38**, Cambridge University Press.

Koller, M. and Mächler, M. (2016) Definitions of ψ Functions Available in Robustbase, Vignette of the 'Robustbase' Package.

Konis, K. (2011) lpSolveAPI: R Interface for lpsolve version 5.5.2.0, R Package Version 5.5.2.0-5.

Laurikari, V. (2001) Efficient submatch addressing for regular expressions. Master's thesis. Helsinki University of Technology.

Lawrence, S., Giles, C.L., and Bollacker, K.D. (1999) Autonomous citation matching. *Proceedings of the 3rd Annual Conference on Autonomous Agents*, ACM, pp. 392–393.

Levenshtein, V.I. (1966) Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, **10**, 707–710.

Liaw, A. and Wiener, M. (2002) Classification and regression by randomforest. *R News*, **2** (3), 18–22.

Lipshutz, S. and Lipson, M. (2009) *Linear Algebra*, Schaum's outlines 4th edn, McGraw Hill.

Little, R.J. and Rubin, D.B. (2002) *Statistical Analysis with Missing Data*, John Wiley & Sons, Inc., Hoboken, NJ.

Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., and Watkins, C. (2002) Text classification using string kernels. *Journal of Machine Learning Research*, **2**, 419–444.

Lowrance, R. and Wagner, R. (1975) An extension of the string-to-string correction problem. *Journal of the Association of Computing Machinery*, **22**, 177–183.

Maronna, R., Martin, D., and Yohai, V. (2006) *Robust Statistics*, John Wiley & Sons, Ltd, Chichester.

Mersmann, O. (2014) *microbenchmark: Accurate Timing Functions, R Package Version 1.4-2.*

Meyer, D., Hornik, K., and Feinerer, I. (2008) Text mining infrastructure in R. *Journal of Statistical Software*, **25** (5), 1–54.

Microsoft (2014) Locale IDs Assigned by Microsoft Web Page, http://msdn.microsoft.com/en-us/goglobal/bb964664.aspx (accessed 17 December 2014).

Mokotov, G. (1997) Soundexing and Genealogy, http://www.avotaynu.com/soundex.htm (accessed 30 November 2015).

Mousseau, V., Figueira, J., Dias, L., da Silva, C.G., and Clìmaco, J. (2003) Resolving inconsistencies among constraints on the parameters of an MCDA model. *European Journal of Operational Research*, **147** (1), 72–93.

NARA (2015) The Soundex Indexing System, http://www.archives.gov/research/census/soundex.html (accessed 30 November 2015).

Nash, J. (2014) *Nonlinear Parameter Optimization Using R Tools*, 1st edn, John Wiley & Sons.

Navarro, G. (2001) A guided tour to approximate string matching. *ACM Computing Surveys*, **33**, 31–88.

Needleman, S. and Wunsch, C.D. (1970) A general method applicable to the search of similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, **48**, 443–453.

Oehlschlägel, J. (2014) bit64: A S3 Class for Vectors of 64bit Integers, R package version 0.9-4.

Ono, M. and Miller, H.P. (1969) *Income Nonresponses in the Current Population Survey*, US Bureau of the Census.

Ooms, J. (2014) The jsonlite package: a practical and consistent mapping between JSON data and R objects, arXiv:1403.2805 [stat.CO].

Oudshoorn, C.G.M., Buuren, S., and Rijckevorsel, J.L.A. (1999) *Flexible Multiple Imputation by Chained Equations of the AVO-95 Survey*, TNO Prevention and Health Leiden.

Pannekoek, J. and Zhang, L.C. (2015) Optimal adjustments for inconsistency in imputed data. *Survey Methodology*, **41** (1), 127–144.

Pannekoek, J., van der Loo, M., and van den Broek, B. (2014) Implementation and evaluation of automatic editing for business surveys. *United Nations Economic Commission for Europe Work Session on Statistical Data Editing*, Paris.

PCRE (2015) Perl-Compatible Regular Expressions, http://www.pcre.org (accessed 11 December 2015).

Perez, A., Dennis, R.J., Gil, J.F., Rondón, M.A., and López, A. (2002) Use of the mean, hot deck and multiple imputation techniques to predict outcome in intensive care unit patients in Colombia. *Statistics in Medicine*, **21** (24), 3885–3896.

