Chapter 18

Data Cleaning

Contents

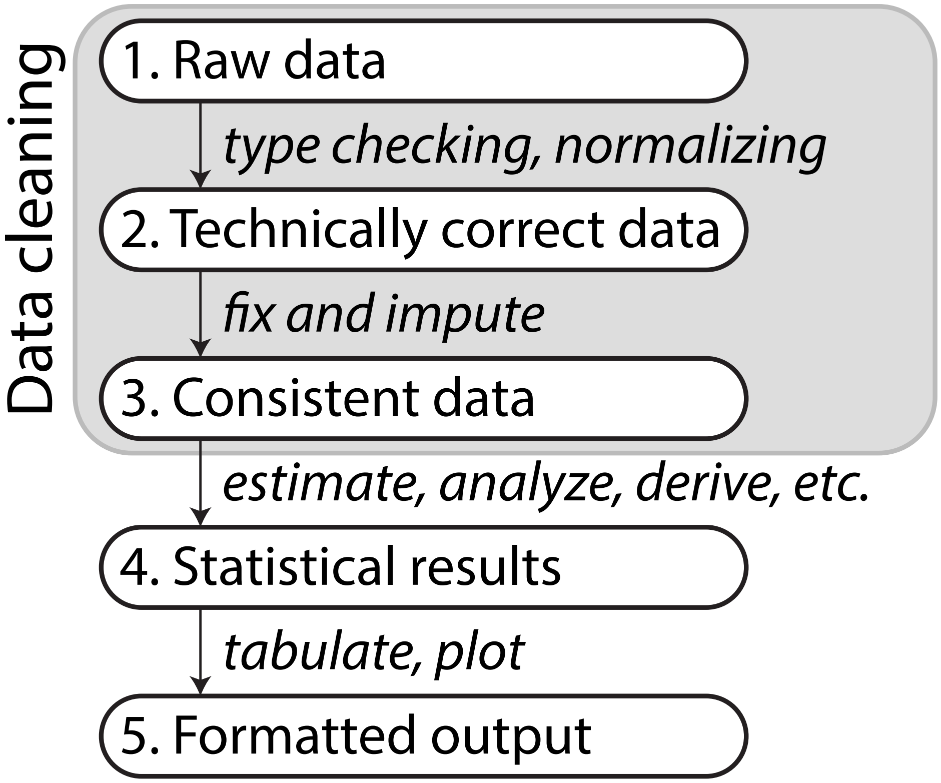
Data cleaning[1], or data preparation, is an essential part of statistical analysis[2]. In fact, in practice it is often more time-consuming than the statistical analysis itself. Data cleaning may profoundly influence the statistical statements based on the data. Typical actions like imputation or outlier 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] Content in this chapter is derived with permission from Statistics Netherlands at `http://cran.r-project.org/doc/contrib/de_Jonge+van_der_Loo-Introduction_to_data_cleaning_with_R.pdf`

[2] Note that this chapter has not been updated to use tidyverse syntax or ideas.

18.1 The five steps of statistical analysis

Statistical analysis can be viewed as the result of a number of value-increasing data processing steps.



Each box represents data in a certain state while each arrow represents the activities needed to get from one state to the other.

1. Raw Data The data "as is" may lack headers, contain wrong data types (e.g., numbers stored as strings), wrong category labels, unknown or unexpected character encoding and so on. Reading such files into an R `data.frame` directly is either difficult or impossible without some sort of preprocessing.

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

3. Consistent data The 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.

4. Statistical results The results of the analysis have been produced and can be stored for reuse.

5. Formatted output The results in tables and figures ready to include in statistical reports or publications.

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

18.2 R background review

18.2.1 Variable types

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)
integer Integer data (whole numbers)
factor Categorical data (simple classifications, like gender)
ordered Ordinal data (ordered classifications, like educational level)
character Character data (strings)
raw Binary data (rarely used)

All basic operations in R work element-wise on vectors where the shortest argument is recycled if necessary. Why does the following code work the way it does?

```
library(tidyverse)

# vectors have variables of _one_ type
c(1, 2, "three")
## [1] "1"     "2"     "three"
# shorter arguments are recycled
(1:3) * 2
## [1] 2 4 6
(1:4) * c(1, 2)
## [1] 1 4 3 8
# warning! (why?)
(1:4) * (1:3)
## Warning in (1:4) * (1:3):  longer object length is not a multiple of shorter object length
## [1] 1 4 9 4
```

18.2.2 Special values and value-checking functions

Below are the definitions and some illustrations of the special values `NA`, `NULL`, ±`Inf`, and `NaN`.

