Statistics Netherlands

Discussion Paper

An introduction to data cleaning with R

The views expressed in this paper are those of the author(s) and do not necesarily reflect the policies of Statistics Netherlands

2013 | 13

Edwin de Jonge
Mark van der Loo

Publisher
Statistics Netherlands
Henri Faasdreef 312, 2492 JP The Hague
www.cbs.nl

Prepress: Statistics Netherlands, Grafimedia
Design: Edenspiekermann

Information
Telephone +31 88 570 70 70, fax +31 70 337 59 94
Via contact form: www.cbs.nl/information

Where to order
verkoop@cbs.nl
Fax +31 45 570 62 68
ISSN 1572-0314

60083 201313- X-10-13

An introduction to data cleaning with R

Edwin de Jonge and Mark van der Loo

Summary. Data cleaning, or data preparation is an essential part of statistical analysis. In fact, in practice it is often more time-consuming than the statistical analysis itself. These lecture notes describe a range of techniques, implemented in the R statistical environment, that allow the reader to build data cleaning scripts for data suffering from a wide range of errors and inconsistencies, in textual format. These notes cover technical as well as subject-matter related aspects of data cleaning. Technical aspects include data reading, type conversion and string matching and manipulation. Subject-matter related aspects include topics like data checking, error localization and an introduction to imputation methods in R. References to relevant literature and R packages are provided throughout.

These lecture notes are based on a tutorial given by the authors at the *useR!2013* conference in Albacete, Spain.

Keywords: methodology, data editing, statistical software

Contents

Notes to the reader . . . 6

1 Introduction 7

1.1 Statistical analysis in five steps . . . 7

1.2 Some general background in R . . . 8

1.2.1 Variable types and indexing techniques . . . 8

1.2.2 Special values . . . 10

Exercises . . . 11

2 From raw data to technically correct data 12

2.1 Technically correct data in R . . . 12

2.2 Reading text data into a R `data.frame` . . . 12

2.2.1 `read.table` and its cousins . . . 13

2.2.2 Reading data with `readLines` . . . 15

2.3 Type conversion . . . 18

2.3.1 Introduction to R's typing system . . . 19

2.3.2 Recoding factors . . . 20

2.3.3 Converting dates . . . 20

2.4 `character` manipulation . . . 23

2.4.1 String normalization . . . 23

2.4.2 Approximate string matching . . . 24

2.5 Character encoding issues . . . 26

Exercises . . . 29

3 From technically correct data to consistent data 31

3.1 Detection and localization of errors . . . 31

3.1.1 Missing values . . . 31

3.1.2 Special values . . . 33

3.1.3 Outliers . . . 33

3.1.4 Obvious inconsistencies . . . 35

3.1.5 Error localization . . . 37

3.2 Correction . . . 39

3.2.1 Simple transformation rules . . . 40

3.2.2 Deductive correction . . . 42

3.2.3 Deterministic imputation . . . 43

3.3 Imputation . . . 45

3.3.1 Basic numeric imputation models . . . 45

3.3.2 Hot deck imputation . . . 47

3.3.3 kNN-imputation 48
3.3.4 Minimal value adjustment 49
Exercises 51

Notes to the reader

This tutorial is aimed at users who have some `R` programming experience. That is, the reader is expected to be familiar with concepts such as variable assignment, `vector`, `list`, `data.frame`, writing simple loops, and perhaps writing simple functions. More complicated constructs, when used, will be explained in the text. We have adopted the following conventions in this text.

Code. All code examples in this tutorial can be executed, unless otherwise indicated. Code examples are shown in gray boxes, like this:

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

where output is preceded by a double hash sign `##`. When code, function names or arguments occur in the main text, these are typeset in `fixed width` font, just like the code in gray boxes. When we refer to `R` data types, like `vector` or `numeric` these are denoted in fixed width font as well.

Variables. In the main text, variables are written in slanted format while their values (when textual) are written in fixed-width format. For example: the *Marital status* is `unmarried`.

Data. Sometimes small data files are used as an example. These files are printed in the document in fixed-width format and can easily be copied from the `pdf` file. Here is an example:

```
%% Data on the Dalton Brothers
Gratt,1861,1892
Bob,1892
1871,Emmet,1937
% Names, birth and death dates
```

Alternatively, the files can be found at `http://tinyurl.com/mblhtsg`.

Tips. Occasionally we have tips, best practices, or other remarks that are relevant but not part of the main text. These are shown in separate paragraphs as follows.

Tip. *To become an R master, you must practice every day.*

Filenames. As is usual in `R`, we use the forward slash (`/`) as file name separator. Under windows, one may replace each forward slash with a double backslash `\\`.

References. For brevity, references are numbered, occurring as superscript in the main text.

1 Introduction

> Analysis of data is a process of inspecting, cleaning, transforming, and modeling data with the goal of highlighting useful information, suggesting conclusions, and supporting decision making.
>
> *Wikipedia, July 2013*

Most statistical theory focuses on data modeling, prediction and statistical inference while it is usually assumed that data are in the correct state for data analysis. In practice, a data analyst spends much if not most of his time on preparing the data before doing any statistical operation. It is very rare that the raw data one works with are in the correct format, are without errors, are complete and have all the correct labels and codes that are needed for analysis. Data Cleaning is the process of transforming raw data into consistent data that can be analyzed. It is aimed at improving the content of statistical statements based on the data as well as their reliability.

Data cleaning may profoundly influence the statistical statements based on the data. Typical actions like imputation or outlier handling obviously influence the results of a statistical analyses. For this reason, data cleaning should be considered a statistical operation, to be performed in a reproducible manner. The R statistical environment provides a good environment for reproducible data cleaning since all cleaning actions can be scripted and therefore reproduced.

1.1 Statistical analysis in five steps

In this tutorial a statistical analysis is viewed as the result of a number of data processing steps where each step increases the ``value" of the data[*].

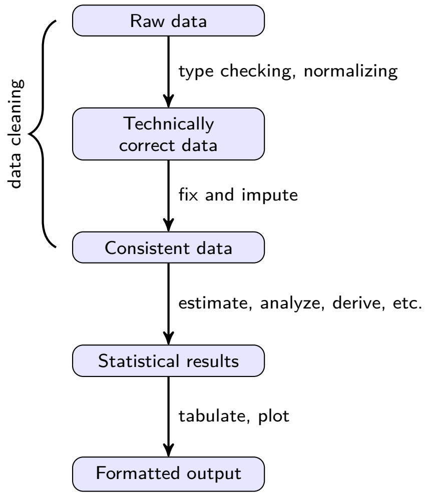


Figure 1: Statistical analysis value chain

Figure 1 shows an overview of a typical data analysis project. Each rectangle represents data in a certain state while each arrow represents the activities needed to get from one state to the other. The first state (*Raw data*) is the data as it comes in. Raw data files may lack headers, contain wrong data types (*e.g.* numbers stored as strings), wrong category labels, unknown or unexpected character encoding and so on. In short, reading such files into an R `data.frame` directly is either difficult or impossible without some sort of preprocessing.

Once this preprocessing has taken place, data can be deemed *Technically correct*. That is, in this state data can be read into an R `data.frame`, with correct names, types and labels, without further trouble. However, that does not mean that the values are error-free or complete. For example, an age variable may be reported negative, an under-aged person may be registered to possess a driver's license, or data may simply be missing. Such inconsistencies obviously depend on the subject matter

[*] In fact, such a value chain is an integral part of Statistics Netherlands business architecture.

that the data pertains to, and they should be ironed out before valid statistical inference from such data can be produced.

Consistent data is the stage where data is ready for statistical inference. It is the data that most statistical theories use as a starting point. Ideally, such theories can still be applied without taking previous data cleaning steps into account. In practice however, data cleaning methods like imputation of missing values will influence statistical results and so must be accounted for in the following analyses or interpretation thereof.

Once *Statistical results* have been produced they can be stored for reuse and finally, results can be *Formatted* to include in statistical reports or publications.

> **Best practice.** *Store the input data for each stage (raw, technically correct, consistent, aggregated and formatted) separately for reuse. Each step between the stages may be performed by a separate* R *script for reproducibility.*

Summarizing, a statistical analysis can be separated in five stages, from raw data to formatted output, where the quality of the data improves in every step towards the final result. Data cleaning encompasses two of the five stages in a statistical analysis, which again emphasizes its importance in statistical practice.

1.2 Some general background in R

We assume that the reader has some proficiency in R. However, as a service to the reader, below we summarize a few concepts which are fundamental to working with R, especially when working with ``dirty data".

1.2.1 Variable types and indexing techniques

If you had to choose to be proficient in just one R-skill, it should be indexing. By indexing we mean all the methods and tricks in R that allow you to select and manipulate data using `logical`, `integer` or named indices. Since indexing skills are important for data cleaning, we quickly review `vectors`, `data.frames` and indexing techniques.

The most basic variable in R is a `vector`. An R vector is a sequence of values of the same type. All basic operations in R act on vectors (think of the element-wise arithmetic, for example). The basic types in R are as follows.

`numeric`	Numeric data (approximations of the real numbers, $\mathbb{R}$)
`integer`	Integer data (whole numbers, $\mathbb{Z}$)
`factor`	Categorical data (simple classifications, like *gender*)
`ordered`	Ordinal data (ordered classifications, like *educational level*)
`character`	Character data (strings)
`raw`	Binary data

All basic operations in R work element-wise on vectors where the shortest argument is recycled if necessary. This goes for arithmetic operations (addition, subtraction,...), comparison operators (==, <=,...), logical operators (&, `|`, `!`,...) and basic math functions like `sin`, `cos`, `exp` and so on. If you want to brush up your basic knowledge of vector and recycling properties, you can execute the following code and think about why it works the way it does.

```
# vectors have variables of _one_ type
c(1, 2, "three")
# shorter arguments are recycled
(1:3) * 2
(1:4) * c(1, 2)
# warning! (why?)
(1:4) * (1:3)
```

Each element of a vector can be given a `name`. This can be done by passing named arguments to the `c()` function or later with the `names` function. Such names can be helpful giving meaning to your variables. For example compare the vector

```
x <- c("red", "green", "blue")
```

with the one below.

```
capColor = c(huey = "red", duey = "blue", louie = "green")
```

Obviously the second version is much more suggestive of its meaning. The names of a vector need not be unique, but in most applications you'll want unique names (if any).

Elements of a vector can be selected or replaced using the square bracket operator `[ ]`. The square brackets accept either a vector of names, index numbers, or a logical. In the case of a logical, the index is recycled if it is shorter than the indexed vector. In the case of numerical indices, negative indices omit, in stead of select elements. Negative and positive indices are not allowed in the same index vector. You can repeat a name or an index number, which results in multiple instances of the same value. You may check the above by predicting and then verifying the result of the following statements.

```
capColor["louie"]
names(capColor)[capColor == "blue"]
x <- c(4, 7, 6, 5, 2, 8)
I <- x < 6
J <- x > 7
x[I | J]
x[c(TRUE, FALSE)]
x[c(-1, -2)]
```

Replacing values in vectors can be done in the same way. For example, you may check that in the following assignment

```
x <- 1:10
x[c(TRUE, FALSE)] <- 1
```

every other value of `x` is replaced with `1`.

A `list` is a generalization of a vector in that it can contain objects of different types, including other lists. There are two ways to index a `list`. The single bracket operator always returns a `sub-list` of the indexed `list`. That is, the resulting type is again a `list`. The `double bracket` operator (`[[ ]]`) may only result in a single item, and it returns the object in the list itself. Besides indexing, the dollar operator `$` can be used to retrieve a single element. To understand the above, check the results of the following statements.

```
L <- list(x = c(1:5), y = c("a", "b", "c"), z = capColor)
L[[2]]
L$y
L[c(1, 3)]
L[c("x", "y")]
L[["z"]]
```

Especially, use the `class` function to determine the type of the result of each statement.

A `data.frame` is not much more than a `list` of vectors, possibly of different types, but with every vector (now columns) of the same length. Since `data.frames` are a type of list, indexing them with a single index returns a sub-`data.frame`; that is, a `data.frame` with less columns. Likewise, the dollar operator returns a vector, not a sub-`data.frame`. Rows can be indexed using two indices in the bracket operator, separated by a comma. The first index indicates rows, the second indicates columns. If one of the indices is left out, no selection is made (so everything is returned). It is important to realize that the result of a two-index selection is simplified by `R` as much as possible. Hence, selecting a single column using a two-index results in a vector. This behaviour may be switched off using `drop=FALSE` as an extra parameter. Here are some short examples demonstrating the above.

```
d <- data.frame(x = 1:10, y = letters[1:10], z = LETTERS[1:10])
d[1]
d[, 1]
d[, "x", drop = FALSE]
d[c("x", "z")]
d[d$x > 3, "y", drop = FALSE]
d[2, ]
```

1.2.2 Special values

Like most programming languages, `R` has a number of Special values that are exceptions to the normal values of a type. These are `NA`, `NULL`, ±`Inf` and `NaN`. Below, we quickly illustrate the meaning and differences between them.

`NA` Stands for *not available*. `NA` is a placeholder for a missing value. All basic operations in `R` handle `NA` without crashing and mostly return `NA` as an answer whenever one of the input arguments is `NA`. If you understand `NA`, you should be able to predict the result of the following `R` statements.

```
NA + 1
sum(c(NA, 1, 2))
median(c(NA, 1, 2, 3), na.rm = TRUE)
length(c(NA, 2, 3, 4))
3 == NA
NA == NA
TRUE | NA
```

The function `is.na` can be used to detect `NA`'s.

`NULL` You may think of `NULL` as the empty set from mathematics. `NULL` is special since it has no `class` (its class is `NULL`) and has length 0 so it does not take up any space in a vector. In particular, if you understand `NULL`, the result of the following statements should be clear to you without starting `R`.

```
length(c(1, 2, NULL, 4))
sum(c(1, 2, NULL, 4))
x <- NULL
c(x, 2)
```

The function `is.null` can be used to detect `NULL` variables.

`Inf` Stands for *infinity* and only applies to vectors of class `numeric`. A vector of class `integer` can never be `Inf`. This is because the `Inf` in `R` is directly derived from the international standard for floating point arithmetic[1]. Technically, `Inf` is a valid `numeric` that results from calculations like division of a number by zero. Since `Inf` is a numeric, operations between `Inf` and a finite numeric are well-defined and comparison operators work as expected. If you understand `Inf`, the result of the following statements should be clear to you.

```
pi/0
2 * Inf
Inf - 1e+10
Inf + Inf
3 < -Inf
Inf == Inf
```

`NaN` Stands for *not a number*. This is generally the result of a calculation of which the result is unknown, but it is surely not a number. In particular operations like `0/0`, `Inf-Inf` and `Inf/Inf` result in `NaN`. Technically, `NaN` is of class numeric, which may seem odd since it is used to indicate that something is *not* numeric. Computations involving numbers and `NaN` always result in `NaN`, so the result of the following computations should be clear.

```
NaN + 1
exp(NaN)
```

The function `is.nan` can be used to detect `NaN`'s.

Tip. *The function `is.finite` checks a vector for the occurrence of any non-numerical or special values. Note that it is not useful on `character` vectors.*

Exercises

Exercise 1.1. *Predict the result of the following R statements. Explain the reasoning behind the results.*

a. *`exp(-Inf)`*
b. *`NA == NA`*
c. *`NA == NULL`*
d. *`NULL == NULL`*
e. *`NA & FALSE`*

Exercise 1.2. *In which of the steps outlined in Figure 1 would you perform the following activities?*

a. *Estimating values for empty fields.*
b. *Setting the font for the title of a histogram.*
c. *Rewrite a column of categorical variables so that they are all written in capitals.*
d. *Use the `knitr` package[38] to produce a statistical report.*
e. *Exporting data from `Excel` to `csv`.*

2 From raw data to technically correct data

A data set is a collection of data that describes attribute values (variables) of a number of real-world objects (units). With data that are *technically correct*, we understand a data set where each value

1. can be directly recognized as belonging to a certain variable;
2. is stored in a data type that represents the value domain of the real-world variable.

In other words, for each unit, a text variable should be stored as text, a numeric variable as a number, and so on, and all this in a format that is consistent across the data set.

2.1 Technically correct data in R

The R environment is capable of reading and processing several file and data formats. For this tutorial we will limit ourselves to `rectangular' data sets that are to be read from a text-based format. In the case of R, we define *technically correct* data as a data set that

- is stored in a `data.frame` with suitable columns names, and
- each column of the `data.frame` is of the R type that adequately represents the value domain of the variable in the column.

The second demand implies that numeric data should be stored as `numeric` or `integer`, textual data should be stored as `character` and categorical data should be stored as a `factor` or `ordered` vector, with the appropriate levels.

Limiting ourselves to textual data formats for this tutorial may have its drawbacks, but there are several favorable properties of textual formats over binary formats:

- It is human-readable. When you inspect a text-file, make sure to use a text-reader (`more`, `less`) or editor (`Notepad`, `vim`) that uses a fixed-width font. Never use an office application for this purpose since typesetting clutters the data's structure, for example by the use of ligature.
- Text is very permissive in the types values that are stored, allowing for comments and annotations.

The task then, is to find ways to read a textfile into R and have it transformed to a well-typed `data.frame` with suitable column names.

Best practice. *Whenever you need to read data from a foreign file format, like a spreadsheet or proprietary statistical software that uses undisclosed file formats, make that software responsible for exporting the data to an open format that can be read by R.*

2.2 Reading text data into a R `data.frame`

In the following, we assume that the text-files we are reading contain data of at most one unit per line. The number of attributes, their format and separation symbols in lines containing data may differ over the lines. This includes files in fixed-width or `csv`-like format, but excludes `XML`-like storage formats.

2.2.1 read.table and its cousins

The following high-level R functions allow you to read in data that is technically correct, or close to it.

```
read.delim   read.delim2
read.csv     read.csv2
read.table   read.fwf
```

The return type of all these functions is a data.frame. If the column names are stored in the first line, they can automatically be assigned to the names attribute of the resulting data.frame.

Best practice. *A freshly read data.frame should always be inspected with functions like head, str, and summary.*

The read.table function is the most flexible function to read tabular data that is stored in a textual format. In fact, the other read-functions mentioned above all eventually use read.table with some fixed parameters and possibly after some preprocessing. Specifically

read.csv	for comma separated values with period as decimal separator.
read.csv2	for semicolon separated values with comma as decimal separator.
read.delim	tab-delimited files with period as decimal separator.
read.delim2	tab-delimited files with comma as decimal separator.
read.fwf	data with a predetermined number of bytes per column.

Each of these functions accept, amongst others, the following optional arguments.

Argument	description
header	Does the first line contain column names?
col.names	character vector with column names.
na.string	Which strings should be considered NA?
colClasses	character vector with the types of columns. Will coerce the columns to the specified types.
stringsAsFactors	If TRUE, converts all character vectors into factor vectors.
sep†	Field separator.

†Used only internally by read.fwf

Except for read.table and read.fwf, each of the above functions assumes by default that the first line in the text file contains column headers. To demonstrate this, we assume that we have the following text file stored under files/unnamed.txt.

```
21,6.0
42,5.9
18,5.7*
21,NA
```

Now consider the following script.

```
# first line is erroneously interpreted as column names
(person <- read.csv("files/unnamed.txt"))
##   X21 X6.0
## 1  42  5.9
## 2  18 5.7*
## 3  21 <NA>
# so we better do the following
person <- read.csv(
  file        = "files/unnamed.txt"
  , header    = FALSE
  , col.names = c("age","height") )
person
##   age height
## 1  21    6.0
## 2  42    5.9
## 3  18   5.7*
## 4  21   <NA>
```

In the first attempt, `read.csv` interprets the first line as column headers and fixes the numeric data to meet R's variable naming standards by prepending an X.