Philips, L. (2000) The double metaphone search algorithm. *C/C++ Users Journal*, **18** (6), 38–43.

Phillips, A. and Davis, M. (2009) Tags for identifying languages. Network working group RFC 5646; BCP 47, http://tools.ietf.org/html/rfc5646#page-6 (accessed 16 December 2014).

Pinto, D., Vilari no, D., Alemán, Y., Gómez, H., Loya, N., and Jiménez-Salazar, H. (2012) in *Text, Speech and Dialogue*, vol. **7499** (eds P. Sojka, A. Horák, I. Kope?ek, and K. Pala), Springer-Verlag, Berlin, Heidelberg, pp. 47–55.

R Core Team (2013) Changes in R, from 2.15.3 to 3.0.1. *The R Journal*, **5**, 221–238.

R Special Interest Group on Databases (2014) DBI: R Database Interface, R Package Version 0.3.1.

Raghavachari, M. (1976) A constructive method to recognize the total unimodularity of a matrix. *Zeitschrift für Operations Research*, **20**, 59–61.

Raghunathan, T.E., Lepkowski, J.M., Van Hoewyk, J., and Solenberger, P. (2001) A multivariate technique for multiply imputing missing values using a sequence of regression models. *Survey Methodology*, **27** (1), 85–96.

Rahman, G. and Islam, Z. (2011) A decision tree-based missing value imputation technique for data pre-processing. *Proceedings of the 9th Australasian Data Mining Conference, vol. 121*, Australian Computer Society, Inc., pp. 41–50.

Rao, J. (1996) On variance estimation with imputed survey data. *Journal of the American Statistical Association*, **91** (434), 499–506.

Rao, J.N. and Shao, J. (1992) Jackknife variance estimation with survey data under hot deck imputation. *Biometrika*, **79** (4), 811–822.

Recchia, G. and Louwerse, M. (2013) A comparison of string similarity measures for toponym matching. *Proceedings of ACM SIGSPATIAL CoMP*.

Reilly, M. and Pepe, M. (1997) The relationship between hot-deck multiple imputation and weighted likelihood. *Statistics in Medicine*, **16** (1), 5–19.

Renssen, R. and Van Delden, A. (2008) Standardisation of design and production of statistics; a service oriented approach at statistics Netherlands. *IAOS conference, Shanghai.*

Revolution Analytics (2015) rhadoop, A collection of R packages for Hadoop connectivity.

Rinker, T.W. (2014) *qdapRegex: Regular Expression Removal, Extraction, and Replacement Tools*, version 0.2.0, University at Buffalo/SUNY Buffalo, New York.

Ripley, B. and Hornik, K. (2001) Date-time classes. *R News*, **1** (2), 8–11, http://www.r-project.org/doc/Rnews/Rnews_2001-2.pdf (accessed 28 November 2013).

Ripley, B. and Lapsley, M. (2015) RODBC: ODBC Database Access, R package version 1.3-12.

Rubin, D.B. (1976) Inference and missing data. *Biometrika*, **63** (3), 581–592.

Rubin, D.B. (1978) Multiple imputations in sample surveys-a phenomenological Bayesian approach to nonresponse. *Proceedings of the Survey Research Methods Section of the American Statistical Association, vol. 1*, American Statistical Association, pp. 20–34.

Rubin, D.B. (1987) *Multiple Imputation for Nonresponse in Surveys*, John Wiley & Sons, Inc., New York.

Rubin, D.B. (1996) Multiple imputation after 18+ years. *Journal of the American Statistical Association*, **91** (434), 473–489.

Rubin, D.B. (2003) Nested multiple imputation of NMES via partially incompatible MCMC. *Statistica Neerlandica*, **57** (1), 3–18.

Russell, R. (1918) Index. US Patent 1,261,167.

Russell, R. (1922) Index. US Patent 1,435,663.

Schafer, J.L. (1997) *Analysis of Incomplete Multivariate Data*, CRC Press.

Schafer, J.L. and Graham, J.W. (2002) Missing data: our view of the state of the art. *Psychological Methods*, **7** (2), 147–177.