- `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
# use is.na() to detect NAs
is.na(c(1, NA, 3))
```

- NULL Think of NULL as the empty set from mathematics; it has no class (its class is NULL) and has length 0 so it does not take up any space in a vector.

```
length(c(1, 2, NULL, 4))
sum(c(1, 2, NULL, 4))
x <- NULL
length(x)
c(x, 2)
# use is.null() to detect NULL variables
is.null(x)
```

- Inf Stands for "infinity" and only applies to vectors of class numeric (not integer). 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.

```
pi/0
2 * Inf
Inf - 1e+10
Inf + Inf
3 < -Inf
Inf == Inf
# use is.infinite() to detect Inf variables
is.infinite(-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.

```
NaN + 1
exp(NaN)
# use is.nan() to detect NULL variables
is.nan(0/0)
```

Note that `is.finite()` checks a numeric vector for the occurrence of any non-numerical or special values.

```
is.finite(c(1, NA, 2, Inf, 3, -Inf, 4, NULL, 5, NaN, 6))

##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE
```

18.3 From raw to technically correct data

18.3.1 Technically correct data

Limiting ourselves to “rectangular” data sets read from a text-based format, **technically correct** data in R

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

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.

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.

18.3.2 Reading text data into an 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.

Reading text

`read.table()` and similar functions below will read a text file and return a `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. The other read-functions below 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.

Additional optional arguments include:

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.

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. The following demonstrates this on the following text file.

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

Read the file with defaults, then specifying necessary options.

```
#### Example: unnamed person text
fn.data <- "http://statacumen.com/teach/ADA2/notes/ADA2_notes_Ch18_unnamed.txt"
# first line is erroneously interpreted as column names
person <- read.csv(fn.data)
person

##   X21 X6.0
## 1  42  5.9
## 2  18 5.7*
## 3  21 <NA>

# instead, use header = FALSE and specify the column names
person <- read.csv(file = fn.data
                 , 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>
```

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 MiB) 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(fn.data
       , header=FALSE
       , colClasses=c("numeric", "numeric")
        )

## Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, :  scan()
expected 'a real', got '5.7*'
# no data.frame output because of error
```

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` Next, one of the `as.`-functions can be applied to convert to the desired type, as shown below.

```
person <- read.csv(file = fn.data
                 , header = FALSE
                 , col.names = c("age", "height")
                 , stringsAsFactors = FALSE)
person

##   age height
## 1  21    6.0
## 2  42    5.9
## 3  18   5.7*
## 4  21   <NA>

person$height <- as.numeric(person$height)

## Warning:  NAs introduced by coercion
person

##   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.

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. We use the following data as an example.

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

And this is the table we want.

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. We want a general strategy so that if we had a file with 10,000 records we could process them all. The table suggests one strategy.

	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`

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.

```
#### Example: Dalton data
fn.data <- "http://statacumen.com/teach/ADA2/notes/ADA2_notes_Ch18_dalton.txt"
dalton.txt <- readLines(fn.data)
dalton.txt

## [1] "%% Data on the Dalton Brothers" "Gratt ,1861,1892"
## [3] "Bob,1892"                       "1871,Emmet ,1937"
## [5] "% Names, birth and death dates"

str(dalton.txt)

##  chr [1:5] "%% Data on the Dalton Brothers" "Gratt ,1861,1892" ...
```

The variable `dalton.txt` has 5 character 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. Regular expressions[3], though challenging to learn, can be used to specify what you're searching for. I usually search for an example and modify it to meet my needs.

```
# detect lines starting (^) with a percentage sign (%)
ind.nodata <- grepl("^%", dalton.txt)
ind.nodata

## [1]  TRUE FALSE FALSE FALSE  TRUE

# and throw them out
!ind.nodata

## [1] FALSE  TRUE  TRUE  TRUE FALSE

dalton.dat <- dalton.txt[!ind.nodata]
dalton.dat

## [1] "Gratt ,1861,1892" "Bob,1892"         "1871,Emmet ,1937"
```

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

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 char-

[3] `http://en.wikipedia.org/wiki/Regular_expression`

acter vectors.

```
# remove whitespace by substituting nothing where spaces appear
dalton.dat2 <- gsub(" ", "", dalton.dat)
# split strings by comma
dalton.fieldList <- strsplit(dalton.dat2, split = ",")
dalton.fieldList

## [[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, and the meaning of a 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 (a) every row has the same number of fields and (b) the fields are in the right order. In `read.table()`, lines that contain fewer 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.

The function below 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 `dalton.fieldList`.