If `colClasses` is not specified by the user, `read.table` will try to determine the column types. Although this may seem convenient, it is noticeably slower for larger files (say, larger than a few MB) and it may yield unexpected results. For example, in the above script, one of the rows contains a malformed numerical variable (`5.7*`), causing R to interpret the whole column as a text variable. Moreover, by default text variables are converted to `factor`, so we are now stuck with a *height* variable expressed as levels in a categorical variable:

```
str(person)
## 'data.frame': 4 obs. of  2 variables:
##  $ age   : int  21 42 18 21
##  $ height: Factor w/ 3 levels "5.7*","5.9","6.0": 3 2 1 NA
```

Using `colClasses`, we can force R to either interpret the columns in the way we want or throw an error when this is not possible.

```
read.csv("files/unnamed.txt",
  header=FALSE,
  colClasses=c('numeric','numeric'))

## Error:  scan() expected 'a real', got '5.7*'
```

This behaviour is desirable if you need to be strict about how data is offered to your R script. However, unless you are prepared to write `tryCatch` constructions, a script containing the above code will stop executing completely when an error is encountered.

As an alternative, columns can be read in as `character` by setting `stringsAsFactors=FALSE`. Next, one of the `as.`-functions can be applied to convert to the desired type, as shown below.

```
dat <- read.csv(
  file              = "files/unnamed.txt"
  , header          = FALSE
  , col.names       = c("age","height")
  , stringsAsFactors=FALSE)
dat$height <- as.numeric(dat$height)

## Warning:  NAs introduced by coercion
```

```
dat
##   age height
## 1  21    6.0
## 2  42    5.9
## 3  18     NA
## 4  21     NA
```

Now, everything is read in and the `height` column is translated to numeric, with the exception of the row containing `5.7*`. Moreover, since we now get a warning instead of an error, a script containing this statement will continue to run, albeit with less data to analyse than it was supposed to. It is of course up to the programmer to check for these extra `NA`'s and handle them appropriately.

2.2.2 Reading data with `readLines`

When the rows in a data file are not uniformly formatted you can consider reading in the text line-by-line and transforming the data to a rectangular set yourself. With `readLines` you can exercise precise control over how each line is interpreted and transformed into fields in a rectangular data set. Table 1 gives an overview of the steps to be taken. Below, each step is discussed in more detail. As an example we will use a file called `daltons.txt`. Below, we show the contents of the file and the actual table with data as it should appear in `R`.

Data file:

```
%% Data on the Dalton Brothers
Gratt,1861,1892
Bob,1892
1871,Emmet,1937
% Names, birth and death dates
```

Actual table:

Name	Birth	Death
Gratt	1861	1892
Bob	NA	1892
Emmet	1871	1937

The file has comments on several lines (starting with a `%` sign) and a missing value in the second row. Moreover, in the third row the name and birth date have been swapped.

Step 1. Reading data. The `readLines` function accepts filename as argument and returns a `character` vector containing one element for each line in the file. `readLines` detects both the end-of-line and carriage return characters so lines are detected regardless of whether the file was created under `DOS`, `UNIX` or `MAC` (each OS has traditionally had different ways of marking an end-of-line). Reading in the Daltons file yields the following.

```
(txt <- readLines("files/daltons.txt"))
## [1] "%% Data on the Dalton Brothers" "Gratt,1861,1892"
## [3] "Bob,1892"                       "1871,Emmet,1937"
## [5] "% Names, birth and death dates"
```

The variable `txt` has 5 elements, equal to the number of lines in the textfile.

Step 2. Selecting lines containing data. This is generally done by throwing out lines containing comments or otherwise lines that do not contain any data fields. You can use `grep` or `grepl` to detect such lines.

```
# detect lines starting with a percentage sign..
I <- grepl("^%", txt)
# and throw them out
(dat <- txt[!I])
## [1] "Gratt,1861,1892" "Bob,1892"        "1871,Emmet,1937"
```

Table 1: Steps to take when converting lines in a raw text file to a `data.frame` *with correctly typed columns.*

	Step	result
1	Read the data with `readLines`	`character`
2	Select lines containing data	`character`
3	Split lines into separate fields	`list` of `character` vectors
4	Standardize rows	`list` of equivalent vectors
5	Transform to `data.frame`	`data.frame`
6	Normalize and coerce to correct type	`data.frame`

Here, the first argument of `grepl` is a search pattern, where the caret ()ˆ indicates a start-of-line. The result of `grepl` is a `logical` vector that indicates which elements of `txt` contain the pattern 'start-of-line' followed by a percent-sign. The functionality of `grep` and `grepl` will be discussed in more detail in section 2.4.2.

Step 3. Split lines into separate fields. This can be done with `strsplit`. This function accepts a `character` vector and a `split` argument which tells `strsplit` how to split a string into substrings. The result is a `list` of `character` vectors.

```
(fieldList <- strsplit(dat, split = ","))
## [[1]]
## [1] "Gratt" "1861"  "1892"
##
## [[2]]
## [1] "Bob"  "1892"
##
## [[3]]
## [1] "1871"  "Emmet" "1937"
```

Here, `split` is a single character or sequence of characters that are to be interpreted as field separators. By default, `split` is interpreted as a regular expression (see Section 2.4.2) which means you need to be careful when the split argument contains any of the special characters listed on page 25. The meaning of these special characters can be ignored by passing `fixed=TRUE` as extra parameter.

Step 4. Standardize rows. The goal of this step is to make sure that 1) every row has the same number of fields and 2) the fields are in the right order. In `read.table`, lines that contain less fields than the maximum number of fields detected are appended with `NA`. One advantage of the do-it-yourself approach shown here is that we do not have to make this assumption. The easiest way to standardize rows is to write a function that takes a single `character` vector as input and assigns the values in the right order.

```
assignFields <- function(x){
  out <- character(3)
  # get names
  i <- grepl("[[:alpha:]]",x)
  out[1] <- x[i]
  # get birth date (if any)
  i <- which(as.numeric(x) < 1890)
  out[2] <- ifelse(length(i)>0, x[i], NA)
  # get death date (if any)
  i <- which(as.numeric(x) > 1890)
  out[3] <- ifelse(length(i)>0, x[i], NA)
  out
}
```

The above function accepts a character vector and assigns three values to an output vector of class `character`. The `grepl` statement detects fields containing alphabetical values `a-z` or `A-Z`. To assign year of birth and year of death, we use the knowledge that all Dalton brothers were born before and died after 1890. To retrieve the fields for each row in the example, we need to apply this function to every element of `fieldList`.

```
standardFields <- lapply(fieldList, assignFields)
standardFields
## [[1]]
## [1] "Gratt" "1861"  "1892"
##
## [[2]]
## [1] "Bob"   NA     "1892"
##
## [[3]]
## [1] "Emmet" "1871"  "1937"
```

Here, we suppressed the warnings about failed conversions that R generates in the output.

The advantage of this approach is having greater flexibility than `read.table` offers. However, since we are interpreting the value of fields here, it is unavoidable to know about the contents of the dataset which makes it hard to generalize the field assigner function. Furthermore, `assignFields` function we wrote is still relatively fragile. That is: it crashes for example when the input vector contains two or more text-fields or when it contains more than one numeric value larger than 1890. Again, no one but the data analyst is probably in a better position to choose how safe and general the field assigner should be.

Tip. *Element-wise operations over lists are easy to parallelize with the `parallel` package that comes with the standard R installation. For example, on a quadcore computer you can do the following.*

```
library(parallel)
cluster <- makeCluster(4)
standardFields <- parLapply(cl=cluster, fieldList, assignFields)
stopCluster(cl)
```

Of course, parallelization only makes sense when you have a fairly long list to process, since there is some overhead in setting up and running the cluster.

Step 5. Transform to `data.frame`. There are several ways to transform a list to a `data.frame` object. Here, first all elements are copied into a `matrix` which is then coerced into a `data.frame`.

```
(M <- matrix(
  unlist(standardFields)
  , nrow=length(standardFields)
  , byrow=TRUE))
##      [,1]    [,2]   [,3]
## [1,] "Gratt" "1861" "1892"
## [2,] "Bob"   NA     "1892"
## [3,] "Emmet" "1871" "1937"
colnames(M) <- c("name","birth","death")
(daltons <- as.data.frame(M, stringsAsFactors=FALSE))
##    name birth death
## 1 Gratt  1861  1892
## 2   Bob  <NA>  1892
## 3 Emmet  1871  1937
```

The function `unlist` concatenates all vectors in a `list` into one large character vector. We then use that vector to fill a `matrix` of class `character`. However, the `matrix` function usually fills up a matrix column by column. Here, our data is stored with rows concatenated, so we need to add the argument `byrow=TRUE`. Finally, we add column names and coerce the matrix to a `data.frame`. We use `stringsAsFactors=FALSE` since we have not started interpreting the values yet.

Step 6. Normalize and coerce to correct types.

This step consists of preparing the `character` columns of our `data.frame` for coercion and translating numbers into `numeric` vectors and possibly `character` vectors to `factor` variables. String normalization is the subject of section 2.4.1 and type conversion is discussed in some more detail in the next section. However, in our example we can suffice with the following statements.

```
daltons$birth <- as.numeric(daltons$birth)
daltons$death <- as.numeric(daltons$death)
daltons
##    name birth death
## 1 Gratt  1861  1892
## 2   Bob    NA  1892
## 3 Emmet  1871  1937
```

Or, using `transform`:

```
daltons = transform( daltons
                   , birth = as.numeric(birth)
                   , death = as.numeric(death)
                   )
```

2.3 Type conversion

Converting a variable from one type to another is called *coercion*. The reader is probably familiar with R's basic coercion functions, but as a reference they are listed here.

```
as.numeric    as.logical
as.integer    as.factor
as.character  as.ordered
```

Each of these functions takes an R object and tries to convert it to the class specified behind the ``as.". By default, values that cannot be converted to the specified type will be converted to a NA value while a warning is issued.

```
as.numeric(c("7", "7*", "7.0", "7,0"))

## Warning:  NAs introduced by coercion

## [1]  7 NA  7 NA
```

In the remainder of this section we introduce R's typing and storage system and explain the difference between R *types* and *classes*. After that we discuss date conversion.

2.3.1 Introduction to R's typing system

Everything in R is an object[4]. An object is a container of data endowed with a label describing the data. Objects can be created, destroyed or overwritten on-the-fly by the user.

The function `class` returns the class label of an R object.

```
class(c("abc", "def"))
## [1] "character"
class(1:10)
## [1] "integer"
class(c(pi, exp(1)))
## [1] "numeric"
class(factor(c("abc", "def")))
## [1] "factor"
```

Tip. *Here's a quick way to retrieve the classes of all columns in a `data.frame` called `dat`.*

```
sapply(dat, class)
```

For the user of R these class labels are usually enough to handle R objects in R scripts. Under the hood, the basic R objects are stored as C structures as C is the language in which R itself has been written. The type of C structure that is used to store a basic type can be found with the `typeof` function. Compare the results below with those in the previous code snippet.

```
typeof(c("abc", "def"))
## [1] "character"
typeof(1:10)
## [1] "integer"
typeof(c(pi, exp(1)))
## [1] "double"
typeof(factor(c("abc", "def")))
## [1] "integer"
```

Note that the `type` of an R object of class `numeric` is `double`. The term `double` refers to `double precision`, which is a standard way for lower-level computer languages such as C to store approximations of real numbers. Also, the `type` of an object of class `factor` is `integer`. The reason is that R saves memory (and computational time!) by storing factor values as integers, while a translation table between factor and integers are kept in memory. Normally, a user should not have to worry about these subtleties, but there are exceptions. An example of this is the subject of Exercise 2.2.

In short, one may regard the *class* of an object as the object's type from the user's point of view while the *type* of an object is the way R looks at the object. It is important to realize that R's coercion functions are fundamentally functions that change the underlying `type` of an object and that class changes are a consequence of the type changes.

Confusingly, R objects also have a *mode* (and *storage.mode*) which can be retrieved or set using functions of the same name. Both `mode` and `storage.mode` differ slightly from `typeof`, and are only there for backwards compatibility with R's precursor language: S. We therefore advise the user to avoid using these functions to retrieve or modify an object's type.

2.3.2 Recoding factors

In `R`, the value of categorical variables is stored in `factor` variables. A `factor` is an integer vector endowed with a table specifying what integer value corresponds to what `level`. The values in this translation table can be requested with the `levels` function.

```
f <- factor(c("a", "b", "a", "a", "c"))
levels(f)
## [1] "a" "b" "c"
```

The use of integers combined with a translation table is not uncommon in statistical software, so chances are that you eventually have to make such a translation by hand. For example, suppose we read in a vector where 1 stands for `male`, 2 stands for `female` and 0 stands for unknown. Conversion to a factor variable can be done as in the example below.

```
# example:
gender <- c(2, 1, 1, 2, 0, 1, 1)
# recoding table, stored in a simple vector
recode <- c(male = 1, female = 2)
(gender <- factor(gender, levels = recode, labels = names(recode)))
## [1] female male   male   female <NA>   male   male
## Levels: male female
```

Note that we do not explicitly need to set `NA` as a label. Every integer value that is encountered in the first argument, but not in the `levels` argument will be regarded missing.

Levels in a factor variable have no natural ordering. However in multivariate (regression) analyses it can be beneficial to fix one of the levels as the reference level. `R`'s standard multivariate routines (`lm`, `glm`) use the first level as reference level. The `relevel` function allows you to determine which level comes first.

```
(gender <- relevel(gender, ref = "female"))
## [1] female male   male   female <NA>   male   male
## Levels: female male
```

Levels can also be reordered, depending on the mean value of another variable, for example:

```
age <- c(27, 52, 65, 34, 89, 45, 68)
(gender <- reorder(gender, age))
## [1] female male   male   female <NA>   male   male
## attr(,"scores")
## female   male
##   30.5   57.5
## Levels: female male
```

Here, the means are added as a named vector attribute to `gender`. It can be removed by setting that attribute to `NULL`.

```
attr(gender, "scores") <- NULL
gender
## [1] female male   male   female <NA>   male   male
## Levels: female male
```

2.3.3 Converting dates

The base `R` installation has three types of objects to store a time instance: `Date`, `POSIXlt` and `POSIXct`. The `Date` object can only be used to store dates, the other two store date and/or time. Here, we focus on converting text to `POSIXct` objects since this is the most portable way to store such information.

Under the hood, a `POSIXct` object stores the number of seconds that have passed since January 1, 1970 00:00. Such a storage format facilitates the calculation of durations by subtraction of two `POSIXct` objects.

When a `POSIXct` object is printed, `R` shows it in a human-readable calender format. For example, the command `Sys.time` returns the system time provided by the operating system in `POSIXct` format.

```
current_time <- Sys.time()
class(current_time)
## [1] "POSIXct" "POSIXt"
current_time
## [1] "2013-10-28 11:12:50 CET"
```

Here, `Sys.time` uses the time zone that is stored in the `locale` settings of the machine running `R`.

Converting from a calender time to `POSIXct` and back is not entirely trivial, since there are many idiosyncrasies to handle in calender systems. These include leap days, leap seconds, daylight saving times, time zones and so on. Converting from text to `POSIXct` is further complicated by the many textual conventions of time/date denotation. For example, both `28 September 1976` and `1976/09/28` indicate the same day of the same year. Moreover, the name of the month (or weekday) is language-dependent, where the language is again defined in the operating system's locale settings.

The `lubridate` package[13] contains a number of functions facilitating the conversion of text to `POSIXct` dates. As an example, consider the following code.

```
library(lubridate)
dates <- c("15/02/2013", "15 Feb  13", "It happened on 15 02 '13")
dmy(dates)
## [1] "2013-02-15 UTC" "2013-02-15 UTC" "2013-02-15 UTC"
```

Here, the function `dmy` assumes that dates are denoted in the order day-month-year and tries to extract valid dates. Note that the code above will only work properly in locale settings where the name of the second month is abbreviated to *Feb*. This holds for English or Dutch locales, but fails for example in a French locale (*Février*).

There are similar functions for all permutations of `d`, `m` and `y`. Explicitly, all of the following functions exist.

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

So once it is known in what order days, months and years are denoted, extraction is very easy.

Note. *It is not uncommon to indicate years with two numbers, leaving out the indication of century. In R, 00-68 are interpreted as 2000-2068 and 69-99 as 1969-1999.*

```
dmy("01 01 68")
## [1] "2068-01-01 UTC"
dmy("01 01 69")
## [1] "1969-01-01 UTC"
```

Table 2: Day, month and year formats recognized by R.

Code	description	Example
%a	Abbreviated weekday name in the current locale.	Mon
%A	Full weekday name in the current locale.	Monday
%b	Abbreviated month name in the current locale.	Sep
%B	Full month name in the current locale.	September
%m	Month number (01-12)	09
%d	Day of the month as decimal number (01-31).	28
%y	Year without century (00-99)	13
%Y	Year including century.	2013

This behaviour is according to the 2008 POSIX standard, but one should expect that this interpretation changes over time.

It should be noted that lubridate (as well as R's base functionality) is only capable of converting certain standard notations. For example, the following notation does not convert.

```
dmy("15 Febr. 2013")

## Warning: All formats failed to parse. No formats found.

## [1] NA
```

The standard notations that can be recognized by R, either using lubridate or R's built-in functionality are shown in Table 2. Here, the names of (abbreviated) week or month names that are sought for in the text depend on the locale settings of the machine that is running R. For example, on a PC running under a Dutch locale, ``maandag" will be recognized as the first day of the week while in English locales ``Monday" will be recognized. If the machine running R has multiple locales installed you may add the argument locale to one of the dmy-like functions. In Linux-alike systems you can use the command locale -a in bash terminal to see the list of installed locales. In Windows you can find available locale settings under ``language and regional settings", under the configuration screen.

If you know the textual format that is used to describe a date in the input, you may want to use R's core functionality to convert from text to POSIXct. This can be done with the as.POSIXct function. It takes as arguments a character vector with time/date strings and a string describing the format.

```
dates <- c("15-9-2009", "16-07-2008", "17 12-2007", "29-02-2011")
as.POSIXct(dates, format = "%d-%m-%Y")
## [1] "2009-09-15 CEST" "2008-07-16 CEST" NA                NA
```

In the format string, date and time fields are indicated by a letter preceded by a percent sign (%). Basically, such a %-code tells R to look for a range of substrings. For example, the %d indicator makes R look for numbers 1-31 where precursor zeros are allowed, so 01, 02,...31 are recognized as well. Table 2 shows which date-codes are recognized by R. The complete list can be found by typing ?strptime in the R console. Strings that are not in the exact format specified by the format argument (like the third string in the above example) will not be converted by as.POSIXct. Impossible dates, such as the leap day in the fourth date above are also not converted.

Finally, to convert dates from POSIXct back to character, one may use the format function that comes with base R. It accepts a POSIXct date/time object and an output format string.

```
mybirth <- dmy("28 Sep 1976")
format(mybirth, format = "I was born on %B %d, %Y")
## [1] "I was born on September 28, 1976"
```

2.4 `character` manipulation

Because of the many ways people can write the same things down, character data can be difficult to process. For example, consider the following excerpt of a data set with a `gender` variable.

```
##   gender
## 1      M
## 2  male
## 3 Female
## 4   fem.
```

If this would be treated as a factor variable without any preprocessing, obviously four, not two classes would be stored. The job at hand is therefore to automatically recognize from the above data whether each element pertains to `male` or `female`. In statistical contexts, classifying such ``messy" text strings into a number of fixed categories is often referred to as *coding*.