Scherer, M. and Davis, M. (2006) BOCU-1: MIME-Compatible Unicode Compression Unicode Technical Note #6, http://www.unicode.org/notes/tn6/ (accessed 14 November 2014).

Scholtus, S. (2009) Automatic Correction of Simple Typing Errors in Numerical Data with Balance Edits. Technical Report 09046, Statistics Netherlands.

Scholtus, S. (2015) A generalized Fellegi–Holt paradigm for automatic error localization. *Survey Methodology*, **42** (1), 1–18.

Schrijver, A. (1998) *Theory of Linear and Integer Programming*, Wiley-Interscience Series in Descrete Mathematics and Optimization, John Wiley & Sons, Inc., New York.

Sehgal, V., Getoor, L., and Viechnicki, P.D. (2006) Entity resolution in geospatial data integration. *Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems*, ACM, pp. 83–90.

Shah, A.D., Bartlett, J.W., Carpenter, J., Nicholas, O., and Hemingway, H. (2014) Comparison of random forest and parametric imputation models for imputing missing data using mice: a caliber study. *American Journal of Epidemiology*, **179** (6), 764–774.

Sheppard, S. (2014) Command Line Reference – Web, Database and OS Scripting Web Page, http://ss64.com (accessed 17 December 2014).

Stekhoven, D.J. and Bühlmann, P. (2012) Missforest-non-parametric missing value imputation for mixed-type data. *Bioinformatics*, **28** (1), 112–118.

Stephens, J. (2014) yaml: Methods to Convert R Data to YAML and Back, R Package Version 2.1.13.

Stoer, J. and Bulirsch, R. (2002) *Introduction to Numerical Analysis*, Texts in Applied Mathematics, 3rd edn, Springer-Verlag, New York.

Tang, L., Song, J., Belin, T.R., and Unützer, J. (2005) A comparison of imputation methods in a longitudinal randomized clinical trial. *Statistics in Medicine*, **24** (14), 2111–2128.

Templ, M., Alfons, A., and Filzmoser, P. (2012) Exploring incomplete data using visualization techniques. *Advances in Data Analysis and Classification*, **6** (1), 29–47.

Templ, M., Alfons, A., Kowarik, A., and Prantner, B. (2016) VIM: Visualization and Imputation of Missing Values, R Package Version 4.5.0.

Terrien, J. (1968) News from the international bureau of weights and measures. *Metrologia*, **4**, 41–44.

The Open Group IEEE Std. 1003.1. (1997) Single Unix Specification, http://pubs.opengroup.org/onlinepubs/007908799/xbd/re.html (accessed 23 January 2015).

The Open Group IEEE Std 1003.1. (2004) iconv – Codeset Conversion Function, 2004 Edition, http://pubs.opengroup.org/onlinepubs/009695399/functions/iconv.html (accessed 17 December 2014).

The Open Group IEEE Std. 1003.1. (2013) *Single Unix Specification*, http://www.unix.org/version4/ieee_std.html (accessed 20 November 2014).

Tibshirani, R. (1996) Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, **58** (1), 267–288.

Twisk, J. and de Vente, W. (2002) Attrition in longitudinal studies: how to deal with missing data. *Journal of Clinical Epidemiology*, **55** (4), 329–337.

Ukkonen, E. (1992) Approximate string-matching with *q*-grams and maximal matches. *Theoretical Computer Science*, **92**, 191–211.

Ushey, K. and Hester, J. (2014) rex: Friendly Regular Expressions, R Package Version 0.2.0.

van Buuren, S. and Groothuis-Oudshoorn, K. (2011) mice: multivariate imputation by chained equations in R. *Journal of Statistical Software*, **45** (3), 1–67.

van den Broek, B., van der Loo, M., and Pannekoek, J. (2014) Kwaliteitsmaten Voor Het Datacorrectieproces. Technical Report 201408, Statistics Netherlands (in Dutch).

van der Laan, J. (2015) LaF: Fast Access to Large ASCII Files, R Package Version 0.6.2.

van der Loo, M. (2014) The stringdist package for approximate string matching. *The R Journal*, **6**, 111–122.

van der Loo, M. (2015a) Experiences with R based data editing systems. *Romanian Statistical Review*, **2/2015**, 141–152.