```
# function to correct column order for Dalton data
f.assignFields <- function(x) {
  # create a blank character vector of length 3
  out <- character(3)
  # get name and put into first position
  ind.alpha <- grepl("[[:alpha:]]", x)
  out[1] <- x[ind.alpha]
  # get birth date (if any) and put into second position
  ind.num.birth <- which(as.numeric(x) < 1890)
```

```
    # if there are more than 0 years <1890,
    #   then return that value to second position,
    #   else return NA to second position
  out[2] <- ifelse(length(ind.num.birth) > 0, x[ind.num.birth], NA)
  # get death date (if any) and put into third position (same strategy as birth)
  ind.num.death <- which(as.numeric(x) > 1890)
  out[3] <- ifelse(length(ind.num.death) > 0, x[ind.num.death], NA)
  out
}
```

The function `lapply()` will apply the function `f.assignFields()` to each list element in `dalton.fieldList`.

```
dalton.standardFields <- lapply(dalton.fieldList, f.assignFields)

## Warning in which(as.numeric(x) < 1890):  NAs introduced by coercion
## Warning in which(as.numeric(x) > 1890):  NAs introduced by coercion
## Warning in which(as.numeric(x) < 1890):  NAs introduced by coercion
## Warning in which(as.numeric(x) > 1890):  NAs introduced by coercion
## Warning in which(as.numeric(x) < 1890):  NAs introduced by coercion
## Warning in which(as.numeric(x) > 1890):  NAs introduced by coercion

dalton.standardFields

## [[1]]
## [1] "Gratt" "1861"  "1892"
##
## [[2]]
## [1] "Bob"   NA      "1892"
##
## [[3]]
## [1] "Emmet" "1871"  "1937"
```

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, `f.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.

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`.

```
# unlist() returns each value in a list in a single object
unlist(dalton.standardFields)

## [1] "Gratt" "1861"  "1892"  "Bob"   NA      "1892"  "Emmet" "1871"
## [9] "1937"

# there are three list elements in dalton.standardFields
length(dalton.standardFields)

## [1] 3

# fill a matrix will the character values
dalton.mat <- matrix(unlist(dalton.standardFields)
                   , nrow = length(dalton.standardFields)
                   , byrow = TRUE
                    )
dalton.mat

##      [,1]    [,2]   [,3]  
## [1,] "Gratt" "1861" "1892"
## [2,] "Bob"   NA     "1892"
## [3,] "Emmet" "1871" "1937"

# name the columns
colnames(dalton.mat) <- c("name", "birth", "death")
dalton.mat

##      name    birth  death 
## [1,] "Gratt" "1861" "1892"
## [2,] "Bob"   NA     "1892"
## [3,] "Emmet" "1871" "1937"

# convert to a data.frame but don't turn character variables into factors
dalton.df <- as.data.frame(dalton.mat, stringsAsFactors=FALSE)
str(dalton.df)

## 'data.frame': 3 obs. of  3 variables:
##  $ name : chr  "Gratt" "Bob" "Emmet"
##  $ birth: chr  "1861" NA "1871"
##  $ death: chr  "1892" "1892" "1937"

dalton.df

##    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 and type conversion are discussed later. In this example we can suffice with the following statements.

```
dalton.df$birth <- as.numeric(dalton.df$birth)
dalton.df$death <- as.numeric(dalton.df$death)
str(dalton.df)

## 'data.frame': 3 obs. of  3 variables:
##  $ name : chr  "Gratt" "Bob" "Emmet"
##  $ birth: num  1861 NA 1871
##  $ death: num  1892 1892 1937

dalton.df

##    name birth death
## 1 Gratt  1861  1892
## 2   Bob    NA  1892
## 3 Emmet  1871  1937
```

18.4 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.integer
as.character
as.logical
as.factor
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.