Below we discuss two complementary approaches to string coding: *string normalization* and *approximate text matching*. In particular, the following topics are discussed.

- Remove prepending or trailing white spaces.
- Pad strings to a certain width.
- Transform to upper/lower case.
- Search for strings containing simple patterns (substrings).
- Approximate matching procedures based on string distances.

2.4.1 String normalization

String normalization techniques are aimed at transforming a variety of strings to a smaller set of string values which are more easily processed. By default, `R` comes with extensive string manipulation functionality that is based on the two basic string operations: *finding* a pattern in a string and *replacing* one pattern with another. We will deal with `R`'s generic functions below but start by pointing out some common string cleaning operations.

The `stringr` package[36] offers a number of functions that make some some string manipulation tasks a lot easier than they would be with `R`'s base functions. For example, extra white spaces at the beginning or end of a string can be removed using `str_trim`.

```
library(stringr)
str_trim("  hello world ")
## [1] "hello world"
str_trim("  hello world ", side = "left")
## [1] "hello world "
str_trim("  hello world ", side = "right")
## [1] "  hello world"
```

Conversely, strings can be padded with spaces or other characters with `str_pad` to a certain width. For example, numerical codes are often represented with prepending zeros.

```
str_pad(112, width = 6, side = "left", pad = 0)
## [1] "000112"
```

Both `str_trim` and `str_pad` accept a `side` argument to indicate whether trimming or padding should occur at the beginning (`left`), end (`right`) or both sides of the string.

Converting strings to complete upper or lower case can be done with `R`'s built-in `toupper` and `tolower` functions.

```
toupper("Hello world")
## [1] "HELLO WORLD"
tolower("Hello World")
## [1] "hello world"
```

2.4.2 Approximate string matching

There are two forms of string matching. The first consists of determining whether a (range of) substring(s) occurs within another string. In this case one needs to specify a range of substrings (called a *pattern*) to search for in another string. In the second form one defines a distance metric between strings that measures how ``different" two strings are. Below we will give a short introduction to pattern matching and string distances with `R`.

There are several pattern matching functions that come with base `R`. The most used are probably `grep` and `grepl`. Both functions take a pattern and a `character` vector as input. The output only differs in that `grepl` returns a logical index, indicating which element of the input `character` vector contains the pattern, while `grep` returns a numerical index. You may think of `grep(...)` as `which(grepl(...))`.

In the most simple case, the pattern to look for is a simple substring. For example, using the data of the example on page 23, we get the following.

```
gender <- c("M", "male ", "Female", "fem.")
grepl("m", gender)
## [1] FALSE  TRUE  TRUE  TRUE
grep("m", gender)
## [1] 2 3 4
```

Note that the result is case sensitive: the capital `M` in the first element of `gender` does not match the lower case `m`. There are several ways to circumvent this case sensitivity. Either by case normalization or by the optional argument `ignore.case`.

```
grepl("m", gender, ignore.case = TRUE)
## [1] TRUE TRUE TRUE TRUE
grepl("m", tolower(gender))
## [1] TRUE TRUE TRUE TRUE
```

Obviously, looking for the occurrence of `m` or `M` in the `gender` vector does not allow us to determine which strings pertain to `male` and which not. Preferably we would like to search for strings that start with an `m` or `M`. Fortunately, the search patterns that `grep` accepts allow for such searches. The beginning of a string is indicated with a caret (^).

```
grepl("^m", gender, ignore.case = TRUE)
## [1]  TRUE  TRUE FALSE FALSE
```

Indeed, the grepl function now finds only the first two elements of gender. The caret is an example of a so-called *meta-character*. That is, it does not indicate the caret itself but something else, namely the beginning of a string. The search patterns that grep, grepl (and sub and gsub) understand have more of these meta-characters, namely:

```
. \ | ( ) [ {  ^ $ * + ?
```

If you need to search a string for any of these characters, you can use the option fixed=TRUE.

```
grepl("^", gender, fixed = TRUE)
## [1] FALSE FALSE FALSE FALSE
```

This will make grepl or grep ignore any meta-characters in the search string.

Search patterns using meta-characters are called *regular expressions*. Regular expressions offer powerful and flexible ways to search (and alter) text. A discussion of regular expressions is beyond the scope of these lecture notes. However, a concise description of regular expressions allowed by R's built-in string processing functions can be found by typing ?regex at the R command line. The books by Fitzgerald[10] or Friedl[11] provide a thorough introduction to the subject of regular expression. If you frequently have to deal with ``messy'' text variables, learning to work with regular expressions is a worthwhile investment. Moreover, since many popular programming languages support some dialect of regexps, it is an investment that could pay off several times.

We now turn our attention to the second method of approximate matching, namely string distances. A string distance is an algorithm or equation that indicates how much two strings differ from each other. An important distance measure is implemented by the R's native adist function. This function counts how many basic operations are needed to turn one string into another. These operations include insertion, deletion or substitution of a single character[19]. For example

```
adist("abc", "bac")
##      [,1]
## [1,]    2
```

The result equals two since turning "abc" into "bac" involves two character substitutions:

abc→bbc→bac.

Using adist, we can compare fuzzy text strings to a list of known codes. For example:

```
codes <- c("male", "female")
D <- adist(gender, codes)
colnames(D) <- codes
rownames(D) <- gender
D
##        male female
## M         4      6
## male      1      3
## Female    2      1
## fem.      4      3
```

Here, adist returns the distance matrix between our vector of fixed codes and the input data. For readability we added row- and column names accordingly. Now, to find out which code matches best with our raw data, we need to find the index of the smallest distance for each row of D. This can be done as follows.

```
i <- apply(D, 1, which.min)
data.frame(rawtext = gender, coded = codes[i])
##   rawtext  coded
## 1       M   male
## 2   male    male
## 3  Female female
## 4    fem. female
```

We use apply to apply which.min to every row of D. Note that in the case of multiple minima, the first match will be returned. At the end of this subsection we show how this code can be simplified with the stringdist package.

Finally, we mention three more functions based on string distances. First, the R-built-in function agrep is similar to grep, but it allows one to specify a maximum Levenshtein distance between the input pattern and the found substring. The agrep function allows for searching for regular expression patterns, which makes it very flexible.

Secondly, the stringdist package[32] offers a function called stringdist which can compute a variety of string distance metrics, some of which are likely to provide results that are better than adist's. Most importantly, the distance function used by adist does not allow for character transpositions, which is a common typographical error. Using the *optimal string alignment distance* (the default choice for stringdist) we get

```
library(stringdist)
stringdist("abc", "bac")
## [1] 1
```

The answer is now 1 (not 2 as with adist), since the optimal string alignment distance allows for transpositions of adjacent characters:

abc→ bac.

Thirdly, the stringdist package provides a function called amatch, which mimics the behaviour of R's match function: it returns an index to the closest match within a maximum distance. Recall the gender and code example of page 25.

```
# this yields the closest match of 'gender' in 'codes' (within a distance of 4)
(i <- amatch(gender,codes,maxDist=4))
## [1] 1 1 2 2

# store results in a data.frame
data.frame(
  rawtext = gender
  , code = codes[i]
  )
##   rawtext   code
## 1       M   male
## 2   male    male
## 3  Female female
## 4    fem. female
```

2.5 Character encoding issues

A *character encoding system* is a system that defines how to translate each character of a given alphabet into a computer byte or sequence of bytes[†]. For example, ASCII is an encoding

[†] In fact, the definition can be more general, for example to include Morse code. However, we limit ourselves to computerized character encodings.

system that prescribes how to translate 127 characters into single bytes (where the first bit of each byte is necessarily 0). The ASCII characters include the upper and lower case letters of the Latin alphabet (a-z, A-Z), Arabic numbers (0-9), a number of punctuation characters and a number of invisible so-called control characters such as newline and carriage return.

Although it is widely used, ASCII is obviously incapable of encoding characters outside the Latin alphabet, so you can say ``hello", but not ``*γεια σας*" in this encoding. For this reason, a number of character encoding systems have been developed that extend ASCII or replace it all together. Some well-known schemes include UTF-8 and latin1. The character encoding scheme that is used by default by your operating system is defined in your locale settings. Most Unix-alikes use UTF-8 by default while older Windows applications, including the Windows version of R use latin1. The UTF-8 encoding standard is widely used to encode web pages: according to a frequently repeated survey of w3techs[35], about 75% of the 10 million most visited web pages are encoded in UTF-8.

You can find out the character encoding of your system by typing (not copy-pasting!) a non-ASCII character and ask for the encoding scheme, like so.

```
Encoding("Queensrÿche")
## [1] "unknown"
```

If the answer is "unknown", this means that the local native encoding is used. The default encoding used by your OS can be requested by typing

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

at the R command-line.

For R to be able to correctly read in a textfile, it must understand which character encoding scheme was used to store it. By default, R assumes that a textfile is stored in the encoding scheme defined by the operating system's locale setting. This may fail when the file was not generated on the same computer that R is running on but was obtained from the web for example. To make things worse, it is impossible to determine automatically with certainty from a file what encoding scheme has been used (although for some encodings it is possible). This means that you may run into situations where you have to tell R literally in which encoding a file has been stored. Once a file has been read into R, a character vector will internally be translated to either UTF-8 or latin1.

The fileEncoding argument of read.table and its relatives tells R what encoding scheme was used to store the file. For readLines the file encoding must be specified when the file is opened, before calling readLines, as in the example below.

```
# 1. open a connection to your file, specifying its encoding
f <- file("myUTF16file.txt", encoding = "UTF-16")
# 2. Read the data with readLines.  Text read from the file is converted to
# uft8 or latin1
input <- readLines(f)
# close the file connection.
close(f)
```

When reading the file, R will not translate the encoding to UTF-8 or latin1 by itself, but instead relies on an external iconv library. Depending on the operating system, R either uses the conversion service offered by the OS, or uses a third-party library included with R. R's iconv function allows users to translate character representations, because of the OS-dependencies just mentioned, not all translations will be possible on all operating systems. With `iconvlist()` you can check what encodings can be translated by your operating system. The only encoding systems that is guaranteed to be available on all platforms are `UTF-8` and `latin1`.

Exercises

Exercise 2.1. *Type conversions.*

a. *Load the built in* `warpbreaks` *data set . Find out, in a single command, which columns of* `warpbreaks` *are either* `numeric` *or* `integer`*.*

b. *Is* `numeric` *a natural data type for the columns which are stored as such? Convert to integer when necessary. (See also* `?warpbreaks` *for an explanation of the data).*

c. *Error messages in R sometimes report the underlying* type *of an object rather than the user-level* class*. Derive from the following code and error message what the underlying type of an R function is.*

```
mean[1]
## Error: object of type 'closure' is not subsettable
```

Confirm your answer using `typeof`*.*

Exercise 2.2. *Type the following code in your R terminal.*

```
v <- factor(c("2", "3", "5", "7", "11"))
```

a. *Convert v to character with* `as.character`*. Explain what just happened.*

b. *Convert v to numeric with* `as.numeric`*. Explain what just happened.*

c. *How would you convert the values of v to integers?*

Exercise 2.3. *In this exercise we'll use* `readLines` *to read in an irregular textfile. The file looks like this (without numbering).*

```
1  // Survey data. Created : 21 May 2013
2  // Field 1: Gender
3  // Field 2: Age (in years)
4  // Field 3: Weight (in kg)
5  M;28;81.3
6  male;45;
7  Female;17;57,2
8  fem.;64;62.8
```

You may copy the text from this pdf file in a textfile called `example.txt` *or download the file from our Github page.*

a. *Read the complete file using* `readLines`*.*

b. *Separate the* `vector` *of lines into a* `vector` *containing comments and a* `vector` *containing the data. Hint: use* `grepl`*.*

c. *Extract the date from the first comment line.*

d. *Read the data into a* `matrix` *as follows.*
   (a) *Split the character vectors in the vector containing* `data` *lines by semicolon (*`;`*) using* `strsplit`*.*
   (b) *Find the maximum number of fields retrieved by* `split`*. Append rows that are shorter with NA's.*
   (c) *Use* `unlist` *and* `matrix` *to transform the data to row-column format.*

e. *From comment lines* `2-4`*, extract the names of the fields. Set these as* `colnames` *for the* `matrix` *you just created.*

Exercise 2.4. *We will coerce the columns of the data of the previous exercise to a structured data set.*

a. *Coerce the matrix to a* `data.frame`*, making sure all columns are* `character` *columns.*

b. Use a string distance technique to transform the `Gender` column into a `factor` variable with labels `man` and `woman`.

c. Coerce the `Age` column to `integer`.

d. Coerce the `weight` column to `numeric`. Hint: use `gsub` to replace comma's with a period.

3 From technically correct data to consistent data

Consistent data are *technically correct data* that are fit for statistical analysis. They are data in which missing values, special values, (obvious) errors and outliers are either removed, corrected or imputed. The data are consistent with constraints based on real-world knowledge about the subject that the data describe.

Consistency can be understood to include *in-record* consistency, meaning that no contradictory information is stored in a single record, and *cross-record* consistency, meaning that statistical summaries of different variables do not conflict with each other. Finally, one can include cross-dataset consistency, meaning that the dataset that is currently analyzed is consistent with other datasets pertaining to the same subject matter. In this tutorial we mainly focus on methods dealing with in-record consistency, with the exception of outlier handling which can be considered a cross-record consistency issue.

The process towards consistent data always involves the following three steps.

1. *Detection* of an inconsistency. That is, one establishes which constraints are violated. For example, an *age* variable is constrained to non-negative values.
2. *Selection* of the field or fields causing the inconsistency. This is trivial in the case of a univariate demand as in the previous step, but may be more cumbersome when cross-variable relations are expected to hold. For example the *marital status* of a child must be `unmarried`. In the case of a violation it is not immediately clear whether *age*, *marital status* or both are wrong.
3. *Correction* of the fields that are deemed erroneous by the selection method. This may be done through deterministic (model-based) or stochastic methods.

For many data correction methods these steps are not necessarily neatly separated. For example, in the deductive correction methods described in subsection 3.2.2 below, the three steps are performed with a single mathematical operation. Nevertheless, it is useful to be able to recognize these sub-steps in order to make clear what assumptions are made during the data cleaning process.

In the following subsection (3.1) we introduce a number of techniques dedicated to the detection of errors and the selection of erroneous fields. If the field selection procedure is performed separately from the error detection procedure, it is generally referred to as *error localization*. The latter is described in subsection 3.1.5. Subsection 3.2 describes techniques that implement correction methods based on \`direct rules' or \`deductive correction'. In these techniques, erroneous values are replaced by better ones by directly deriving them from other values in the same record. Finally, subsection 3.3 gives an overview of some commonly used imputation techniques that are available in R.

3.1 Detection and localization of errors

This section details a number of techniques to detect univariate and multivariate constraint violations. Special attention is paid tot the error localization problem in subsection 3.1.5.

3.1.1 Missing values

A missing value, represented by NA in R, is a placeholder for a datum of which the type is known but its value isn't. Therefore, it is impossible to perform statistical analysis on data where one or more values in the data are missing. One may choose to either omit elements from a dataset that contain missing values or to impute a value, but missingness is something to be dealt with prior to any analysis.

In practice, analysts, but also commonly used numerical software may confuse a missing value with a default value or category. For instance, in `Excel 2010`, the result of adding the contents of a field containing the number 1 with an empty field results in 1. This behaviour is most definitely unwanted since `Excel` silently imputes `0' where it should have said something along the lines of `unable to compute'. It should be up to the analyst to decide how empty values are handled, since a default imputation may yield unexpected or erroneous results for reasons that are hard to trace. Another commonly encountered mistake is to confuse an `NA` in categorical data with the category *unknown*. If *unknown* is indeed a category, it should be added as a `factor` level so it can be appropriately analyzed. Consider as an example a categorical variable representing *place of birth*. Here, the category *unknown* means that we have no knowledge about where a person is born. In contrast, `NA` indicates that we have no information to determine whether the birth place is known or not.

The behaviour of `R`'s core functionality is completely consistent with the idea that the analyst must decide what to do with missing data. A common choice, namely `leave out records with missing data' is supported by many base functions through the `na.rm` option.

```
age <- c(23, 16, NA)
mean(age)
## [1] NA
mean(age, na.rm = TRUE)
## [1] 19.5
```

Functions such as `sum`, `prod`, `quantile`, `sd` and so on all have this option. Functions implementing bivariate statistics such as `cor` and `cov` offer options to include complete or pairwise complete values.

Besides the `is.na` function, that was already mentioned in section 1.2.2, `R` comes with a few other functions facilitating `NA` handling. The `complete.cases` function detects rows in a `data.frame` that do not contain any missing value. Recall the `person` data set example from page 14.

```
print(person)
##   age height
## 1  21    6.0
## 2  42    5.9
## 3  18   5.7*
## 4  21   <NA>
complete.cases(person)
## [1]  TRUE  TRUE  TRUE FALSE
```

The resulting `logical` can be used to remove incomplete records from the `data.frame`. Alternatively the `na.omit` function, does the same.

```
(persons_complete <- na.omit(person))
##   age height
## 1  21    6.0
## 2  42    5.9
## 3  18   5.7*
na.action(persons_complete)
## 4
## 4
## attr(,"class")
## [1] "omit"
```

The result of the `na.omit` function is a `data.frame` where incomplete rows have been deleted. The `row.names` of the removed records are stored in an attribute called `na.action`.

Note. *It may happen that a missing value in a data set means* 0 *or* `Not applicable`*. If that is the case, it should be explicitly imputed with that value, because it is not unknown, but was coded as empty.*

3.1.2 Special values

As explained in section 1.2.2, numeric variables are endowed with several formalized special values including ±`Inf`, `NA` and `NaN`. Calculations involving special values often result in special values, and since a statistical statement about a real-world phenomenon should never include a special value, it is desirable to handle special values prior to analysis.

For numeric variables, special values indicate values that are not an element of the mathematical set of real numbers ($\mathbb{R}$). The function `is.finite` determines which values are `regular' values.

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

This function accepts vectorial input. With little effort we can write a function that may be used to check every numerical column in a data.frame

```
is.special <- function(x){
  if (is.numeric(x)) !is.finite(x) else is.na(x)
}
person
##   age height
## 1  21    6.0
## 2  42    5.9
## 3  18   5.7*
## 4  21   <NA>
sapply(person, is.special)
##        age height
## [1,] FALSE  FALSE
## [2,] FALSE  FALSE
## [3,] FALSE  FALSE
## [4,] FALSE   TRUE
```

Here, the `is.special` function is applied to each column of `person` using `sapply`. `is.special` checks its input vector for numerical special values if the type is numeric, otherwise it only checks for `NA`.

3.1.3 Outliers

There is a vast body of literature on outlier detection, and several definitions of *outlier* exist. A general definition by Barnett and Lewis[2] defines an outlier in a data set as *an observation (or set of observations) which appear to be inconsistent with that set of data*. Although more precise definitions exist (see e.g. the book by Hawkins[15]), this definition is sufficient for the current tutorial. Below we mention a few fairly common graphical and computational techniques for outlier detection in univariate numerical data.

Note. *Outliers do not equal errors. They should be detected, but not necessarily removed. Their inclusion in the analysis is a statistical decision.*

For more or less unimodal and symmetrically distributed data, Tukey's box-and-whisker method[29] for outlier detection is often appropriate. In this method, an observation is an outlier when it is larger than the so-called ``whiskers" of the set of observations. The upper whisker is computed by adding 1.5 times the interquartile range to the third quartile and rounding to the nearest lower observation. The lower whisker is computed likewise.