van der Loo, M. (2015b) A formal typology of data validation functions. *United Nations Economic Committee for Europe work session on statistical data editing (Budapest).*

van der Loo, M. (2017a) lumberjack: Track Changes in Data the Tidy Way, R Package Version 0.1.0.

van der Loo, M. (2017b) rspa: Adapt Numerical Records to Fit (in)Equality Restrictions, R Package Version 0.2.1.

van der Loo, M. (2017c) Simputation: Simple Imputation, R Package Version 0.2.1.

van der Loo, M. and de Jonge, E. (2015a) dcmodify: Modify Data Using Externally Defined Modification Rules, R Package Version 0.0.1.

van der Loo, M. and de Jonge, E. (2015b) validate: data validation infrastructure.

van der Loo, M., and de Jonge, E. (2017) deductive: Data Correction and Imputation Using Deductive Methods, R Package Version 0.1.2.

van der Loo, M. and Pannekoek, J. (2014) Towards generic evaluation of data validation functions. *United Nations Economic Committee for Europe work session on statistical data editing (Paris).*

van der Loo, M., de Jonge, E., and Scholtus, S. (2011) Correction of Rounding, Typing and Sign Errors with the Deducorrect Package. Technical Report 2011019, Statistics Netherlands, The Hague/Heerlen.

van der Loo, M., Pannekoek, J., and Rijnveld, L. (2017) Computational estimates of data editing-related variance. *Work Session on Statistical Data Editing*, UNECE.

Venkataraman, S. (2013) sparkR: R frontend for spark, R Package Version 0.1.

W3C (2008) Extensible Markup Language (XML) 1.0 (5th edition) W3C Recommendation 26 November 2008, http://www.w3.org/TR/xml/ (accessed 14 November 2014).

w3tech (2014) Historical Trends in the Usage of Character Encodings for Websites, http://w3techs.com/technologies/history_overview/character_encoding/ms/y (accessed 15 November 2014).

Whistler, K., Davis, M., and Freytag, A. (2008) Unicode Character Encoding Model Unicode. Technical Report #17, http://www.unicode.org/reports/tr17/ (accessed 14 November 2014).

Wickham, H. (2007) Reshaping data with the reshape package. *Journal of Statistical Software*, **21** (12), 1–20.

Wickham, H. (2009) *ggplot2: Elegant Graphics for Data Analysis*, Springer-Verlag, New York.

Wickham, H. (2010) stringr: modern, consistent string processing. *The R Journal*, **2**, 38–40.

Wickham, H. (2011) The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, **40** (1), 1–29.

Wickham, H. (2014a) *Advanced R*, CRC Press.

Wickham, H. (2014b) Tidy data. *Journal of Statistical Software*, **59** (10), 1.

Wickham, H. and Francois, R. (2014) dplyr: A Grammar of Data Manipulation, R Package Version 0.3.0.2.

Wickham, H. and Francois, R. (2015) *readr: Read Tabular Data*. R package version 0.2.2.

Wickham, H. and Miller, E. (2015) haven: Import SPSS, Stata and SAS Files, R Package Version 0.2.0.

Wickham, H., James, D.A., and Falcon, S. (2014) RSQLite: SQLite Interface for R, R Package Version 1.0.0.

Wilkinson, L. (2005) *The Grammar of Graphics (Statistics and Computing)*, Springer-Verlag, New York, Secaucus, NJ.

Willeboordse, A. (2000) Towards a new Statistics Netherlands; blueprint for a process-oriented organisation structure. *Netherlands Official Statistics*, **15**, 46–50. The journal has been discontinued, but the paper is still available online.

Winkler, W. (1990) String comparator metrics and enhanced decision rules in the Fellegi–Sunter model of record linkage. *Proceedings of the Section on Survey Research Methods (American Statistical Association)*, pp. 354–359.

yaml.org (2015) YAML Aint Markup Language, http://yaml.org/ (accessed 13 August 2015).

Zamora, E., Pollock, J.J., and Zamora, A. (1981) The use of trigram analysis for spelling error detection. *Information Processing & Management*, **17** (6), 305–316.

Zhang, P. (2003) Multiple imputation: theory and method. *International Statistical Review*, **71** (3), 581–592.

Zou, H. and Hastie, T. (2005) Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **67** (2), 301–320.

Index