18.4.1 Introduction to R's typing system

Everything in R is an object. 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"
# all columns in a data.frame
sapply(dalton.df, class)
##        name       birth       death
## "character"   "numeric"   "numeric"
```

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 (the homework includes an example of the subtleties).

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.

18.4.2 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"))
f
```

```
## [1] a b a a c
## Levels: a b c
levels(f)
## [1] "a" "b" "c"
as.numeric(f)
## [1] 1 2 1 1 3
```

You may need to create a translation table 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)
gender
## [1] 2 1 1 2 0 1 1
# recoding table, stored in a simple vector
recode <- c(male = 1, female = 2)
recode
##   male female
##      1      2
gender <- factor(gender, levels = recode, labels = names(recode))
gender
## [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")
gender
## [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)
gender
## [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
```

18.4.3 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] "2020-03-04 14:14:11 MST"
```

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 contains a number of functions facilitating the conversion of text to `POSIXct` dates. As an example, consider the following code.

```r
library(lubridate)
dates <- c("15/02/2013"
         , "15 Feb 13"
         , "It happened on 15 02 '13")
dmy(dates)

## [1] "2013-02-15" "2013-02-15" "2013-02-15"
```

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 (Fevrier).

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

```r
dmy()
dym()
mdy()
myd()
ydm()
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. Recently in R, 00-69 was interpreted as 2000-2069 and 70-99 as 1970-1999; this behaviour is according to the 2008 POSIX standard, but one should expect that this interpretation changes over time. Currently all are now 2000-2099.

```r
dmy("01 01 68")

## [1] "2068-01-01"

dmy("01 01 69")

## [1] "1969-01-01"

dmy("01 01 90")

## [1] "1990-01-01"
```

```
dmy("01 01 00")
## [1] "2000-01-01"
```

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 below. The complete list can be found by typing `?strptime` in the R console. These are the 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`

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.

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 MDT" "2008-07-16 MDT" NA
## [4] 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. 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"
```

18.5 Character-type 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
M
male
Female
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.

18.5.1 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 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"
```

18.5.2 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, from the previous example, 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 (and thereby search for the “^” character).

Search patterns using meta-characters are called *regular expressions*. Regular expressions[4] offer powerful and flexible ways to search (and alter) text. 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. 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. 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")
# calculate pairwise distances between the gender strings and codes strings
dist.g.c <- adist(gender, codes)
# add column and row names
colnames(dist.g.c) <- codes
rownames(dist.g.c) <- gender
dist.g.c

##        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 dist.g.c. This can be

[4] `http://en.wikipedia.org/wiki/Regular_expression`

done as follows.

```
ind.min <- apply(dist.g.c, 1, which.min)
data.frame(rawtext = gender, coded = codes[ind.min])

##   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 `dist.g.c`. 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[5] 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 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 earlier gender and code example.

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

[5] Informally, the Levenshtein distance between two words is the minimum number of single-character edits (i.e., insertions, deletions, or substitutions) required to change one word into the other: `https://en.wikipedia.org/wiki/Levenshtein_distance`.

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

# store results in a data.frame
data.frame(rawtext = gender, code = codes[ind])

##   rawtext   code
## 1       M   male
## 2   male    male
## 3  Female female
## 4    fem. female
```

18.6 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.

- **Detection** *of an inconsistency.* That is, one establishes which constraints are violated. For example, an age variable is constrained to non-negative values.

- **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.

- **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.

First, 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**. Next, we describe 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, we give an overview of some commonly used imputation techniques that are available in R.

18.6.1 Detection and localization of errors

This section details a number of techniques to detect univariate and multivariate constraint violations.

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 previously, 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 earlier.

```
print(person)

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

complete.cases(person)

## [1]  TRUE  TRUE FALSE 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)
persons_complete

##   age height
## 1  21    6.0
## 2  42    5.9

na.action(persons_complete)

## 3 4
## 3 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.

Special values

As explained previously, 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. 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`.

```
f.is.special <- function(x) {
  if (is.numeric(x)) {
    return(!is.finite(x))
  } else {
    return(is.na(x))
  }
}
person

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

sapply(person, f.is.special)

##        age height
## [1,] FALSE  FALSE
## [2,] FALSE  FALSE
## [3,] FALSE   TRUE
## [4,] FALSE   TRUE
```

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

Outliers

There is a vast body of literature on outlier detection, and several definitions of **outlier** exist. A general definition by Barnett and Lewis 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), this definition is sufficient for the current chapter. Below we mention a few fairly common graphical and computational techniques for outlier detection in univariate numerical data. In a previous chapter, we've discussed using PCA as a graphical technique to help detect multivariate outliers.

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 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)
```

```
## $stats
## [1]  1.0  3.5  6.5  9.5 10.0
##
## $n
## [1] 12
##
## $conf
## [1] 3.76336 9.23664
##
## $out
## [1] 20 30
```

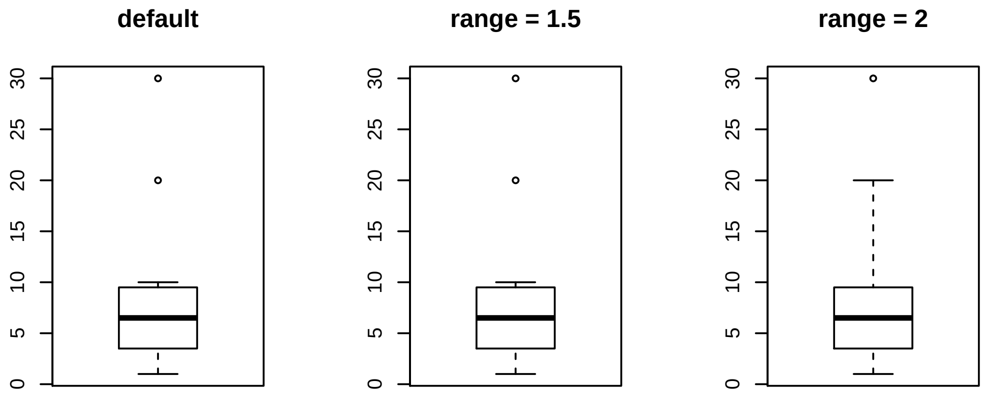
Here, `20` and `30` are detected as outliers since they are above the upper whisker of the observations in `x`. The factor 1.5 used to compute the whisker is to an extent arbitrary and it can be altered by setting the coef option of `boxplot.stats()`. A higher coefficient means a higher outlier detection 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.

```
op <- par(no.readonly = TRUE)          # save plot settings
par(mfrow=c(1,3))
boxplot(x, main="default")
boxplot(x, range = 1.5, main="range = 1.5")
boxplot(x, range = 2,   main="range = 2")
par(op)                                # restore plot settings
```



The box-and-whisker method fails when data distribution is skewed, as in an exponential or log-normal distribution. 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 method for outlier detection with positive observations is by Hiridoglou and Berthelot. In this method, an observation is an outlier when

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

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.

```
#### Example: Hiridoglou and Berthelot outlier detection function
f.hb.outlier <- function(x,r) {
  x <- x[is.finite(x)]
  stopifnot(length(x) > 0 , all(x>0)) # if empty vector or non-positive values, quit
  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
}
f.hb.outlier(x, r = 4)

##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12]  TRUE
```

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

18.6.2 Edit rules for detecting 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 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()`.

```
#### Example: people data for cleaning
fn.data <- "http://statacumen.com/teach/ADA2/notes/ADA2_notes_Ch18_people.txt"
people <- read.csv(fn.data)
people
```

```
##   age agegroup height  status yearsmarried
## 1  21    adult    6.0  single           -1
## 2   2    child    3.0 married            0
## 3  18    adult    5.7 married           20
## 4 221  elderly    5.0 widowed            2
## 5  34    child   -7.0 married            3
```

```
library(editrules)
E <- editset(c("age >=0", "age <= 150"))
E
```

```
##
## 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 221 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 (note, you can't include braces in your `if()` statement).

```
# 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"
```

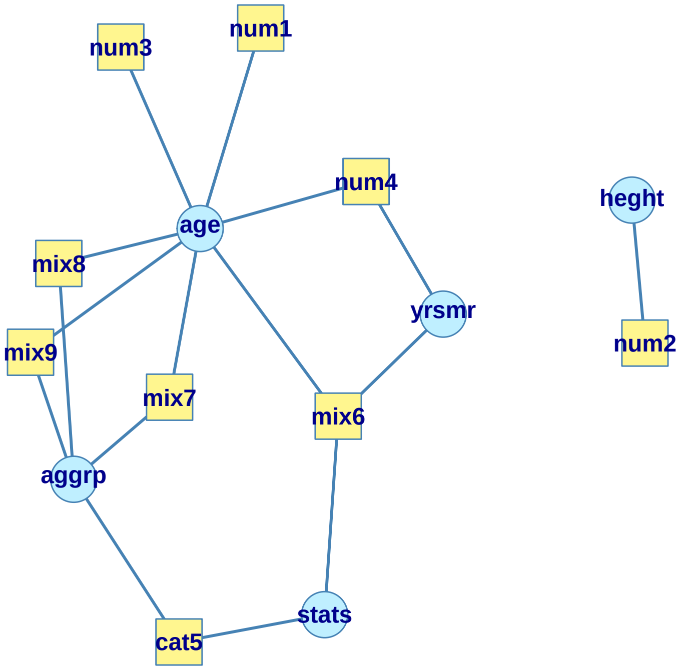
There are rules pertaining to purely numerical, purely categorical and rules pertaining to both data types. Moreover, there are univariate as well as multivariate rules. Comments are written behind the usual # character. The rule set can be read as follows.

```
fn.data <- "http://statacumen.com/teach/ADA2/notes/ADA2_notes_Ch18_edits.txt"
E <- editfile(fn.data)
E

##
## Data model:
## dat6 : agegroup %in% c('adult', 'child', 'elderly')
## dat7 : status %in% c('married', 'single', 'widowed')
##
## Edit set:
## num1 : 0 <= age
## num2 : 0 < height
## num3 : age <= 150
## num4 : yearsmarried < age
## cat5 : if( agegroup == 'child' ) status != 'married'
## mix6 : if( age < yearsmarried + 17 ) !( status %in% c('married', 'widowed') )
## mix7 : if( age < 18 ) !( agegroup %in% c('adult', 'elderly') )
## mix8 : if( 18 <= age & age < 65 ) !( agegroup %in% c('child', 'elderly') )
## mix9 : if( 65 <= age ) !( agegroup %in% c('adult', 'child') )
```

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. Below the graph plot shows the interconnection of restrictions. Blue circles represent variables and yellow boxes represent restrictions. The lines indicate which restrictions pertain to what variables.

```
op <- par(no.readonly = TRUE)          # save plot settings
par(mfrow=c(1,1), mar = c(0,0,0,0))
plot(E)
par(op)                                # restore plot settings
```

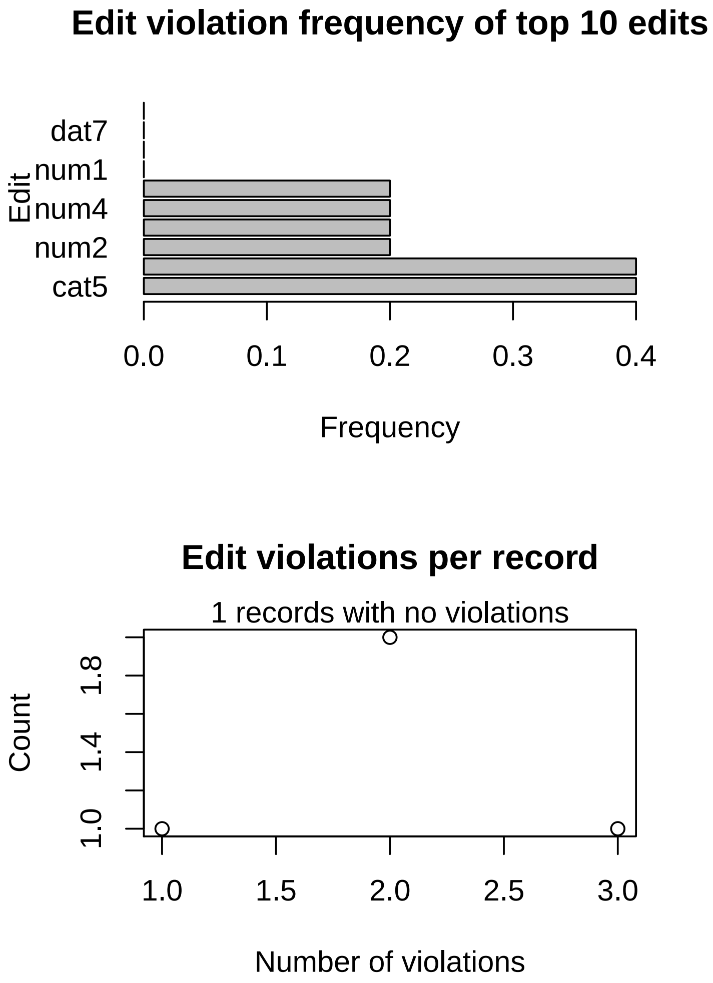


As the number of rules grows, looking at the full array produced by `violatedEdits(` becomes cumbersome. For this reason, `editrules` offers methods to summarize or visualize the result.

```
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.

Error localization

The interconnectivity of edits is what makes error localization difficult. For example, the graph above shows that a record violating edit `num4` may contain an error in `age` and/or `yrsmr` (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`.

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, 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 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  dat6  dat7 cat5  mix6  mix7  mix8
##      2 FALSE FALSE FALSE FALSE FALSE FALSE TRUE  TRUE FALSE FALSE
##      5 FALSE  TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE  TRUE
##       edit
## record  mix9
##      2 FALSE
##      5 FALSE
```