The base R installation comes with function `boxplot.stats`, which, amongst other things, list the outliers.

```
x <- c(1:10, 20, 30)
boxplot.stats(x)$out
## [1] 20 30
```

Here, 20 and 50 are detected as outliers since they are above the upper whisker of the observations in `x`. The factor 1.5 used to compute the whisker is to an extent arbitrary and it can be altered by setting the `coef` option of `boxplot.stats`. A higher coefficient means a higher outlier detection limit (so for the same dataset, generally less upper or lower outliers will be detected).

```
boxplot.stats(x, coef = 2)$out
## [1] 30
```

The box-and-whisker method can be visualized with the box-and-whisker plot, where the box indicates the interquartile range and the median, the whiskers are represented at the ends of the box-and-whisker plots and outliers are indicated as separate points above or below the whiskers.

The box-and-whisker method fails when data are distributed skewly, as in an exponential or log-normal distribution for example. In that case one can attempt to transform the data, for example with a logarithm or square root transformation. Another option is to use a method that takes the skewness into account.

A particularly easy-to-implement example is the method of Hiridoglou and Berthelot[16] for positive observations. In this method, an observation is an outlier when

$$h(x) = \max\left(\frac{x}{x^*}, \frac{x^*}{x}\right) \geq r, \text{ and } x > 0. \qquad (1)$$

Here, r is a user-defined reference value and x^* is usually the median observation, although other measures of centrality may be chosen. Here, the score function $h(x)$ grows as $1/x$ as x approaches zero and grows linearly with x when it is larger than x^*. It is therefore appropriate for finding outliers on both sides of the distribution. Moreover, because of the different behaviour for small and large x-values, it is appropriate for skewed (long-tailed) distributions. An implementation of this method in R does not seem available but it is implemented simple enough as follows.

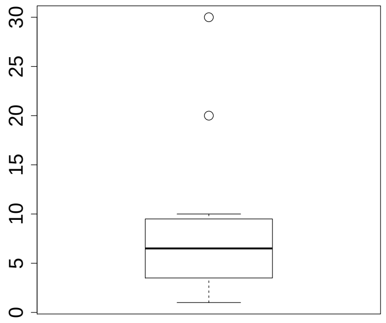


Figure 2: A box-and-whisker plot, produced with the `boxplot` function.

```
hboutlier <- function(x,r){
  x <- x[is.finite(x)]
  stopifnot(
    length(x) > 0
    , all(x>0)
  )
  xref <- median(x)
  if (xref <= sqrt(.Machine$double.eps))
    warning("Reference value close to zero: results may be inaccurate")
  pmax(x/xref, xref/x) > r
}
```

The above function returns a `logical` vector indicating which elements of x are outliers.

3.1.4 Obvious inconsistencies

An obvious inconsistency occurs when a record contains a value or combination of values that cannot correspond to a real-world situation. For example, a person's age cannot be negative, a man cannot be pregnant and an under-aged person cannot possess a drivers license.

Such knowledge can be expressed as *rules* or constraints. In data editing literature these rules are referred to as *edit rules* or *edits*, in short. Checking for obvious inconsistencies can be done straightforwardly in R using logical indices and recycling. For example, to check which elements of x obey the rule `x must be non negative' one simply uses the following.

```
x_nonnegative <- x >= 0
```

However, as the number of variables increases, the number of rules may increase rapidly and it may be beneficial to manage the rules separate from the data. Moreover, since multivariate rules may be interconnected by common variables, deciding which variable or variables in a record cause an inconsistency may not be straightforward.

The `editrules` package[6] allows one to define rules on categorical, numerical or mixed-type data sets which each record must obey. Furthermore, `editrules` can check which rules are obeyed or not and allows one to find the minimal set of variables to adapt so that all rules can be obeyed. The package also implements a number of basic rule operations allowing users to test rule sets for contradictions and certain redundancies.

As an example, we will work with a small file containing the following data.

```
age,agegroup,height,status,yearsmarried
21,adult,6.0,single,-1
2,child,3,married, 0
18,adult,5.7,married, 20
221,elderly, 5,widowed, 2
34,child, -7,married, 3
```

We read this data into a variable called `people` and define some restrictions on *age* using `editset`.

```
people <- read.csv("files/people.txt")
library(editrules)
(E <- editset(c("age >=0", "age <= 150")))
##
## Edit set:
## num1 : 0 <= age
## num2 : age <= 150
```

The `editset` function parses the textual rules and stores them in an `editset` object. Each rule is assigned a name according to it's type (numeric, categorical, or mixed) and a number. The data can be checked against these rules with the `violatedEdits` function. Record 4 contains an error according to one of the rules: an age of 21 is not allowed.

```
violatedEdits(E, people)
##       edit
## record  num1  num2
##      1 FALSE FALSE
##      2 FALSE FALSE
##      3 FALSE FALSE
##      4 FALSE  TRUE
##      5 FALSE FALSE
```

`violatedEdits` returns a `logical` array indicating for each row of the data, which rules are violated.

The number and type of rules applying to a data set usually quickly grow with the number of variables. With `editrules`, users may read rules, specified in a limited R-syntax, directly from a text file using the `editfile` function. As an example consider the contents of the following text file.

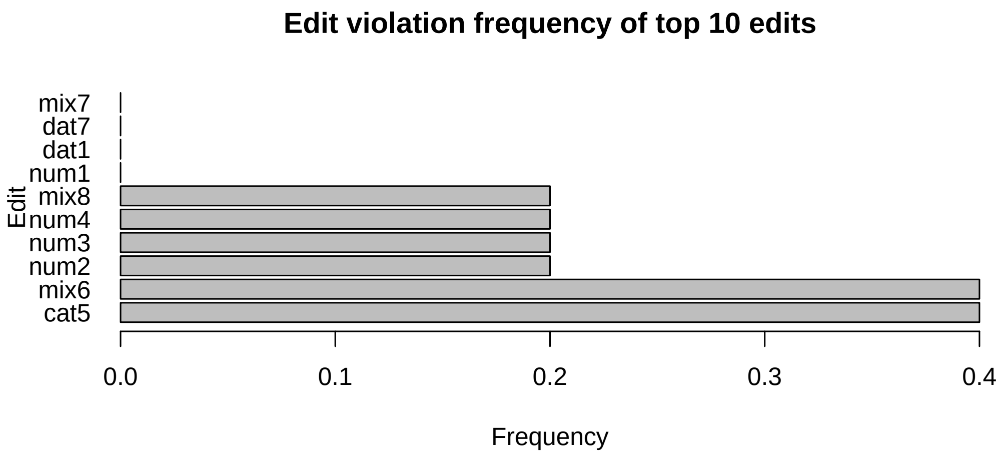
```
# numerical rules
age >= 0
height > 0
age <= 150
age > yearsmarried

# categorical rules
status %in% c("married","single","widowed")
agegroup %in% c("child","adult","elderly")
if ( status == "married" ) agegroup %in% c("adult","elderly")

# mixed rules
if ( status %in% c("married","widowed")) age - yearsmarried >= 17
if ( age < 18 ) agegroup == "child"
if ( age >= 18 && age <65 ) agegroup == "adult"
if ( age >= 65 ) agegroup == "elderly"
```

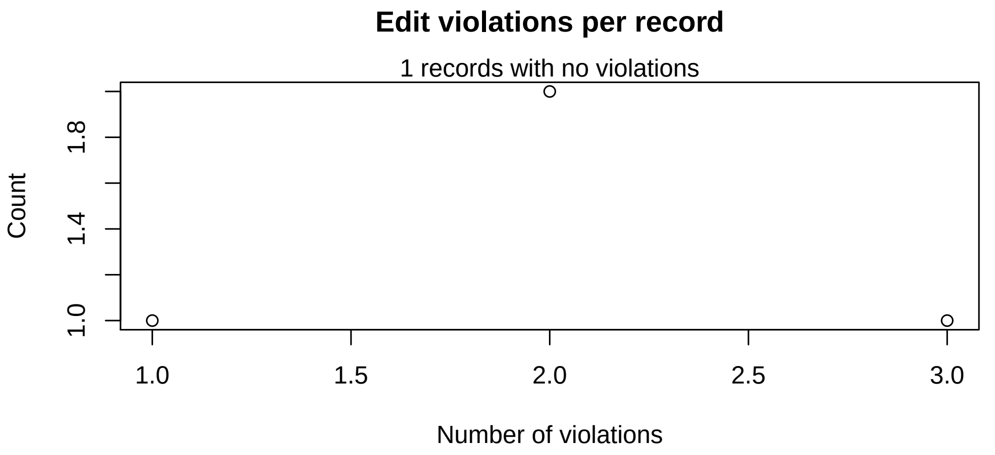
There are rules pertaining to purely numerical, purely categorical and rules pertaining to both data types. Moreover, there are univariate as well as multivariate rules. Comments are written behind the usual # character. The rule set can be read as follows.

```
E <- editfile("files/edits.txt")
```

As the number of rules grows, looking at the full array produced by `violatedEdits` becomes cumbersome. For this reason, `editrules` offers methods to summarize or visualize the result.

```
ve <- violatedEdits(E, people)
summary(ve)
## Edit violations, 5 observations, 0 completely missing (0%):
##
##  editname freq rel
##      cat5    2 40%
##      mix6    2 40%
##      num2    1 20%
##      num3    1 20%
##      num4    1 20%
##      mix8    1 20%
##
## Edit violations per record:
##
##  errors freq rel
##       0    1 20%
##       1    1 20%
##       2    2 40%
##       3    1 20%
plot(ve)
```





Here, the edit labeled `cat5` is violated by two records (20% of all records). Violated edits are sorted from most to least often violated. The plot visualizes the same information.

Since rules may pertain to multiple variables, and variables may occur in several rules (*e.g.* the *age* variable in the current example), there is a dependency between rules and variables. It can be informative to show these dependencies in a graph using the `plot` function, as in Figure 3.

3.1.5 Error localization

The interconnectivity of edits is what makes error localization difficult. For example, the graph in Figure 3 shows that a record violating edit `num4` may contain an error in *age* and/or *years married*. Suppose that we alter *age* so that `num4` is not violated anymore. We then run the risk of violating up to *six* other edits containing *age*.

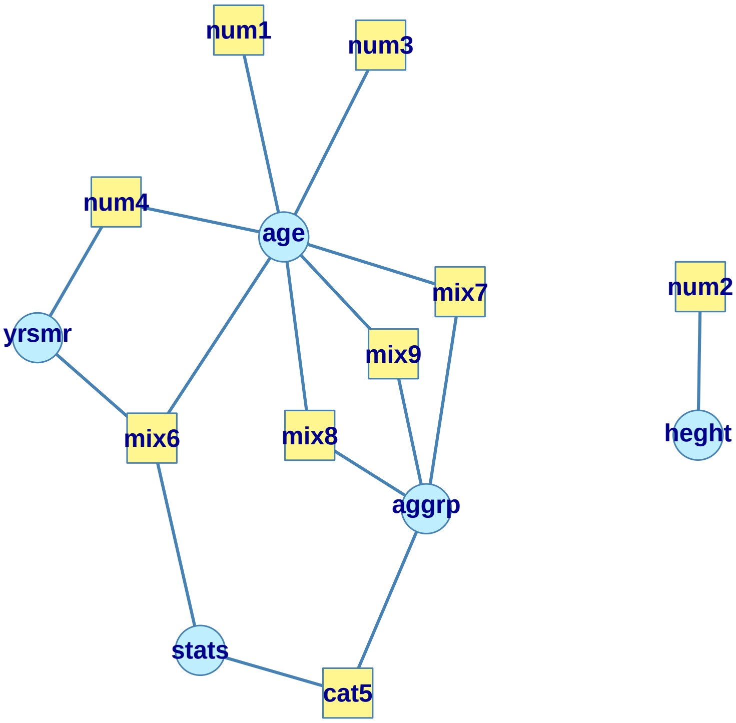


Figure 3: A graph plot, created with `plot(E)`, showing the interconnection of restrictions. Blue circles represent variables and yellow boxes represent restrictions. The lines indicate which restrictions pertain to what variables.

If we have no other information available but the edit violations, it makes sense to minimize the number of fields being altered. This principle, commonly referred to as the principle of Fellegi and Holt[9], is based on the idea that errors occur relatively few times and when they do, they occur randomly across variables. Over the years several algorithms have been developed to solve this minimization problem[7] of which two have been implemented in `editrules`. The `localizeErrors` function provides access to this functionality.

As an example we take two records from the `people` dataset from the previous subsection.

```
id <- c(2, 5)
people[id, ]
##   age agegroup height  status yearsmarried
## 2   2    child      3 married            0
## 5  34    child     -7 married            3
violatedEdits(E, people[id, ])
##       edit
## record  num1  num2  num3  num4  dat1  dat7 cat5  mix6  mix7  mix8  mix9
##      2 FALSE FALSE FALSE FALSE FALSE FALSE TRUE  TRUE FALSE FALSE FALSE
##      5 FALSE  TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE  TRUE FALSE
```

Record 2 violates `mix6` while record 5 violates edits `num2`, `cat5` and `mix8`. We use `localizeErrors`, with a mixed-integer programming approach to find the minimal set of variables to adapt.

```
le <- localizeErrors(E, people[id, ], method = "mip")
le$adapt
##     age agegroup height status yearsmarried
## 1 FALSE    FALSE  FALSE   TRUE        FALSE
## 2 FALSE     TRUE   TRUE  FALSE        FALSE
```

Here, the `le` object contains some processing metadata and a logical array labeled `adapt` which indicates the minimal set of variables to be altered in each record. It can be used in correction and imputation procedures for filling in valid values. Such procedures are not part of `editrules`, but for demonstration purposes we will manually fill in new values showing that the solution computed by `localizeErrors` indeed allows one to repair records to full compliance with all edit rules.

```
people[2, "status"] <- "single"
people[5, "height"] <- 7
people[5, "agegroup"] <- "adult"
summary(violatedEdits(E, people[id, ]))

## No violations detected, 0 checks evaluated to NA
```

The behaviour of `localizeErrors` can be tuned with various options. It is possible to supply a confidence weight for each variable allowing for fine grained control on which values should be adapted. It is also possible to choose a branch-and-bound based solver (instead of the MIP solver used here), which is typically slower but allows for more control.

3.2 Correction

Correction methods aim to fix inconsistent observations by altering invalid values in a record based on information from valid values. Depending on the method this is either a single-step procedure or a two-step procedure where first, an error localization method is used to empty certain fields, followed by an imputation step.

In some cases, the cause of errors in data can be determined with enough certainty so that the solution is almost automatically known. In recent years, several such methods have been developed and implemented in the `deducorrect` package[33]. In this section we give a quick overview of the possibilities of `deducorrect`.

3.2.1 Simple transformation rules

In practice, data cleaning procedures involve a lot of *ad-hoc* transformations. This may lead to long scripts where one selects parts of the data, changes some variables, selects another part, changes some more variables, etc. When such scripts are neatly written and commented, they can almost be treated as a log of the actions performed by the analyst. However, as scripts get longer it is better to store the transformation rules separately and log which rule is executed on what record. The `deducorrect` package offers functionality for this. Consider as an example the following (fictitious) dataset listing the body length of some brothers.

```
(marx <- read.csv("files/marx.csv", stringsAsFactors = FALSE))
##     name height unit
## 1 Gaucho 170.00   cm
## 2  Zeppo   1.74    m
## 3  Chico  70.00 inch
## 4  Gummo 168.00   cm
## 5  Harpo   5.91   ft
```

The task here is to standardize the lengths and express all of them in meters. The obvious way would be to use the indexing techniques summarized in Section 1.2.1, which would look something like this.

```
marx_m <- marx
I <- marx$unit == "cm"
marx_m[I, "height"] <- marx$height[I]/100
I <- marx$unit == "inch"
marx_m[I, "inch"] <- marx$height[I]/39.37
I <- marx$unit == "ft"
marx_m[I, "ft"] <- marx$height[I]/3.28
marx_m$unit <- "m"
```

Such operations quickly become cumbersome. Of course, in this case one could write a for-loop but that would hardly save any code. Moreover, if you want to check afterwards which values have been converted and for what reason, there will be a significant administrative overhead.

The deducorrect package takes all this overhead of your hands with the `correctionRules` functionality. For example, to perform the above task, one first specifies a file with correction rules as follows.

```
# convert centimeters
if ( unit == "cm" ){
  height <- height/100
}
# convert inches
if (unit == "inch" ){
  height <- height/39.37
}
# convert feet
if (unit == "ft" ){
  height <- height/3.28
}
# set all units to meter
unit <- "m"
```

With deducorrect we can read these rules, apply them to the data and obtain a log of all actual changes as follows.

```
# read the conversion rules.
R <- correctionRules("files/conversions.txt")
R
## Object of class 'correctionRules'
## ##  1-------
##   if (unit == "cm") {
##       height <- height/100
##   }
## ##  2-------
##   if (unit == "inch") {
##       height <- height/39.37
##   }
## ##  3-------
##   if (unit == "ft") {
##       height <- height/3.28
##   }
## ##  4-------
##   unit <- "m"
```

correctionRules has parsed the rules and stored them in a correctionRules object. We may now apply them to the data.

```
cor <- correctWithRules(R, marx)
```

The returned value, cor, is a list containing the corrected data

```
cor$corrected
##     name height unit
## 1 Gaucho  1.700    m
## 2  Zeppo  1.740    m
## 3  Chico  1.778    m
## 4  Gummo  1.680    m
## 5  Harpo  1.802    m
```

as well as a log of applied corrections.

```
cor$corrections[1:4]
##   row variable  old   new
## 1   1   height  170   1.7
## 2   1     unit   cm     m
## 3   3   height   70 1.778
## 4   3     unit inch     m
## 5   4   height  168  1.68
## 6   4     unit   cm     m
## 7   5   height 5.91 1.802
## 8   5     unit   ft     m
```

The log lists for each row, what variable was changed, what the old value was and what the new value is. Furthermore, the fifth column of cor$corrections shows the corrections that were applied (not shown above for formatting reasons)

```
cor$corrections[5]
##                                              how
## 1     if (unit == "cm") { height <- height/100 }
## 2                                    unit <- "m"
## 3 if (unit == "inch") { height <- height/39.37 }
## 4                                    unit <- "m"
## 5     if (unit == "cm") { height <- height/100 }
## 6                                    unit <- "m"
## 7    if (unit == "ft") { height <- height/3.28 }
## 8                                    unit <- "m"
```

So here, with just two commands, the data is processed and all actions logged in a `data.frame` which may be stored or analyzed. The rules that may be applied with `deducorrect` are rules that can be executed record-by-record.

By design, there are some limitations to which rules can be applied with `correctWithRules`. The processing rules should be executable record-by-record. That is, it is not permitted to use functions like `mean` or `sd`. The symbols that may be used can be listed as follows.

```
getOption("allowedSymbols")
##  [1] "if"         "else"       "is.na"      "is.finite" "=="
##  [6] "<"          "<="         "="          ">="         ">"
## [11] "!="         "!"          "%in%"       "identical" "sign"
## [16] "abs"        "||"         "|"          "&&"         "&"
## [21] "("          "{"          "<-"         "="          "+"
## [26] "-"          "*"          "^"          "/"          "%%"
## [31] "%/%"
```

When the rules are read by `correctionRules`, it checks whether any symbol occurs that is not in the list of allowed symbols and returns an error message when such a symbol is found as in the following example.

```
correctionRules(expression(x <- mean(x)))
##
## Forbidden symbols found:
## ## ERR  1 ------
## Forbidden symbols: mean
##   x <- mean(x)

## Error:  Forbidden symbols found
```

Finally, it is currently not possible to add new variables using `correctionRules` although such a feature will likely be added in the future.

3.2.2 Deductive correction

When the data you are analyzing is generated by people rather than machines or measurement devices, certain typical human-generated errors are likely to occur. Given that data has to obey certain edit rules, the occurrence of such errors can sometimes be detected from raw data with (almost) certainty. Examples of errors that can be detected are typing errors in numbers (under linear restrictions) rounding errors in numbers and sign errors or variable swaps[22]. The `deducorrect` package has a number of functions available that can correct such errors. Below we give some examples, every time with just a single edit rule. The functions can handle larger sets of edits however.

With `correctRoundings` deviations of the size of one or two measurement units can be repaired. The function randomly selects one variable to alter such that the rule violation(s) are nullified while no new violations are generated.

```
e <- editmatrix("x + y == z")
d <- data.frame(x = 100, y = 101, z = 200)
cor <- correctRounding(e, d)
cor$corrected
##     x   y   z
## 1 100 101 201
cor$corrections
##   row variable old new
## 1   1        z 200 201
```

The function `correctSigns` is able to detect and repair sign errors. It does this by trying combinations of variable swaps on variables that occur in violated edits.

```
e <- editmatrix("x + y == z")
d <- data.frame(x = 100, y = -100, z = 200)
cor <- correctSigns(e, d)
cor$corrected
##     x   y   z
## 1 100 100 200
cor$corrections
##   row variable  old new
## 1   1        y -100 100
```

Finally, the function `correctTypos` is capable of detecting and correcting typographic errors in numbers. It does this by computing candidate solutions and checking whether those candidates are less than a certain string distance (see Section 2.4.2) removed from the original.

```
e <- editmatrix("x + y == z")
d <- data.frame(x = 123, y = 132, z = 246)
cor <- correctTypos(e, d)
cor$corrected
##     x   y   z
## 1 123 123 246
cor$corrections
##   row variable old new
## 1   1        y 132 123
```

Indeed, swapping the 3 and the 2 in $y = 132$ solves the edit violation.

Tip. *Every `correct-` function in `deducorrect` is an object of class `deducorrect`. When printed, it doesn't show the whole contents (the corrected data and logging information) but a summary of what happened with the data. A `deducorrect` object also contains data on timing, user, and so on. See `?"deducorrect-object"` for a full explanation.*

3.2.3 Deterministic imputation

In some cases a missing value can be determined because the observed values combined with their constraints force a unique solution. As an example, consider a record with variables listing the costs for *staff cleaning*, *housing* and the total *total*. We have the following rules.

$$\begin{array}{l} \mathit{staff} + \mathit{cleaning} + \mathit{housing} = \mathit{total} \\ \mathit{staff} \geq 0 \\ \mathit{housing} \geq 0 \\ \mathit{cleaning} \geq 0 \end{array} \tag{2}$$

In general, if one of the variables is missing the value can clearly be derived by solving it for the first rule (providing that the solution doesn't violate the last rule). However, there are other cases where unique solutions exist. Suppose that we have $staff = total$. Assuming that these values are correct, the only possible values for the other two variables is $housing = cleaning = 0$. The deducorrect function deduImpute is able to recognize such cases and compute the unique imputations.

```
E <- editmatrix(expression(
  staff + cleaning + housing == total,
  staff    >= 0,
  housing  >= 0,
  cleaning >= 0
))
dat <- data.frame(
  staff = c(100,100,100),
  housing = c(NA,50,NA),
  cleaning = c(NA,NA,NA),
  total = c(100,180,NA)
)
dat
##   staff housing cleaning total
## 1   100      NA       NA   100
## 2   100      50       NA   180
## 3   100      NA       NA    NA
cor <- deduImpute(E,dat)
cor$corrected
##   staff housing cleaning total
## 1   100       0        0   100
## 2   100      50       30   180
## 3   100      NA       NA    NA
```

Note that deduImpute only imputes those values that can be derived with absolute certainty (uniquely) from the rules. In the example, there are many possible solutions to impute the last record and hence deduImpute leaves it untouched.

Similar situations exist for categorical data, which are handled by deduImpute as well.

```
E <- editarray(expression(
  age %in% c("adult","under-aged"),
  driverslicense %in% c(TRUE, FALSE),
  if ( age == "under-aged" ) !driverslicense
))
dat <- data.frame(
  age = NA,
  driverslicense = TRUE
)
dat
##   age driverslicense
## 1  NA           TRUE
cor <- deduImpute(E,dat)
cor$corrected
##     age driverslicense
## 1 adult           TRUE
```

Here, deduImpute uses automated logic to derive from the conditional rules that if someone has a drivers license, he has to be an adult.

3.3 Imputation

Imputation is the process of estimating or deriving values for fields where data is missing. There is a vast body of literature on imputation methods and it goes beyond the scope of this tutorial to discuss all of them. In stead, we present in Table 3 an overview of packages that offer some kind of imputation method and list them against a number of popular model-based imputation methods. We note that the list of packages and methods are somewhat biased towards applications at Statistics Netherlands, as the table was originally produced during a study for internal purposes[31]. Nevertheless we feel that this overview can be a quite useful place to start. De Waal, Pannekoek and Scholtus[7] (Chpt. 7) give a consice overview of several well-established imputation methods.

The packages `Amelia`, `deducorrect` and `mix` do not implement any of the methods mentioned in the table. That is because `Amelia` implements a multiple imputation method which was out of scope for the study mentioned above. The `deducorrect` package implements deductive and deterministic methods, and we choose not to call this model-based in this tutorial. The `mix` package implements a Bayesian estimation method based on an Markov Chain Monte Carlo algorithm.

There is no one single best imputation method that works in all cases. The imputation model of choice depends on what auxiliary information is available and whether there are (multivariate) edit restrictions on the data to be imputed. The availability of R software for imputation under edit restrictions is, to our best knowledge, limited. However, a viable strategy for imputing numerical data is to first impute missing values without restrictions, and then minimally adjust the imputed values so that the restrictions are obeyed. Separately, these methods are available in R.

In the following subsections we discuss three types of imputation models and give some pointers to how to implement them using R. The next section (3.3.4) is devoted to value adjustment.

3.3.1 Basic numeric imputation models

Here, we distinguish three imputation models. The first is imputation of the mean:

$$\hat{x}_i = \bar{x}, \tag{3}$$

where the $\hat{x}_i$ is the imputation value and the mean is taken over the observed values. The usability of this model is limited since it obviously causes a bias in measures of spread, estimated from the sample after imputation. However, in base R it is implemented simply enough.

```
x[is.na(x)] <- mean(x, na.rm = TRUE)
```

In principle one can use other measures of centrality, and in fact, the `Hmisc` package has a convenient wrapper function allowing you to specify what function is used to compute imputed values from the non-missing. For example, imputation of the mean or median can be done as follows.

```
library(Hmisc)
x <- impute(x, fun = mean)  # mean imputation
x <- impute(x, fun = median)  # median imputation
```

An nice feature of the `impute` function is that the resulting vector ``remembers'' what values were imputed. This information may be requested with `is.imputed` as in the example below.

Table 3: An overview of imputation functionality offered by some R packages. reg: regression, rand: random, seq: sequential, NN: nearest neighbor, pmm: predictive mean matching, kNN: k-nearest-neighbors, int: interpolation, lo/no last observation carried forward / next observation carried backward, LS: method of Little and Su.

	Numeric			Hot deck				Longitudinal		
Package	mean	ratio	reg.	rand	seq	pmm	kNN	int	lo/no	LS
Amelia[17]	·	·	·	·	·	·	·	·	·	·
BaBoon[20]	·	·	·	·	·	✓	·	·	·	·
cat[27]	·	·	·	✓	·	·	·	·	·	·
deducorrect[33]	·	·	·	·	·	·	·	·	·	·
e1071[21]	✓	·	·	·	·	·	·	·	·	·
ForImp	✓	·	·	·	·	·	✓‡	·	·	·
Hmisc[18]	✓	·	·	✓	·	✓	·	·	·	·
imputation[37]	✓	·	✓†	·	·	·	✓	·	·	·
impute[14]	·	·	·	·	·	·	✓	·	·	·
mi[23]	·	·	✓*	✓	·	✓	·	·	·	·
mice[30]	✓	·	✓*	✓	·	✓	·	·	·	·
mix[3]	·	·	·	·	·	·	·	·	·	·
norm[26]	·	·	·	·	·	·	·	·	·	·
robCompositions[25]	·	·	✓*	✓	·	·	✓	·	·	·
rrcovNA[28]	·	·	·	·	·	·	·	·	·	·
StatMatch[8]	·	·	·	✓	·	·	✓	·	·	·
VIM[24]	·	·	✓*	✓	·	·	✓	·	·	·
yaImpute[5]	·	·	·	·	·	·	✓	·	·	·
zoo[39]	✓	·	·	·	·	·	·	✓	✓	·

*Methods are ultimately based on some form of regression, but are more involved than simple linear regression.

†Uses a non-informative auxiliary variable (row number).

‡Uses nearest neighbor as part of a more involved imputation scheme.

```
x <- 1:5  # create a vector...
x[2] <- NA  # ...with an empty value
x <- impute(x, mean)
x
##     1     2     3     4     5
##  1.00 3.25*  3.00  4.00  5.00
is.imputed(x)
## [1] FALSE  TRUE FALSE FALSE FALSE
```

Note also that imputed values are printed with a post-fixed asterix.

The second imputation model we discuss is ratio imputation. Here, the imputation estimate $\hat{x}_i$ is given by

$$\hat{x}_i = \hat{R} y_i, \tag{4}$$

Where y_i is a covariate and $\hat{R}$ is an estimate of the average ratio between x and y. Often this will be given by the sum of observed x values divided by the sum of corresponding y values although variants are possible. Ratio imputation has the property that the estimated value equals $\hat{x} = 0$ when $y = 0$, which is in general not guaranteed in linear regression. Ratio imputation may be a good model to use when restrictions like $x \geq 0$ and/or $y \geq 0$ apply. There is no package directly implementing ratio imputation, unless it is regarded a special case of regression imputation. It is easily implemented using plain R though. Below, we suppose that x and y are numeric vectors of equal length, x contains missing values and y is complete.

```
I <- is.na(x)
R <- sum(x[!I])/sum(y[!I])
x[I] <- R * y[I]
```

Unfortunately, it is not possible to simply wrap the above in a function and pass this to `HMisc`'s `impute` function. There seem to be no packages that implement ratio imputation in a single function.

The third, and last numerical model we treat are (generalized) linear regression models. In such models, missing values are imputed as follows

$$\hat{x}_i = \hat{\beta}_0 + \hat{\beta}_1 y_{1,i} + \cdots + \hat{\beta}_k y_{k,i}, \tag{5}$$

where the $\hat{\beta}_0, \hat{\beta}_1 \dots \hat{\beta}_k$ are estimated linear regression coefficients for each of the auxiliary variables $y_1, y_2 \dots, y_k$. Estimating linear models is easy enough using `lm` and `predict` in standard R. Here, we use the built-in `iris` data set as an example.

```
data(iris)
iris$Sepal.Length[1:10] <- NA
model <- lm(Sepal.Length ~ Sepal.Width + Petal.Width, data = iris)
I <- is.na(iris$Sepal.Length)
iris$Sepal.Length[I] <- predict(model, newdata = iris[I, ])
```

Above, `lm` automatically omits rows with empty values and estimates the model based on the remaining records. Next, we used `predict` to estimate the missing values.

It should be noted that although `Hmisc`, `VIM`, `mi` and `mice` all implement imputation methods that ultimately use some form of regression, they do not include a simple interface for the case described above. The more advanced methods in those packages are aimed to be more precise and/or robust against outliers than standard (generalized) linear regression. Moreover, both `mice` and `mi` implement methods for multiple imputation, which allows for estimation of the imputation variance. Such methods are beyond the scope of the current text. The `imputation` package has a function `lmImpute` but it uses the row number as auxiliary variable which limits its practical usability to cases where the row number is directly related to a covariate value.

3.3.2 Hot deck imputation

In hot deck imputation, missing values are imputed by copying values from similar records in the same dataset. Or, in notation:

$$\hat{x}_i = x_j, \tag{6}$$

where x_j is taken from the observed values. Hot-deck imputation can be applied to numerical as well as categorical data but is only viable when enough donor records are available.

The main question in hot-deck imputation is how to choose the replacement value x_j from the observed values. Here, we shortly discuss four well-known flavors.

In *random* hot-deck imputation, a value is chosen randomly, and uniformly from the same data set. When meaningful, random hot deck methods can be applied per stratum. For example, one may apply random hot-deck imputation of body height for male and female respondents separately. Random hot-deck on a single vector can be applied with the `impute` function of the `Hmisc` package.

```
data(women)
# add some missings
height <- women$height
height[c(6, 9)] <- NA
height
##  [1] 58 59 60 61 62 NA 64 65 NA 67 68 69 70 71 72
(height <- impute(height, "random"))
##   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
##  58  59  60  61  62 64*  64  65 70*  67  68  69  70  71  72
```

Note that the outcome of this imputation is very likely tot be different each time it is executed. If you want the results to be random, but repeatable (e.g. for testing purposes) you can use `set.seed(<a number>)` prior to calling `impute`.

In *sequential* hot deck imputation, the vector containing missing values is sorted according to one or more auxiliary variables so that records that have similar auxiliaries occur sequentially in the `data.frame`. Next, each missing value is imputed with the value from the first following record that has an observed value. There seems to be no package that implements the sequential hot deck technique. However, given a vector that is sorted, the following function offers some basic functionality.

```
# x    :  vector to be imputed
# last : value to use if last value of x is empty
seqImpute <- function(x,last){
  n <- length(x)
  x <- c(x,last)
  i <- is.na(x)
  while(any(i)){
    x[i] <- x[which(i) + 1]
    i <- is.na(x)
  }
  x[1:n]
}
```

We will use this function in exercise 3.4.

Predictive mean matching (pmm) is a form of nearest-neighbor hot-deck imputation with a specific distance function. Suppose that in record j, the ith value is missing. One then estimates the value $\hat{x}_{ji}$ using a prediction model (often some form of regression) for the record with the missing values as well as for possible donor records $k \neq j$ where the $i$the value is not missing. One then determines $k^* = \arg\min_{k\neq j} d(\hat{x}_{ki}, \hat{x}_{ji})$ with d a distance function and copies the observed value x_{k^*i} to the jth record.

Predictive mean matching imputation has been implemented in several packages (`mi`, `mice`, `BaBoon`) but each time either for a specific type of data and/or using a specific prediction model (*e.g.* `Bayesglm`). Moreover, the packages known to us always use pmm as a part of a more elaborate multiple imputation scheme, which we deemed out of scope for this tutorial.

3.3.3 kNN-imputation

In k *nearest neighbor* imputation one defines a distance function $d(i, j)$ that computes a measure of dissimilarity between records. A missing value is then imputed by finding first the k records nearest to the record with one or more missing values. Next, a value is chosen from or computed out of the k nearest neighbors. In the case where a value is picked from the k nearest neighbors, kNN-imputation is a form of hot-deck imputation.

The VIM package contains a function called kNN that uses Gowers distance[12] to determine the k nearest neighbors. Gower's distance between two records labeled i and j is defined as

$$d_g(i,j)\frac{\sum_k w_{ijk} d_k(i,j)}{\sum_k w_{ijk}}, \tag{7}$$

where the sum runs over all variables in the record and $d_k(i,j)$ is the distance between the value of variable k in record i and record j. For categorical variables, $d_k(i,j) = 0$ when the value for the $k'th$ variable is the same in record i and j and 1 otherwise. For numerical variables the distance is given by $1 - (x_i - x_j)/(\max(x) - \min(x))$. The weight $w_{ijk} = 0$ when the k'th variable is missing in record i or record j and otherwise 1.

Here is an example of kNN.

```
library(VIM)
data(iris)
n <- nrow(iris)
# provide some empty values (10 in each column, randomly)
for (i in 1:ncol(iris)) {
    iris[sample(1:n, 10, replace = FALSE), i] <- NA
}
iris2 <- kNN(iris)
## Time difference of -0.05939 secs
```

The kNN function determines the k (default: 5) nearest neighbors of a record with missing values. For numerical variables the median of the k nearest neighbors is used as imputation value, for categorical variables the category that most often occurs in the k nearest neighbors is used. Both these functions may be replaced by the user. Finally, kNN prints (negatively) the time it took to complete the imputation. We encourage the reader to execute the above code and experiment further.

3.3.4 Minimal value adjustment

Once missing numerical values have been adequately imputed, there is a good chance that the resulting records violate a number of edit rules. The obvious reason is that there are not many methods that can perform imputation under arbitrary edit restrictions. One viable option is therefore to minimally adjust imputed values such that after adjustment the record passes every edit check within a certain tolerance.

We need to specify more clearly what *minimal* adjustment means here. The rspa package[34] is able to take a numerical record $\boldsymbol{x}^0$ and replace it with a record $\boldsymbol{x}$, such that the weighted Euclidean distance

$$\sum_i w_i(x_i - x_i^0)^2, \tag{8}$$

is minimized and $\boldsymbol{x}$ obeys a given set of (in)equality restrictions

$$\boldsymbol{Ax} \leq \boldsymbol{b}. \tag{9}$$

As a practical example consider the following script.

```
library(editrules)
library(rspa)
E <- editmatrix(expression(x + y == z, x >= 0, y >= 0))
d <- data.frame(x = 10, y = 10, z = 21)
```

```
d1 <- adjustRecords(E, d)
d1$adjusted
##       x     y     z
## 1 10.33 10.33 20.67
```

The function `adjustRecords` adjusts every record minimally to obey the restrictions in `E`. Indeed, we may check that the adjusted data now obeys every rule within `adjustRecords`' default tolerance of 0.01.

```
violatedEdits(E, d1$adjusted, tol = 0.01)
##       edit
## record  num1  num2  num3
##      1 FALSE FALSE FALSE
```

By default, all variables in the input `data.frame` are adjusted with equal amounts. However, suppose that z is an imputed value while x and y are observed, original values. We can tell `adjustRecords` to only adjust `z` as follows.

```
A <- array(c(x = FALSE, y = FALSE, z = TRUE), dim = c(1, 3))
A
##       [,1]  [,2] [,3]
## [1,] FALSE FALSE TRUE
d2 <- adjustRecords(E, d, adjust = A)
d2$adjusted
##    x  y  z
## 1 10 10 20
```

Exercises

In the following exercises we are going to use the `dirty_iris` dataset. You can download this dataset from
`https://raw.github.com/edwindj/datacleaning/master/data/dirty_iris.csv`

Exercise 3.1. *Reading and manually checking.*

a. *View the file in a text-editor to determine its format and read the file into R. Make sure that strings are not converted to factor.*
b. *Calculate the number and percentage of observations that are complete.*
c. *Does the data contain other special values? If it does, replace them with NA*

Exercise 3.2. *Checking with rules*

a. *Besides missing values, the data set contains errors. We have the following background knowledge:*
   - *`Species` should be one of the following values: `setosa`, `versicolor` or `virginica`.*
   - *All measured numerical properties of an iris should be positive.*
   - *The petal length of an iris is at least 2 times its petal width.*
   - *The sepal length of an iris cannot exceed 30 cm.*
   - *The sepals of an iris are longer than its petals.*

   Define these rules in a separate text file and read them into R using `editfile` (package `editrules`). Print the resulting constraint object.
b. *Determine how often each rule is broken (`violatedEdits`). Also summarize and plot the result.*
c. *What percentage of the data has no errors?*
d. *Find out which observations have too long petals using the result of `violatedEdits`.*
e. *Find outliers in sepal length using `boxplot` and `boxplot.stats`. Retrieve the corrosponding observations and look at the other values. Any ideas what might have happened? Set the outliers to NA (or a value that you find more appropiate)*

Exercise 3.3. *Correcting*

a. *Replace non positive values from `Petal.Width` with NA using `correctWithRules` from the library `deducorrect`.*
b. *Replace all erronous values with NA using (the result of) `localizeErrors`*

Exercise 3.4. *Imputing*

a. *Use kNN imputation (VIM) to impute all missing values.*
b. *Use sequential hotdeck imputation to impute `Petal.Width` by sorting the dataset on `Species`. Compare the imputed `Petal.Width` with the sequential hotdeck imputation method. Note the ordering of the data!*
c. *Do the same but now by sorting the dataset on `Species` and `Sepal.Length`.*

References

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1--58, 2008.