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

```
le <- localizeErrors(E, people[id, ], method = "mip")
le$adapt

##     age agegroup height status yearsmarried
## 2 FALSE    FALSE  FALSE   TRUE        FALSE
## 5 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
## NULL
```

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.

18.6.3 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.

For the purposes of ADA2, we will manually correct errors, either by replacing values or by excluding observations.

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.

```
#### Example: marx brothers data
marx <- read.table(text = "
name    height unit
Groucho 170.00 cm
Zeppo     1.74 m
Chico    70.00 inch
Gummo   168.00 cm
Harpo     5.91 ft
", header=TRUE, stringsAsFactors = FALSE)
marx
```

```
##      name height unit
## 1 Groucho 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 indexing techniques, which would look something like this.

```
marx_m <- marx
ind <- (marx$unit == "cm")                     # indexes for cm
marx_m[ind, "height"] <- marx$height[ind] / 100
marx_m[ind, "unit"]   <- "m"
ind <- (marx$unit == "inch")                   # indexes for inch
marx_m[ind, "height"] <- marx$height[ind] / 39.37
marx_m[ind, "unit"]   <- "m"
ind <- (marx$unit == "ft")                     # indexes for ft
marx_m[ind, "height"] <- marx$height[ind] / 3.28
marx_m[ind, "unit"]   <- "m"
marx_m
```

```
##      name   height unit
## 1 Groucho 1.700000    m
## 2   Zeppo 1.740000    m
## 3   Chico 1.778004    m
## 4   Gummo 1.680000    m
## 5   Harpo 1.801829    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 off 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
  unit <- "m" # set all units to meter
}
# convert inches
if (unit == "inch" ){
  height <- height / 39.37
  unit <- "m" # set all units to meter
}
# convert feet
if (unit == "ft" ){
  height <- height / 3.28
  unit <- "m" # set all units to meter
}
```

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

```
#### Conversions for marx bros data
library(deducorrect)
fn.data <- "http://statacumen.com/teach/ADA2/notes/ADA2_notes_Ch18_conversions.txt"
# read the conversion rules.
R <- correctionRules(fn.data)
R

## Object of class 'correctionRules'
## ##  1-------
##   if (unit == "cm") {
##       height <- height/100
##       unit <- "m"
##   }
## ##  2-------
##   if (unit == "inch") {
##       height <- height/39.37
##       unit <- "m"
##   }
## ##  3-------
##   if (unit == "ft") {
##       height <- height/3.28
##       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 Groucho 1.700000    m
## 2   Zeppo 1.740000    m
## 3   Chico 1.778004    m
## 4   Gummo 1.680000    m
## 5   Harpo 1.801829    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   1.77800355600711
## 4   3     unit inch m
## 5   4   height  168             1.68
## 6   4     unit  cm                m
## 7   5   height 5.91 1.80182926829268
## 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 unit <- "m" }
## 2     if (unit == "cm") { height <- height/100 unit <- "m" }
## 3 if (unit == "inch") { height <- height/39.37 unit <- "m" }
## 4 if (unit == "inch") { height <- height/39.37 unit <- "m" }
## 5     if (unit == "cm") { height <- height/100 unit <- "m" }
## 6     if (unit == "cm") { height <- height/100 unit <- "m" }
## 7    if (unit == "ft") { height <- height/3.28 unit <- "m" }
## 8    if (unit == "ft") { height <- height/3.28 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 in correctionRules.expression(expression(x <- mean(x))):  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.

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. 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.

[*I will complete this section if we need it for our Spring semester.*]

Deterministic imputation

In some cases a missing value can be determined because the observed values combined with their constraints force a unique solution.

[*I will complete this section if we need it for our Spring semester.*]

18.6.4 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 chapter to discuss all of them.

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 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.