[2] V. Barnett and T. Lewis. *Outliers in statistical data*. Wiley, New York, NY, 3rd edition, 1994.

[3] Original by J.L. Schafer. *mix: Estimation/multiple Imputation for Mixed Categorical and Continuous Data*, 2010. R package version 1.0-8.

[4] J.M. Chambers. *Software for data analyses; programming with R*. Springer, 2008.

[5] N. L. Crookston and A. O. Finley. yaimpute: An r package for knn imputation. *Journal of Statistical Software*, 23(10), 10 2007.

[6] Edwin de Jonge and Mark van der Loo. *editrules: R package for parsing, applying, and manipulating data cleaning rules*, 2012. R package version 2.8.

[7] T. De Waal, J. Pannekoek, and S. Scholtus. *Handbook of statistical data editing and imputation*. Wiley handbooks in survey methodology. John Wiley & Sons, 2011.

[8] M. D'Orazio. *StatMatch: Statistical Matching*, 2012. R package version 1.2.0.

[9] I. P. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *Journal of the Americal Statistical Association*, 71:17--35, 1976.

[10] M. Fitzgerald. *Introducing regular expressions*. O'Reilley Media, 2012.

[11] J. Friedl. *Mastering regular expressions*. O'Reilley Media, 2006.

[12] J.C. Gower. A general coefficient of similarity and some of its properties. *Biometrics*, 27:857--874, 1971.

[13] G. Grolemund and H. Wickham. Dates and times made easy with lubridate. *Journal of Statistical Software*, 40(3):1--25, 2011.

[14] T. Hastie, R. Tibshirani, B. Narasimhan, and G. Chu. *impute: impute: Imputation for microarray data*. R package version 1.34.0.

[15] D.M. Hawkins. *Identification of outliers*. Monographs on applied probability and statistics. Chapman and Hall, 1980.

[16] M.A. Hiridoglou and J.-M. Berthelot. Statistical editing and imputation for periodic business surveys. *Survey methodology*, 12(1):73--83, 1986.

[17] J. Honaker, G. King, and M. Blackwell. Amelia II: A program for missing data. *Journal of Statistical Software*, 45(7):1--47, 2011.

[18] F.E. Harrell Jr, with contributions from C. Dupont, and many others. *Hmisc: Harrell Miscellaneous*, 2013. R package version 3.10-1.1.

[19] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707--710, 1966.

[20] F. Meinfelder. *BaBooN: Bayesian Bootstrap Predictive Mean Matching - Multiple and single imputation for discrete data*, 2011. R package version 0.1-6.

[21] D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, and F. Leisch. *e1071: Misc Functions of the Department of Statistics (e1071), TU Wien*, 2012. R package version 1.6-1.

[22] S. Scholtus. Algorithms for correcting sign errors and rounding errors in business survey data. *Journal of Official Statistics*, 27(3):467--490, 2011.

[23] Y.-S. Su, A. Gelman, J. Hill, and M. Yajima. Multiple imputation with diagnostics (mi) in R: Opening windows into the black box. *Journal of Statistical Software*, 45(2):1--31, 2011.

[24] M. Templ, A. Alfons, A. Kowarik, and B. Prantner. *VIM: Visualization and Imputation of Missing Values*, 2012. R package version 1.2.0.

[25] M. Templ, K. Hron, and P. Filzmoser. *robCompositions: an R-package for robust statistical analysis of compositional data*. John Wiley and Sons, 2011.

[26] Ported to R by Alvaro A. Novo. Original by Joseph L. Schafer. *norm: Analysis of multivariate normal datasets with missing values*, 2013. R package version 1.0-9.5.

[27] Ported to R by T. Harding and F. Tusell. Original by J. L. Schafer. *cat: Analysis of categorical-variable datasets with missing values*, 2012. R package version 0.0-6.5.

[28] V. Todorov. *Robust Location and Scatter Estimation and Robust Multivariate Analysis with High Breakdown Point for Incomplete Data*, 2012. R package version 0.4-03.

[29] J.W. Tukey. *Exploratory data analysis*. Adison-Wesley, 1977.

[30] S. van Buuren and K. Groothuis-Oudshoorn. mice: Multivariate imputation by chained equations in R. *Journal of Statistical Software*, 45(3):1--67, 2011.

[31] B. van den Broek. Imputation in R, 2012. Statistics Netherlands, internal report.

[32] M. van der Loo. *stringdist: String distance functions for R*, 2013. R package version 0.5.0.

[33] M. van der Loo, E. de Jonge, and S. Scholtus. *deducorrect: Deductive correction, deductive imputation, and deterministic correction.*, 2011-2013. R package version 1.3-2.

[34] Mark van der Loo. *rspa: Adapt numerical records to fit (in)equality restrictions with the Successive Projection Algorithm*, 2012. R package version 0.1-2.

[35] w3techs. Usage of character encodings for websites, 2013.

[36] H. Wickham. stringr: modern, consistent string processing. *R Journal*, 2(2):38--40, 2010.

[37] J. Wong. *imputation: imputation*, 2013. R package version 2.0.1.

[38] Y. Xi. *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, 2013.

[39] A. Zeileis and G. Grothendieck. zoo: S3 infrastructure for regular and irregular time series. *Journal of Statistical Software*, 14(6):1--27, 2005.

Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

Tidy Data

Hadley Wickham
RStudio

Abstract

A huge amount of effort is spent cleaning data to get it ready for analysis, but there has been little research on how to make data cleaning as easy and effective as possible. This paper tackles a small, but important, component of data cleaning: data tidying. Tidy datasets are easy to manipulate, model and visualise, and have a specific structure: each variable is a column, each observation is a row, and each type of observational unit is a table. This framework makes it easy to tidy messy datasets because only a small set of tools are needed to deal with a wide range of un-tidy datasets. This structure also makes it easier to develop tidy tools for data analysis, tools that both input and output tidy datasets. The advantages of a consistent data structure and matching tools are demonstrated with a case study free from mundane data manipulation chores.

Keywords: data cleaning, data tidying, relational databases, R.

1. Introduction

It is often said that 80% of data analysis is spent on the process of cleaning and preparing the data (Dasu and Johnson 2003). Data preparation is not just a first step, but must be repeated many over the course of analysis as new problems come to light or new data is collected. Despite the amount of time it takes, there has been surprisingly little research on how to clean data well. Part of the challenge is the breadth of activities it encompasses: from outlier checking, to date parsing, to missing value imputation. To get a handle on the problem, this paper focusses on a small, but important, aspect of data cleaning that I call data **tidying**: structuring datasets to facilitate analysis.

The principles of tidy data provide a standard way to organise data values within a dataset. A standard makes initial data cleaning easier because you don't need to start from scratch and reinvent the wheel every time. The tidy data standard has been designed to facilitate initial exploration and analysis of the data, and to simplify the development of data analysis tools that work well together. Current tools often require translation. You have to spend time munging the output from one tool so you can input it into another. Tidy datasets and tidy tools work hand in hand to make data analysis easier, allowing you to focus on the interesting domain problem, not on the uninteresting logistics of data.

The principles of tidy data are closely tied to those of relational databases and Codd's relational algebra (Codd 1990), but are framed in a language familiar to statisticians. Computer scientists have also contributed much to the study of data cleaning. For example, Lakshmanan, Sadri, and Subramanian (1996) define an extension to SQL to allow it to operate on messy datasets, Raman and Hellerstein (2001) provide a framework for cleaning datasets, and Kandel, Paepcke, Hellerstein, and Heer (2011) develop an interactive tool with a friendly user interface which automatically creates code to clean data. These tools are useful but they are presented in a language foreign to most statisticians, they fail to give much advice on how datasets should be structured, and they lack connections to the tools of data analysis.

The development of tidy data has been driven by my experience working with real-world datasets. With few, if any, constraints on their organisation, such datasets are often constructed in bizarre ways. I have spent countless hours struggling to get such datasets organised in a way that makes data analysis possible, let alone easy. I have also struggled to impart these skills to my students so they could tackle real-world datasets on their own. In the course of these struggles I developed the **reshape** and **reshape2** (Wickham 2007) packages. While I could intuitively use the tools and teach them through examples, I lacked the framework to make my intuition explicit. This paper provides that framework. It provides a comprehensive "philosophy of data": one that underlies my work in the **plyr** (Wickham 2011) and **ggplot2** (Wickham 2009) packages.

The paper proceeds as follows. Section 2 begins by defining the three characteristics that make a dataset tidy. Since most real world datasets are not tidy, Section 3 describes the operations needed to make messy datasets tidy, and illustrates the techniques with a range of real examples. Section 4 defines tidy tools, tools that input and output tidy datasets, and discusses how tidy data and tidy tools together can make data analysis easier. These principles are illustrated with a small case study in Section 5. Section 6 concludes with a discussion of what this framework misses and what other approaches might be fruitful to pursue.

2. Defining tidy data

> Happy families are all alike; every unhappy family is unhappy in its own way
>
> Leo Tolstoy

Like families, tidy datasets are all alike but every messy dataset is messy in its own way. Tidy datasets provide a standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning). In this section, I'll provide some standard vocabulary for describing the structure and semantics of a dataset, and then use those definitions to define tidy data.

2.1. Data structure

Most statistical datasets are rectangular tables made up of **rows** and **columns**. The columns are almost always labelled and the rows are sometimes labelled. Table 1 provides some data about an imaginary experiment in a format commonly seen in the wild. The table has two columns and three rows, and both rows and columns are labelled.

	treatmenta	treatmentb
John Smith	—	2
Jane Doe	16	11
Mary Johnson	3	1

Table 1: Typical presentation dataset.

There are many ways to structure the same underlying data. Table 2 shows the same data as Table 1, but the rows and columns have been transposed. The data is the same, but the layout is different. Our vocabulary of rows and columns is simply not rich enough to describe why the two tables represent the same data. In addition to appearance, we need a way to describe the underlying semantics, or meaning, of the values displayed in table.

	John Smith	Jane Doe	Mary Johnson
treatmenta	—	16	3
treatmentb	2	11	1

Table 2: The same data as in Table 1 but structured differently.

2.2. Data semantics

A dataset is a collection of **values**, usually either numbers (if quantitative) or strings (if qualitative). Values are organised in two ways. Every value belongs to a **variable** and an **observation**. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes.

Table 3 reorganises Table 1 to make the values, variables and obserations more clear. The dataset contains 18 values representing three variables and six observations. The variables are:

1. `person`, with three possible values (John, Mary, and Jane).

2. `treatment`, with two possible values (a and b).

3. `result`, with five or six values depending on how you think of the missing value (-, 16, 3, 2, 11, 1).

The experimental design tells us more about the structure of the observations. In this experiment, every combination of of `person` and `treatment` was measured, a completely crossed design. The experimental design also determines whether or not missing values can be safely dropped. In this experiment, the missing value represents an observation that should have been made, but wasn't, so it's important to keep it. Structural missing values, which represent measurements that can't be made (e.g., the count of pregnant males) can be safely removed.

name	trt	result
John Smith	a	—
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

Table 3: The same data as in Table 1 but with variables in columns and observations in rows.

For a given dataset, it's usually easy to figure out what are observations and what are variables, but it is surprisingly difficult to precisely define variables and observations in general. For example, if the columns in the Table 1 were `height` and `weight` we would have been happy to call them variables. If the columns were `height` and `width`, it would be less clear cut, as we might think of height and width as values of a `dimension` variable. If the columns were `home phone` and `work phone`, we could treat these as two variables, but in a fraud detection environment we might want variables `phone number` and `number type` because the use of one phone number for multiple people might suggest fraud. A general rule of thumb is that it is easier to describe functional relationships between variables (e.g., `z` is a linear combination of `x` and `y`, `density` is the ratio of `weight` to `volume`) than between rows, and it is easier to make comparisons between groups of observations (e.g., average of group a vs. average of group b) than between groups of columns.

In a given analysis, there may be multiple levels of observation. For example, in a trial of new allergy medication we might have three observational types: demographic data collected from each person (`age`, `sex`, `race`), medical data collected from each person on each day (`number of sneezes`, `redness of eyes`), and meterological data collected on each day (`temperature`, `pollen count`).

2.3. Tidy data

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In **tidy data**:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

This is Codd's 3rd normal form (Codd 1990), but with the constraints framed in statistical language, and the focus put on a single dataset rather than the many connected datasets common in relational databases. **Messy data** is any other other arrangement of the data.

Table 3 is the tidy version of Table 1. Each row represents an observation, the `result` of one `treatment` on one `person`, and each column is a variable.

Tidy data makes it easy for an analyst or a computer to extract needed variables because it provides a standard way of structuring a dataset. Compare Table 3 to Table 1: in Table 1 you need to use different strategies to extract different variables. This slows analysis and invites errors. If you consider how many data analysis operations involve all of the values in a variable (every aggregation function), you can see how important it is to extract these values in a simple, standard way. Tidy data is particularly well suited for vectorised programming languages like R, because the layout ensures that values of different variables from the same observation are always paired.

While the order of variables and observations does not affect analysis, a good ordering makes it easier to scan the raw values. One way of organising variables is by their role in the analysis: are values fixed by the design of the data collection, or are they measured during the course of the experiment? Fixed variables describe the experimental design and are known in advance. Computer scientists often call fixed variables dimensions, and statisticians usually denote them with subscripts on random variables. Measured variables are what we actually measure in the study. Fixed variables should come first, followed by measured variables, each ordered so that related variables are contiguous. Rows can then be ordered by the first variable, breaking ties with the second and subsequent (fixed) variables. This is the convention adopted by all tabular displays in this paper.

3. Tidying messy datasets

Real datasets can, and often do, violate the three precepts of tidy data in almost every way imaginable. While occasionally you do get a dataset that you can start analysing immediately, this is the exception, not the rule. This section describes the five most common problems with messy datasets, along with their remedies:

- 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.

Surprisingly, most messy datasets, including types of messiness not explicitly described above, can be tidied with a small set of tools: melting, string splitting, and casting. The following sections illustrate each problem with a real dataset that I have encountered, and show how to tidy them. The complete datasets and the R code used to tidy them are available online at `https://github.com/hadley/tidy-data`, and in the online supplementary materials for this paper.

3.1. Column headers are values, not variable names

A common type of messy dataset is tabular data designed for presentation, where variables form both the rows and columns, and column headers are values, not variable names. While

I would call this arrangement messy, in some cases it can be extremely useful. It provides efficient storage for completely crossed designs, and it can lead to extremely efficient computation if desired operations can be expressed as matrix operations. This issue is discussed in depth in Section 6.

Table 4 shows a subset of a typical dataset of this form. This dataset explores the relationship between income and religion in the US. It comes from a report[1] produced by the Pew Research Center, an American think-tank that collects data on attitudes to topics ranging from religion to the internet, and produces many reports that contain datasets in this format.

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovah's Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

Table 4: The first ten rows of data on income and religion from the Pew Forum. Three columns, `$75-100k`, `$100-150k` and `>150k`, have been omitted

This dataset has three variables, `religion`, `income` and `frequency`. To tidy it, we need to **melt**, or stack it. In other words, we need to turn columns into rows. While this is often described as making wide datasets long or tall, I will avoid those terms because they are imprecise. Melting is parameterised by a list of columns that are already variables, or **colvar**s for short. The other columns are converted into two variables: a new variable called `column` that contains repeated column headings and a new variable called `value` that contains the concatenated data values from the previously separate columns. This is illustrated in Table 5 with a toy dataset. The result of melting is a **molten** dataset.

The Pew dataset has one colvar, `religion`, and melting yields Table 6. To better reflect their roles in this dataset, the `variable` column has been renamed to `income`, and the `value` column to `freq`. This form is tidy because each column represents a variable and each row represents an observation, in this case a demographic unit corresponding to a combination of `religion` and `income`.

Another common use of this data format is to record regularly spaced observations over time. For example, the Billboard dataset shown in Table 7 records the date a song first entered the Billboard Top 100. It has variables for `artist`, `track`, `date.entered`, `rank` and `week`. The rank in each week after it enters the top 100 is recorded in 75 columns, `wk1` to `wk75`. If a song is in the Top 100 for less than 75 weeks the remaining columns are filled with missing values. This form of storage is not tidy, but it is useful for data entry. It reduces duplication since

[1] http://religions.pewforum.org/pdf/comparison-Income%20Distribution%20of%20Religious%20Traditions.pdf

row	a	b	c
A	1	4	7
B	2	5	8
C	3	6	9

(a) Raw data

row	column	value
A	a	1
B	a	2
C	a	3
A	b	4
B	b	5
C	b	6
A	c	7
B	c	8
C	c	9

(b) Molten data

Table 5: A simple example of melting. (a) is melted with one colvar, row, yielding the molten dataset (b). The information in each table is exactly the same, just stored in a different way.

religion	income	freq
Agnostic	<$10k	27
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Agnostic	$30-40k	81
Agnostic	$40-50k	76
Agnostic	$50-75k	137
Agnostic	$75-100k	122
Agnostic	$100-150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Table 6: The first ten rows of the tidied Pew survey dataset on income and religion. The `column` has been renamed to `income`, and `value` to `freq`.

otherwise each song in each week would need its own row, and song metadata like title and artist would need to be repeated. This issue will be discussed in more depth in Section 3.4.

year	artist	track	time	date.entered	wk1	wk2	wk3
2000	2 Pac	Baby Don't Cry	4:22	2000-02-26	87	82	72
2000	2Ge+her	The Hardest Part Of ...	3:15	2000-09-02	91	87	92
2000	3 Doors Down	Kryptonite	3:53	2000-04-08	81	70	68
2000	98^0	Give Me Just One Nig...	3:24	2000-08-19	51	39	34
2000	A*Teens	Dancing Queen	3:44	2000-07-08	97	97	96
2000	Aaliyah	I Don't Wanna	4:15	2000-01-29	84	62	51
2000	Aaliyah	Try Again	4:03	2000-03-18	59	53	38
2000	Adams, Yolanda	Open My Heart	5:30	2000-08-26	76	76	74

Table 7: The first eight Billboard top hits for 2000. Other columns not shown are `wk4`, `wk5`, ..., `wk75`.

This dataset has colvars `year`, `artist`, `track`, `time`, and `date.entered`. Melting yields Table 8. I have also done a little cleaning as well as tidying: `column` has been converted to `week` by extracting the number, and `date` has been computed from `date.entered` and `week`.

year	artist	time	track	date	week	rank
2000	2 Pac	4:22	Baby Don't Cry	2000-02-26	1	87
2000	2 Pac	4:22	Baby Don't Cry	2000-03-04	2	82
2000	2 Pac	4:22	Baby Don't Cry	2000-03-11	3	72
2000	2 Pac	4:22	Baby Don't Cry	2000-03-18	4	77
2000	2 Pac	4:22	Baby Don't Cry	2000-03-25	5	87
2000	2 Pac	4:22	Baby Don't Cry	2000-04-01	6	94
2000	2 Pac	4:22	Baby Don't Cry	2000-04-08	7	99
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-02	1	91
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-09	2	87
2000	2Ge+her	3:15	The Hardest Part Of ...	2000-09-16	3	92
2000	3 Doors Down	3:53	Kryptonite	2000-04-08	1	81
2000	3 Doors Down	3:53	Kryptonite	2000-04-15	2	70
2000	3 Doors Down	3:53	Kryptonite	2000-04-22	3	68
2000	3 Doors Down	3:53	Kryptonite	2000-04-29	4	67
2000	3 Doors Down	3:53	Kryptonite	2000-05-06	5	66

Table 8: First fifteen rows of the tidied billboard dataset. The `date` column does not appear in the original table, but can be computed from `date.entered` and `week`.

3.2. Multiple variables stored in one column

After melting, the `column` variable names often becomes a combination of multiple underlying variable names. This is illustrated by the tuberculosis (TB) dataset, a sample of which is shown in Table 9. This dataset comes from the World Health Organisation, and records the counts of confirmed tuberculosis cases by `country`, `year`, and demographic group. The demographic groups are broken down by `sex` (m, f) and `age` (0–14, 15–25, 25–34, 35–44, 45–54, 55–64, unknown).

country	year	m014	m1524	m2534	m3544	m4554	m5564	m65	mu	f014
AD	2000	0	0	1	0	0	0	0	—	—
AE	2000	2	4	4	6	5	12	10	—	3
AF	2000	52	228	183	149	129	94	80	—	93
AG	2000	0	0	0	0	0	0	1	—	1
AL	2000	2	19	21	14	24	19	16	—	3
AM	2000	2	152	130	131	63	26	21	—	1
AN	2000	0	0	1	2	0	0	0	—	0
AO	2000	186	999	1003	912	482	312	194	—	247
AR	2000	97	278	594	402	419	368	330	—	121
AS	2000	—	—	—	—	1	1	—	—	—

Table 9: Original TB dataset. Corresponding to each 'm' column for males, there is also an 'f' column for females, `f1524`, `f2534` and so on. These are not shown to conserve space. Note the mixture of 0s and missing values (—). This is due to the data collection process and the distinction is important for this dataset.

Column headers in this format are often separated by some character (`.`, `-`, `_`, `:`). While the string can be broken into pieces using that character as a divider, in other cases, such as for this dataset, more careful string processing is required. For example, the variable names can be matched to a lookup table that converts single compound value into multiple component values.

Table 10(a) shows the results of melting the TB dataset, and Table 10(b) shows the results of splitting the single column `column` into two real variables: `age` and `sex`.

Storing the values in this form resolves another problem in the original data. We want to compare rates, not counts. But to compute rates, we need to know the population. In the original format, there is no easy way to add a population variable. It has to be stored in a separate table, which makes it hard to correctly match populations to counts. In tidy form, adding variables for population and rate is easy. They are just additional columns.

3.3. Variables are stored in both rows and columns

The most complicated form of messy data occurs when variables are stored in both rows and columns. Table 11 shows daily weather data from the Global Historical Climatology Network for one weather station (MX17004) in Mexico for five months in 2010. It has variables in individual columns (`id`, `year`, `month`), spread across columns (`day`, d1–d31) and across rows (`tmin`, `tmax`) (minimum and maximum temperature). Months with less than 31 days have structural missing values for the last day(s) of the month. The `element` column is not a variable; it stores the names of variables.

To tidy this dataset we first melt it with colvars `id`, `year`, `month` and the column that contains variable names, `element`. This yields Table 12(a). For presentation, we have dropped the missing values, making them implicit rather than explicit. This is permissible because we know how many days are in each month and can easily reconstruct the explicit missing values.

This dataset is mostly tidy, but we have two variables stored in rows: `tmin` and `tmax`, the

country	year	column	cases
AD	2000	m014	0
AD	2000	m1524	0
AD	2000	m2534	1
AD	2000	m3544	0
AD	2000	m4554	0
AD	2000	m5564	0
AD	2000	m65	0
AE	2000	m014	2
AE	2000	m1524	4
AE	2000	m2534	4
AE	2000	m3544	6
AE	2000	m4554	5
AE	2000	m5564	12
AE	2000	m65	10
AE	2000	f014	3

(a) Molten data

country	year	sex	age	cases
AD	2000	m	0-14	0
AD	2000	m	15-24	0
AD	2000	m	25-34	1
AD	2000	m	35-44	0
AD	2000	m	45-54	0
AD	2000	m	55-64	0
AD	2000	m	65+	0
AE	2000	m	0-14	2
AE	2000	m	15-24	4
AE	2000	m	25-34	4
AE	2000	m	35-44	6
AE	2000	m	45-54	5
AE	2000	m	55-64	12
AE	2000	m	65+	10
AE	2000	f	0-14	3

(b) Tidy data

Table 10: Tidying the TB dataset requires first melting, and then splitting the `column` column into two variables: `sex` and `age`.

type of observation. Not shown in this example are the other meteorological variables `prcp` (precipitation) and `snow` (snowfall). Fixing this requires the cast, or unstack, operation. This performs the inverse of melting by rotating the `element` variable back out into the columns (Table 12(b)). This form is tidy. There is one variable in each column, and each row represents a day's observations. The cast operation is described in depth in Wickham (2007).

3.4. Multiple types in one table

Datasets often involve values collected at multiple levels, on different types of observational units. During tidying, each type of observational unit should be stored in its own table. This is closely related to the idea of database normalisation, where each fact is expressed in only one place. If this is not done, it's possible for inconsistencies to occur.

The Billboard dataset described in Table 8 actually contains observations on two types of observational units: the song and its rank in each week. This manifests itself through the duplication of facts about the song: `artist` and `time` are repeated for every song in each week. The billboard dataset needs to be broken down into two datasets: a song dataset which stores `artist`, `song name` and `time`, and a ranking dataset which gives the `rank` of the `song` in each `week`. Table 13 shows these two datasets. You could also imagine a week dataset which would record background information about the week, maybe the total number of songs sold or similar demographic information.

Normalisation is useful for tidying and eliminating inconsistencies. However, there are few data analysis tools that work directly with relational data, so analysis usually also requires denormalisation or the merging the datasets back into one table.

id	year	month	element	d1	d2	d3	d4	d5	d6	d7	d8
MX17004	2010	1	tmax	—	—	—	—	—	—	—	—
MX17004	2010	1	tmin	—	—	—	—	—	—	—	—
MX17004	2010	2	tmax	—	27.3	24.1	—	—	—	—	—
MX17004	2010	2	tmin	—	14.4	14.4	—	—	—	—	—
MX17004	2010	3	tmax	—	—	—	—	32.1	—	—	—
MX17004	2010	3	tmin	—	—	—	—	14.2	—	—	—
MX17004	2010	4	tmax	—	—	—	—	—	—	—	—
MX17004	2010	4	tmin	—	—	—	—	—	—	—	—
MX17004	2010	5	tmax	—	—	—	—	—	—	—	—
MX17004	2010	5	tmin	—	—	—	—	—	—	—	—

Table 11: Original weather dataset. There is a column for each possible day in the month. Columns `d9` to `d31` have been omitted to conserve space.

id	date	element	value
MX17004	2010-01-30	tmax	27.8
MX17004	2010-01-30	tmin	14.5
MX17004	2010-02-02	tmax	27.3
MX17004	2010-02-02	tmin	14.4
MX17004	2010-02-03	tmax	24.1
MX17004	2010-02-03	tmin	14.4
MX17004	2010-02-11	tmax	29.7
MX17004	2010-02-11	tmin	13.4
MX17004	2010-02-23	tmax	29.9
MX17004	2010-02-23	tmin	10.7

(a) Molten data

id	date	tmax	tmin
MX17004	2010-01-30	27.8	14.5
MX17004	2010-02-02	27.3	14.4
MX17004	2010-02-03	24.1	14.4
MX17004	2010-02-11	29.7	13.4
MX17004	2010-02-23	29.9	10.7
MX17004	2010-03-05	32.1	14.2
MX17004	2010-03-10	34.5	16.8
MX17004	2010-03-16	31.1	17.6
MX17004	2010-04-27	36.3	16.7
MX17004	2010-05-27	33.2	18.2

(b) Tidy data

Table 12: (a) Molten weather dataset. This is almost tidy, but instead of values, the `element` column contains names of variables. Missing values are dropped to conserve space. (b) Tidy weather dataset. Each row represents the meteorological measurements for a single day. There are two measured variables, minimum (`tmin`) and maximum (`tmax`) temperature; all other variables are fixed.

id	artist	track	time
1	2 Pac	Baby Don't Cry	4:22
2	2Ge+her	The Hardest Part Of ...	3:15
3	3 Doors Down	Kryptonite	3:53
4	3 Doors Down	Loser	4:24
5	504 Boyz	Wobble Wobble	3:35
6	98^0	Give Me Just One Nig...	3:24
7	A*Teens	Dancing Queen	3:44
8	Aaliyah	I Don't Wanna	4:15
9	Aaliyah	Try Again	4:03
10	Adams, Yolanda	Open My Heart	5:30
11	Adkins, Trace	More	3:05
12	Aguilera, Christina	Come On Over Baby	3:38
13	Aguilera, Christina	I Turn To You	4:00
14	Aguilera, Christina	What A Girl Wants	3:18
15	Alice Deejay	Better Off Alone	6:50

id	date	rank
1	2000-02-26	87
1	2000-03-04	82
1	2000-03-11	72
1	2000-03-18	77
1	2000-03-25	87
1	2000-04-01	94
1	2000-04-08	99
2	2000-09-02	91
2	2000-09-09	87
2	2000-09-16	92
3	2000-04-08	81
3	2000-04-15	70
3	2000-04-22	68
3	2000-04-29	67
3	2000-05-06	66

Table 13: Normalised billboard dataset split up into song dataset (left) and rank dataset (right). First 15 rows of each dataset shown; `genre` omitted from song dataset, `week` omitted from rank dataset.

3.5. One type in multiple tables

It's also common to find data values about a single type of observational unit spread out over multiple tables or files. These tables and files are often split up by another variable, so that each represents a single year, person, or location. As long as the format for individual records is consistent, this is an easy problem to fix:

1. Read the files into a list of tables.
2. For each table, add a new column that records the original file name (because the file name is often the value of an important variable).
3. Combine all tables into a single table.

The **plyr** package makes this a straightforward task in R. The following code generates a vector of file names in a directory (`data/`) which match a regular expression (ends in `.csv`). Next we name each element of the vector with the name of the file. We do this because **plyr** will preserve the names in the following step, ensuring that each row in the final data frame is labelled with its source. Finally, `ldply()` loops over each path, reading in the csv file and combining the results into a single data frame.

```
R> paths <- dir("data", pattern = "\\.csv$", full.names = TRUE)
R> names(paths) <- basename(paths)
R> ldply(paths, read.csv, stringsAsFactors = FALSE)
```

Once you have a single table, you can perform additional tidying as needed. An example of this type of cleaning can be found at `https://github.com/hadley/data-baby-names` which takes 129 yearly baby name tables provided by the US Social Security Administration and combines them into a single file.

A more complicated situation occurs when the dataset structure changes over time. For example, the datasets may contain different variables, the same variables with different names, different file formats, or different conventions for missing values. This may require you to tidy each file to individually (or, if you're lucky, in small groups) and then combine them once tidied. An example of this type of tidying is illustrated in `https://github.com/hadley/data-fuel-economy`, which shows the tidying of EPA fuel economy data for over 50,000 cars from 1978 to 2008. The raw data is available online, but each year is stored in a separate file and there are four major formats with many minor variations, making tidying this dataset a considerable challenge.

4. Tidy tools

Once you have a tidy dataset, what can you do with it? Tidy data is only worthwhile if it makes analysis easier. This section discusses tidy tools, tools that take tidy datasets as input and return tidy datasets as output. Tidy tools are useful because the output of one tool can be used as the input to another. This allows you to simply and easily compose multiple tools to solve a problem. Tidy data also ensures that variables are stored in a consistent, explicit manner. This makes each tool simpler, because it doesn't need a Swiss Army knife of parameters for dealing with different dataset structures.

Tools can be messy for two reasons: either they take messy datasets as input (messy-input tools) or they produce messy datasets as output (messy-output tools). Messy-input tools are typically more complicated than tidy-input tools because they need to include some parts of the tidying process. This can be useful for common types of messy datasets, but it typically makes the function more complex, harder to use and harder to maintain. Messy-output tools are frustrating and slow down analysis because they can not be easily composed and you must constantly think about how to convert from one format to another. We'll see examples of both in the following sections.

Next, I give examples of tidy and messy tools for three important components of analysis: data manipulation, visualisation and modelling. I will focus particularly on tools provided by R (R Development Core Team 2011), because it has many existing tidy tools, but I will also touch on other statistical programming environments.

4.1. Manipulation

Data manipulation includes variable-by-variable transformation (e.g., `log` or `sqrt`), as well as aggregation, filtering and reordering. In my experience, these are the four fundamental verbs of data manipulation:

- Filter: subsetting or removing observations based on some condition.
- Transform: adding or modifying variables. These modifications can involve either a single variable (e.g., log-transformation), or multiple variables (e.g., computing density from weight and volume).

- Aggregate: collapsing multiple values into a single value (e.g., by summing or taking means).
- Sort: changing the order of observations.

All these operations are made easier when there is a consistent way to refer to variables. Tidy data provides this because each variable resides in its own column.

In R, filtering and transforming are performed by the base R functions `subset()` and `transform()`. These are input and output-tidy. The `aggregate()` function performs group-wise aggregation. It is input-tidy. Provided that a single aggregation method is used, it is also output-tidy . The **plyr** package provides tidy `summarise()` and `arrange()` functions for aggregation and sorting.

The four verbs can be, and often are, modified by the "by" preposition. We often need group-wise aggregates, transformations and subsets, to pick the biggest in each group, to average over replicates and so on. Combining each of the four verbs with a by operator allows them to operate on subsets of a data frame at a time. Most SAS PROCs possess a BY statement which allows the operation to be performed by group, and are generally input-tidy. Base R possesses a `by()` function, which is input-tidy, but not output-tidy, because it produces a list. The `ddply()` function from the **plyr** package is a tidy alternative.

Other tools are needed when we have multiple datasets. An advantage of tidy data is the ease with which it can be combined with other tidy datasets. All that is needed is a join operator that works by matching common variables and adding new columns. This is implemented in the `merge()` function in base R, or the `join()` function in **plyr**. Compare these operators with the difficulty of combining datasets stored in arrays. This task typically requires painstaking alignment before matrix operations can be used, which can can make errors very hard to detect.

4.2. Visualisation

Tidy visualisation tools only need to be input-tidy as their output is visual. Domain specific languages work particularly well for the visualisation of tidy datasets because they can describe a visualisation as a mapping between variables and aesthetic properties of the graph (e.g., position, size, shape and colour). This is the idea behind the grammar of graphics (Wilkinson 2005), and the layered grammar of graphics (Wickham 2010), an extension tailored specifically for R.

Most graphical tools in R are input-tidy, including the **base** `plot()` function, the **lattice** family of plots (Sarkar 2008) and **ggplot2** (Wickham 2009). Some specialised tools exist for visualising messy datasets. Some base R functions like `barplot()`, `matplot()`, `dotchart()`, and `mosaicplot()`, work with messy datasets where variables are spread out over multiple columns. Similarly, the parallel coordinates plot (Wegman 1990; Inselberg 1985) can be used to create time series plots for messy datasets where each time point is a column.

4.3. Modelling

Modelling is the driving inspiration of this work because most modelling tools work best with tidy datasets. Every statistical language has a way of describing a model as a connection among different variables, a domain specific language that connects responses to predictors:

id	x	y
1	22.19	24.05
2	19.82	22.91
3	19.81	21.19
4	17.49	18.59
5	19.44	19.85

(a) Data for paired t-test

id	variable	value
1	x	22.19
2	x	19.82
3	x	19.81
4	x	17.49
5	x	19.44
1	y	24.05
2	y	22.91
3	y	21.19
4	y	18.59
5	y	19.85

(b) Data for mixed effects model

Table 14: Two data sets for performing the same test.

- R (`lm()`): `y ~ a + b + c * d`.
- SAS (`PROC GLM`): `y = a + b + c + d + c * d`.
- SPSS (`glm`): `y BY a b c d / DESIGN a b c d c * d`.
- Stata (`regress`): `y a b c#d`.

This is not to say that tidy data is the format used internally to compute the regression. Significant transformations take place to produce a numeric matrix that can easily be fed to standard linear algebra routines. Common transformations include adding an intercept column (a column of ones), turning categorical variables into multiple binary dummy variables, and projecting data onto the appropriate basis of a spline function.

There have been some attempts to adapt modelling functions for specific types of messy datasets. For example, in SAS's `proc glm`, multiple variables on the response side of the equation will be interpreted as repeated measures if the REPEATED keyword is present. For the raw Billboard data, we could construct a model of the form `wk1-wk76 = track` instead of `rank = week * track` on the tidy data.

Another interesting example is the classic paired t-test, which can be computed in two ways depending on how the data is stored. If the data is stored as in Table 14(a), then a paired t-test is just a t-test applied to the mean difference between x and y. If it is stored in the form of Table 14(b), then an equivalent result can be produced by fitting a mixed effects model, with a fixed dummy variable representing the `variable`, and a random intercept for each id. (In R's lmer4 notation, this is expressed as `value ~ variable + (1 | id)`). Either way of modelling the data yields the same result. Without more information we can't say which form of the data is tidy: if x and y represent length of left and right arms, then Table 14(a) is tidy, if x and y represent measurements on day 1 and day 10, then Table 14(b) is tidy.

While model inputs usually require tidy inputs, such attention to detail doesn't carry over to model outputs. Outputs such as predictions and estimated coefficients aren't always tidy.

This makes it more difficult to combine results from multiple models. For example, in R, the default representation of model coefficients is not tidy because it does not have an explicit variable that records the variable name for each estimate, they are instead recorded as row names. In R, row names must be unique, so combining coefficients from many models (e.g., from bootstrap resamples, or subgroups) requires workarounds to avoid losing important information. This knocks you out of the flow of analysis and makes it harder to combine the results from multiple models. I'm not currently aware of any packages that resolve this problem.

5. Case study

The following case study illustrates how tidy data and tidy tools make data analysis easier by easing the transitions between manipulation, visualisation and modelling. You will not see any code that exists solely to get the output of one function into the right format to input to another. I'll show the R code that performs the analysis, but even if you're not familiar with R or the exact idioms I use, I've tried to make it easy to understand by tightly interleaving code, results and explanation.

The case study uses individual-level mortality data from Mexico. The goal is to find causes of death with unusual temporal patterns within a day. Figure 1 shows the temporal pattern, the number of deaths per hour, for all causes of death. My goal is to find the diseases that differ most from this pattern.

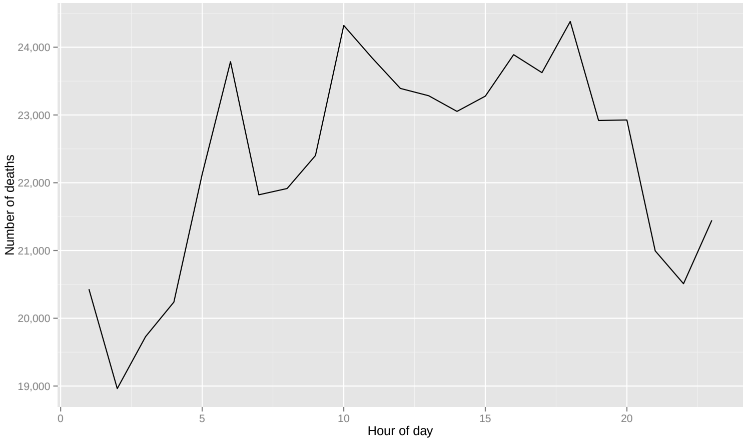


Figure 1: Temporal pattern of all causes of death.

The full dataset has information on 539,530 deaths in Mexico in 2008 and 55 variables, including the the location and time of death, the cause of death, and demographics of the deceased. Table 15 shows a small sample of the dataset, focussing on variables related to time of death (`year`, `month`, `day` and `hour`), and cause of death (`cod`).

To achieve our goal of finding unusual temporal patterns, we do the following. First, we count the number of deaths in each hour (`hod`) for each cause (`cod`) with the tidy `count` function.

```
R> hod2 <- count(deaths, c("hod", "cod"))
```

yod	mod	dod	hod	cod
2008	1	1	1	B20
2008	1	2	4	I67
2008	1	3	8	I50
2008	1	4	12	I50
2008	1	5	16	K70
2008	1	6	18	I21
2008	1	7	20	I21
2008	1	8	—	K74
2008	1	10	5	K74
2008	1	11	9	I21
2008	1	12	15	I25
2008	1	13	20	R54
2008	1	15	2	I61
2008	1	16	7	I21
2008	1	17	13	I21

Table 15: A sample of 16 rows and 5 columns from the original dataset of 539,530 rows and 55 columns.

Then we remove missing (and hence uninformative for our purpose) values with `subset`.

```
R> hod2 <- subset(hod2, !is.na(hod))
```

This gives Table 16(a). To provide informative labels for disease, we next join the dataset to the `codes` dataset, connected by the `cod` variable. This adds a new variable, `disease`, shown in Table 16(b).

```
R> hod2 <- join(hod2, codes, by = "cod")
```

The total deaths for each cause varies over several orders of magnitude: there are 46,794 deaths from heart attack but only 10 from avalanche. This means that rather than the total number, it makes more sense to compare the proportion of deaths in each hour. We compute this by breaking the dataset down by `cod`, and then `transform()`ing to add a new `prop` column, the hourly frequency divided by the total number of deaths from that cause. This new column is Table 16(c).

`ddply()` breaks down the first argument (`hod2`) by its second (the `cod` variable), and then applies the third argument (`transform`) to each resulting piece. The fourth argument (`prop = freq / sum(freq)`) is then passed on to `transform()`.

```
R> hod2 <- ddply(hod2, "cod", transform, prop = freq / sum(freq))
```

We then compute the overall average death rate for each hour, and merge that back into the original dataset. This yields Table 16(d). By comparing `prop` to `prop_all`, we can easily compare each disease with the overall incidence pattern.

First, we work out the number of people dying each hour. We break down `hod2` by `hod`, and compute the total for each cause of death.

```
R> overall <- ddply(hod2, "hod", summarise, freq_all = sum(freq))
```

Next, we work out the overall proportion of people dying each hour:

```
R> overall <- transform(overall, prop_all = freq_all / sum(freq_all))
```

Finally, we join the overall dataset with the individual dataset to make it easier to compare the two:

```
R> hod2 <- join(hod2, overall, by = "hod")
```

hod	cod	freq	disease	prop	freq_all	prop_all
8	B16	4	Acute hepatitis B	0.04	21915	0.04
8	E84	3	Cystic fibrosis	0.03	21915	0.04
8	I21	2205	Acute myocardial infarction	0.05	21915	0.04
8	N18	315	Chronic renal failure	0.04	21915	0.04
9	B16	7	Acute hepatitis B	0.07	22401	0.04
9	E84	1	Cystic fibrosis	0.01	22401	0.04
9	I21	2209	Acute myocardial infarction	0.05	22401	0.04
9	N18	333	Chronic renal failure	0.04	22401	0.04
10	B16	10	Acute hepatitis B	0.10	24321	0.05
10	E84	7	Cystic fibrosis	0.07	24321	0.05
10	I21	2434	Acute myocardial infarction	0.05	24321	0.05
10	N18	343	Chronic renal failure	0.04	24321	0.05
11	B16	6	Acute hepatitis B	0.06	23843	0.05
11	E84	3	Cystic fibrosis	0.03	23843	0.05
11	I21	2128	Acute myocardial infarction	0.05	23843	0.05
	(a)		(b)	(c)	(d)	

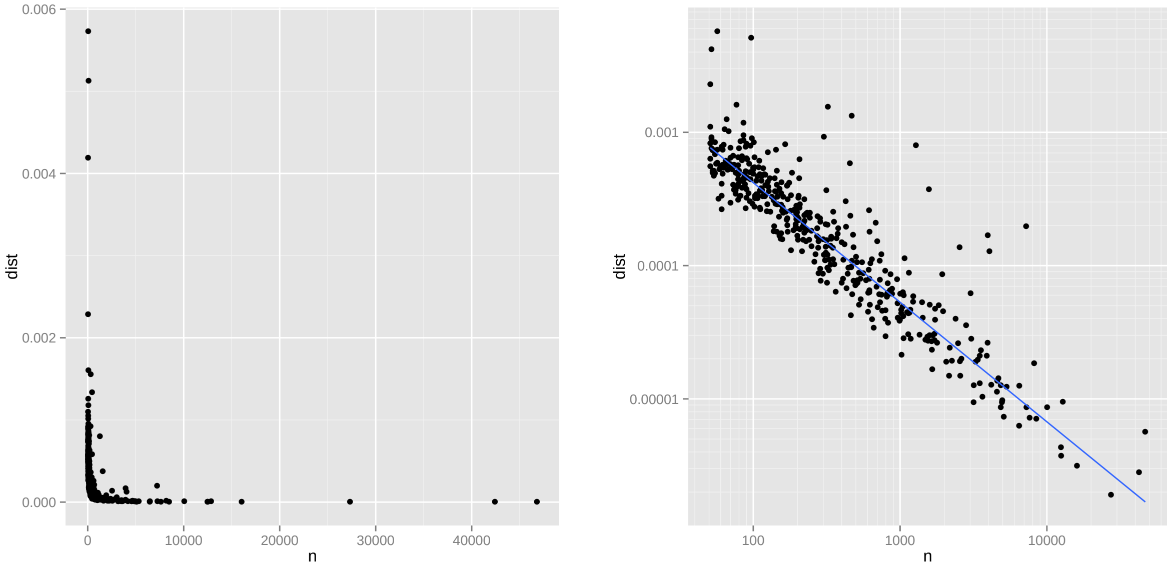
Table 16: A sample of four diseases and four hours from `hod2` data frame.

Next we compute a distance between the temporal pattern of each cause of death and the overall temporal pattern. There are many ways to measure this distance, but I found a simple mean squared deviation to be revealing. We also record the sample size, the total number of deaths from that cause. To ensure that the diseases we consider are sufficiently representative we'll only work with diseases with more than 50 total deaths (∼2/hour).

```
R> devi <- ddply(hod2, "cod", summarise,
R>   n = sum(freq),
R>   dist = mean((prop - prop_all)^2))
R>
R> devi <- subset(devi, n > 50)
```

We don't know the variance characteristics of this estimator, but we can explore it visually by plotting `n` vs. `deviation`, Figure 2(a). On a linear scale, the plot shows little, except that variability decreases with sample size. But on the log-log scale, Figure 2(b), there is a clear pattern. This is particularly easy to see the pattern when we add the line of best fit from a robust linear model.

```
R> ggplot(data = devi, aes(x = n, y = dist) + geom_point()
R>
R> last_plot() +
R>   scale_x_log10() +
R>   scale_y_log10() +
R>   geom_smooth(method = "rlm", se = F)
```



(a) Linear scales

(b) Log scales

Figure 2: (a) Plot of n vs deviation. Variability of deviation is dominated by sample size: small samples have large variability. (b) Log-log plot makes it easy to see the pattern of variation as well as unusually high values. The blue line is a robust line of best fit.

We are interested in points that have high y-values, relative to their x-neighbours. Controlling for the number of deaths, these points represent the diseases which depart the most from the overall pattern.

To find these unusual points, we fit a robust linear model and plot the residuals, Figure 3. The plot shows an empty region around a residual of 1.5. So somewhat arbitrarily, we'll select those diseases with a residual greater than 1.5. We do this in two steps: first, we select the appropriate rows from `devi` (one row per disease), and then we find the matching temporal course information from the original summary dataset (24 rows per disease).

```
R> devi$resid <- resid(rlm(log(dist) ˜ log(n), data = devi))
R> unusual <- subset(devi, resid > 1.5)
R> hod_unusual <- match_df(hod2, unusual)
```

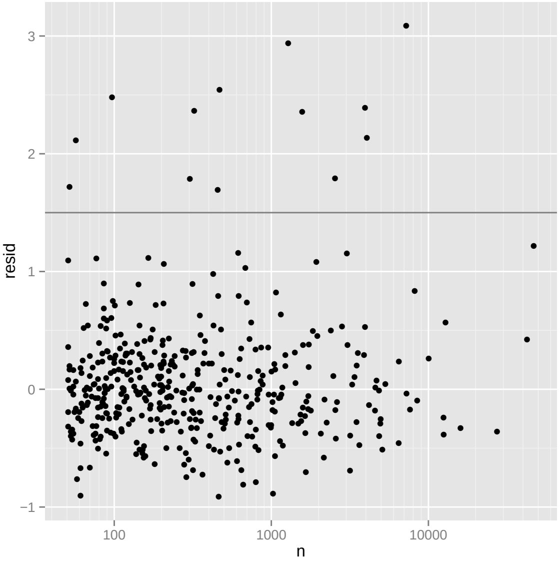


Figure 3: Residuals from a robust linear model predicting log($dist$) by log(n). Horizontal line at 1.5 shows threshold for further exploration.

Finally, we plot the temporal course for each unusual cause, Figure 4. We split the diseases into two plots because of differences in variability. The top plot shows diseases with over 350 deaths and the bottom with under 350. The causes of death fall into three main groups: murder, drowning, and transportation related. Murder is more common at night, drowning in the afternoon, and transportation related deaths during commute times. The pale gray line in the background shows the temporal course across all diseases.

```
R> ggplot(data = subset(hod_unusual, n > 350), aes(x = hod, y = prop)) +
R>   geom_line(aes(y = prop_all), data = overall, colour = "grey50") +
R>   geom_line() +
R>   facet_wrap(~ disease, ncol = 3)
```

6. Discussion

Data cleaning is an important problem, but it is an uncommon subject of study in statistics. This paper carves out a small but important subset of data cleaning that I've called data tidying: structuring datasets to facilitate manipulation, visualisation and modelling. There is still much work to be done. Incremental improvements will happen as our understanding of tidy data and tidy tools improves, and as we improve our ability to reduce the friction of getting data into a tidy form.

Bigger improvements may be possible by exploring alternative formulations of tidiness. There is a chicken-and-egg problem with tidy data: if tidy data is only as useful as the tools that work with it, then tidy tools will be inextricably linked to tidy data. This makes it easy to get stuck in a local maxima where independently changing data structures or data tools will not improve workflow. Breaking out of this local maxima is hard. It requires long-term concerted

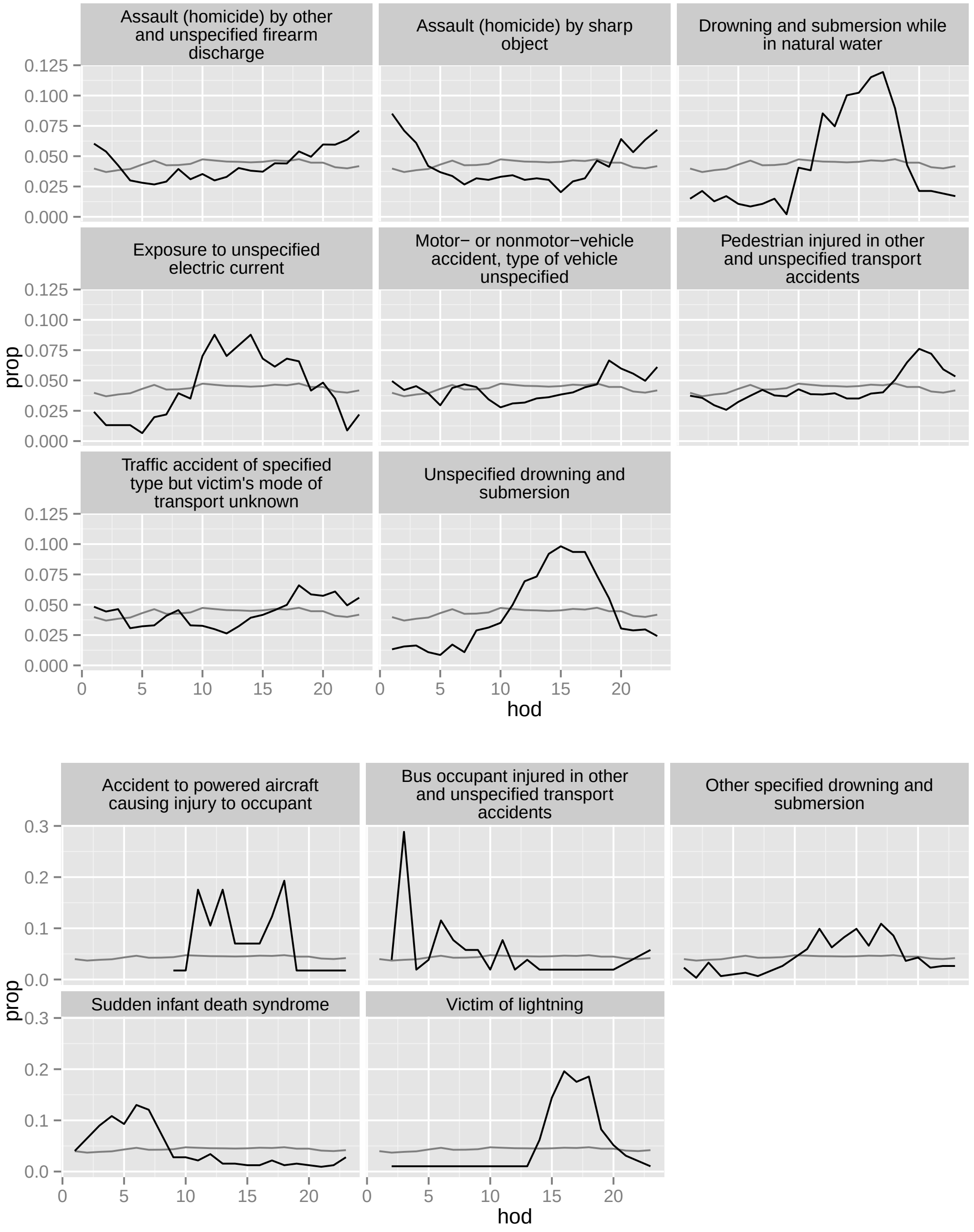


Figure 4: Causes of death with unusual temporal courses. Overall hourly death rate shown in grey. (Top) Causes of death with more than 350 deaths over a year. (Bottom) Causes of death with fewer than 350 deaths. Note that the y-axes are on different scales.

effort with the prospect of many false starts. While I hope that the tidy data framework is not one of those false starts, I also don't see it as the final solution. I hope others will build on this framework to develop even better data storage strategies and better tools.

Surprisingly, I have found few principles to guide the design of tidy data, which acknowledge both statistical and cognitive factors. To date, my work has been driven by my experience doing data analysis, my knowledge of relational database design, and my own rumination on the tools of data analysis. The human factors, user-centered design, and human-computer interaction communities may be able to add to this conversation, but the design of data and tools to work with it has not been an active research topic in those fields. In the future, I hope to use methodologies from these fields (user-testing, ethnography, talk-aloud protocols) to improve our understanding of the cognitive side of data analysis, and to further improve our ability to design appropriate tools.

Other formulations of tidy data are possible. For example, it would be possible to construct a set of tools for dealing with values stored in multidimensional arrays. This is a common storage format for large biomedical datasets generated by microarrays or fMRI's. It's also necessary for many multivariate methods based on matrix manipulation. Fortunately, because there are many efficient tools for working with high-dimensional arrays, even sparse ones, such an array-tidy format is not only likely to be quite compact and efficient, it should also be able to easily connect with the mathematical basis of statistics. This, in fact, is the approach taken by the Pandas python data analysis library (McKinney 2010). Even more interestingly, we could consider tidy tools that can ignore the underlying data representation and automatically choose between array-tidy and dataframe-tidy formats to optimise memory usage and performance.

Apart from tidying, there are many other tasks involved in cleaning data: parsing dates and numbers, identifying missing values, correcting character encodings (for international data), matching similar but not identical values (created by typos), verifying experimental design, and filling in structural missing values, not to mention model-based data cleaning that identifies suspicious values. Can we develop other frameworks to make these tasks easier?

7. Acknowledgements

This work wouldn't be possible without the many conversations I've had about data and how to deal with them statistically. I'd particularly like to thank Phil Dixon, Di Cook, and Heike Hofmann, who have put up with numerous questions over the years. I'd also like to thank the users of the **reshape** package who have provided many challenging problems, and my students who continue to challenge me to explain what I know in a way that they can understand. I'd also like to thank Bob Muenchen, Burt Gunter, Nick Horton and Garrett Grolemund who gave detailed comments on earlier drafts, and to particularly thank Ross Gayler who provided the nice example of the challenges of defining a variable and Ben Bolker who showed me the natural equivalence between a paired t-test and a mixed effects model.

References

Codd EF (1990). *The Relational Model for Database Management: Version 2*. Addison-Wesley

Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-14192-2.

Dasu T, Johnson T (2003). *Exploratory Data Mining and Data Cleaning.* Wiley-IEEE.

Inselberg A (1985). "The Plane with Parallel Coordinates." *The Visual Computer*, **1**, 69–91.

Kandel S, Paepcke A, Hellerstein J, Heer J (2011). "Wrangler: Interactive Visual Specification of Data Transformation Scripts." In *ACM Human Factors in Computing Systems (CHI).* URL `http://vis.stanford.edu/papers/wrangler`.

Lakshmanan L, Sadri F, Subramanian I (1996). "SchemaSQL-a language for interoperability in relational multi-database systems." In *Proceedings of the International Conference on Very Large Data Bases*, pp. 239–250. ISSN 1047-7349.

McKinney W (2010). "Data Structures for Statistical Computing in Python." In S van der Walt, J Millman (eds.), *Proceedings of the 9th Python in Science Conference*, pp. 51 – 56.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.R-project.org/`.

Raman V, Hellerstein J (2001). "Potter's Wheel: An Interactive Data Cleaning System." In *Proceedings of the International Conference on Very Large Data Bases*, pp. 381–390. ISSN 1047-7349.

Sarkar D (2008). ***Lattice**: Multivariate Data Visualization with R.* Springer-Verlag.

Wegman EJ (1990). "Hyperdimensional Data Analysis Using Parallel Coordinates." *Journal of the American Statistical Association*, **85**(411), 664–675.

Wickham H (2007). "Reshaping Data with the **Reshape** Package." *Journal of Statistical Software*, **21**(12), 1–20. URL `http://www.jstatsoft.org/v21/i12/paper`.

Wickham H (2009). ***ggplot2**: Elegant Graphics for Data Analysis.* useR. Springer-Verlag.

Wickham H (2010). "A Layered Grammar of Graphics." *Journal of Computational and Graphical Statistics*, **19**(1), 3–28.

Wickham H (2011). "The Split-apply-combine Strategy for Data Analysis." *Journal of Statistical Software*, **40**(1), 1–29. URL `http://www.jstatsoft.org/v40/i01/`.

Wilkinson L (2005). *The Grammar of Graphics.* Statistics and Computing, 2nd edition. Springer-Verlag.

Affiliation:

Hadley Wickham
Chief Scientist, RStudio
Adjunct Professor, Rice University
E-mail: h.wickham@gmail.com
URL: http://had.co.nz

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd