Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

cleanR: Maid for Cleaning Datasets in R

Anne H. Petersen
Biostatistics
Department of Public Health
University of Copenhagen

Claus Thorn Ekstrøm
Biostatistics
Department of Public Health
University of Copenhagen

Abstract

Data cleaning and -validation are the first steps in any data analysis, as the validity of the conclusions from the analysis hinges on the quality of the input data. Mistakes in the data can arise for any number of reasons, including erroneous codings, malfunctioning measurement equipment, inconsistent data generation manuals and many more. Ideally, a human investigator should go through each variable in the dataset and look for potential errors — both in input values and codings — but that process can be very time-consuming, expensive and error-prone in itself.

We describe an R package which implements an extensive and customizeable suite of quality assessment tools that can be applied to a dataset in order to identify potential problems in its variables. The results can be presented in an auto-generated, non-technical, stand-alone overview document, intended to be perused by an investigator with an understanding of the variables in the data, but not necessarily knowledge of R. Thereby, **cleanR** aids the dialogue between data analysts and field experts, while also providing easy documentation of reproducible data cleaning steps and data quality control. Moreover, the **cleanR** solution changes the data cleaning process from the usual ad hoc approach to a systematic, well-documented endeavor. **cleanR** also provides a suite of more typical R tools for interactive data quality assessment and -cleaning, where the data inspections all live and die within the R console.

Keywords: data cleaning, quality control, R.

1. Introduction

Though data cleaning might be regarded as a somewhat tedious activity, adequate data cleaning is crucial in any data analysis. With ever-growing dataset sizes and complexities, statisticians and data analysts find themselves spending a large portion of their time on data cleaning and on data wrangling. While a computer should never make unsupervised decisions

on what should be done to potential errors in the dataset, it can still be an extremely useful tool in the data cleaning process. Some errors can be tracked down and flagged by a computer without further ado, while other types of errors need an analytic context in order to shine through. But even in the latter case, well-designed software can aid the process tremendously by giving the human investigator the information needed for identifying issues.

Online tools such as OpenRefine (`http://openrefine.org/`) and R-packages such as **plyr**, and **data.table** have made data wrangling a lot easier, but only a handful of packages such as **editrules**, **validate**, **DataCombine**, and **janitor** attempt to implement systematic, reproducible data cleaning. These packages use different approaches for data cleaning: **editrules** and **validate** provide frameworks for setting up and checking constraints on the variables, while **DataCombine** and **janitor** both provide a few functions for identifying problems (e.g, duplicates, dates coded as numbers, etc.) in data.

While these tools attempt to alleviate the ubiquitous ad hoc approach to data cleaning, they are primarily intended for the data savvy users and less so for the general researcher with a knowledge about a specific field and the context of the available data. The **cleanR** package tries to address this by providing a framework that both allows for extendable, systematic, reproducible data cleaning, and summarizing findings for researchers from other fields such that they can act as human experts when tracking down potential errors.

Data cleaning is a time consuming endeavor, as it inherently requires human interaction since every dataset is different and the variables in the dataset can only be understood in the proper context of their origin. This often requires a collaborative effort between an expert in the field and a statistician or data scientist, which may be why the process of proper data cleaning is not always undertaken. In many situations, these errors are discovered in the process of the data analysis (e.g., a categorical variable with numeric labels for each category may be wrongly classified as a quantitative variable or a variable where all values have erroneously been coded to the same value), but in other cases a human with knowledge about the data context area is needed to identify possible mistakes in the data (e.g., if there are 4 categories for a variable that should only have 3).

The **cleanR** approach to data cleaning and -quality assessment is governed by two fundamental paradigms. First of all, there is no need for data cleaning to be an ad hoc procedure. Often, we have a very clear idea of what flags are raisable in a given dataset before we look at it, as we were the ones to produce it in the first place. This means that data cleaning can easily be a well-documented, well-specified procedure. In order to aid this paradigm, **cleanR** provides easy-to-use, automated tools for data quality assessment in R on which data cleaning decisions can be made. This quality assessment is presented in an auto-generated overview document, readable by data analysts and field experts alike, thereby also contributing to an inter-field dialogue about the data at hand. Oftentimes, e.g. distinguishing between faulty codings of a numeric value and unusual, but correct, values requires problem-specific expertise that might not be held by the data analyst. Hopefully, having easy access to data descriptions through **cleanR** will help this necessary knowledge sharing.

While **cleanR**s primary raison d'être is auto-generating data quality assessment overview documents, we still wish to emphasize that it is *not* a tool for unsupervised data cleaning. This qualifies as the second paradigm of **cleanR**: Data cleaning decisions should always be made by humans. Therefore, **cleanR** does not supply any tools for "fixing" errors in the data. However, we do provide interactive functions that can be used to identify potentially

erroneous entries in a dataset and that can make it easier to solve data issues, one variable at a time.

This manuscript is structured as follows: First, in Section 2, we introduce the main representative of the first paradigm, namely the `clean()` function, which generates data cleaning overview documents. In the **cleanR** package, we have provided a number of default cleaning steps that cover the data cleaning challenges, we find to be most common. Next, in Section 3, we present the interactive mode of **cleanR**, as motivated by the second paradigm above. But, as any data analyst knows, every dataset is different, and some datasets might include problems that cannot be detected by our data checking functions. In Section 4, we turn to the question of how **cleanR** extensions can be made, such that they are integrable with the `clean()` function and with the other tools available in **cleanR**. At last, in Section 5, we discuss a number of examples of specific data cleaning challenges and how **cleanR** can be used to solve them.

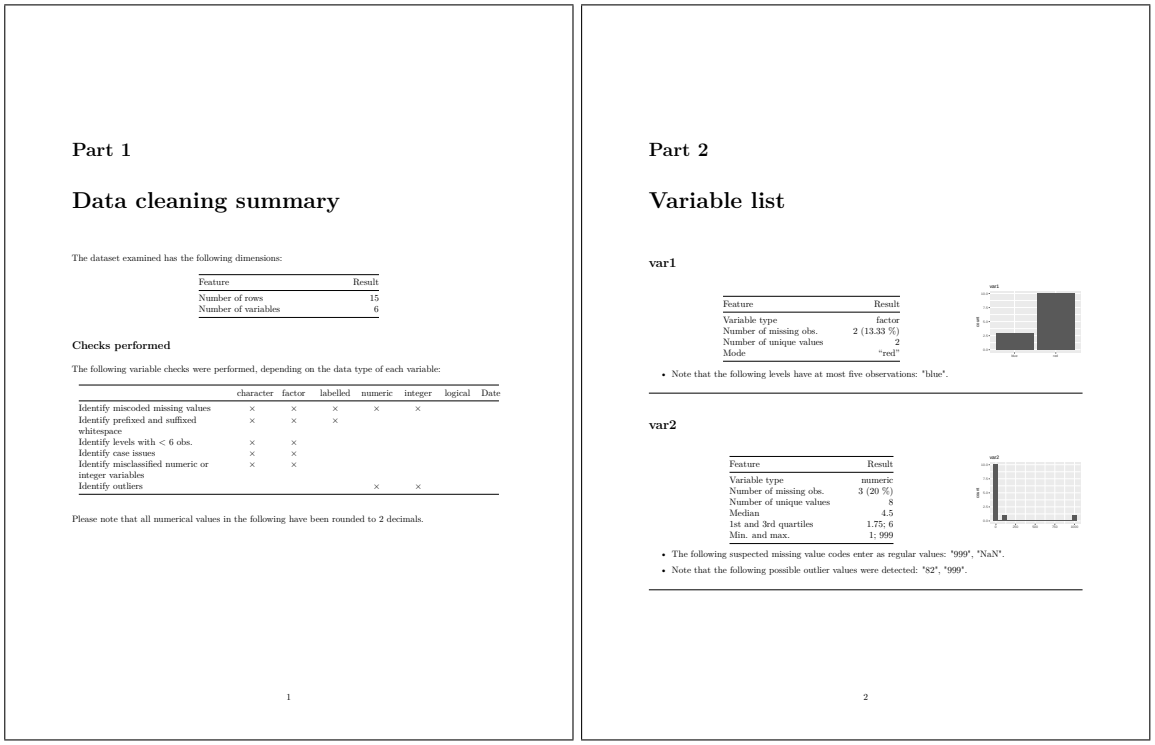
2. Creating a data cleaning overview

The `clean()` function is the primary workhorse of **cleanR** and this is the only function needed if a standard battery of tests are used to generate data cleaning summaries. The data cleaning summary itself is an overview document, intended for reading by humans, in either pdf or html format. Appendix A provides an example of a data cleaning output document, produced by calling `clean()` on the dataset `toyData` available in **cleanR**. The first two pages of this data cleaning output are shown in Figure 2. `toyData` is a very small (15 rows of 6 variables), artificial dataset which was created to illustrate the main capabilities of **cleanR**. The following commands load the dataset and produce the cleaning output:

```
> library(cleanR)
> data(toyData)
> toydata
```

```
   var1 var2  var3        var4 var5       var6
1   red    1     a -0.65959383    1 Irrelevant
2   red    1     a  0.08671649    2 Irrelevant
3   red    1     a -0.10951326    3 Irrelevant
4   red    2     a  0.08630221    4 Irrelevant
5   red    2     a -1.84311184    5 Irrelevant
6   red    6     b  0.92210680    6 Irrelevant
7   red    6     b  1.01921086    7 Irrelevant
8   red    6     b -0.92428326    8 Irrelevant
9   red  999     c -0.65340163    9 Irrelevant
10  red   NA     c  0.21133941   10 Irrelevant
11 blue    4     c  0.91783009   11 Irrelevant
12 blue   82     .  0.10313983   12 Irrelevant
13 blue   NA        0.16954218   13 Irrelevant
14 <NA>  NaN other  0.41967230   14 Irrelevant
15 <NA>    5 OTHER  0.77143836   15 Irrelevant
```

```
> clean(toyData)
```



Part 1

Data cleaning summary

The dataset examined has the following dimensions:

Feature	Result
Number of rows	15
Number of variables	6

Checks performed

The following variable checks were performed, depending on the data type of each variable:

	character	factor	labelled	numeric	integer	logical	Date
Identify miscoded missing values	×	×	×	×	×		
Identify prefixed and suffixed whitespace	×	×	×				
Identify levels with < 6 obs.	×	×					
Identify case issues	×	×					
Identify misclassified numeric or integer variables	×	×					
Identify outliers				×	×		

Please note that all numerical values in the following have been rounded to 2 decimals.

1

Part 2

Variable list

var1

Feature	Result
Variable type	factor
Number of missing obs.	2 (13.33 %)
Number of unique values	2
Mode	"red"

- Note that the following levels have at most five observations: "blue".

var2

Feature	Result
Variable type	numeric
Number of missing obs.	3 (20 %)
Number of unique values	8
Median	4.5
1st and 3rd quartiles	1.75; 6
Min. and max.	1; 999

- The following suspected missing value codes enter as regular values: "999", "NaN".
- Note that the following possible outlier values were detected: "82", "999".

2

Figure 1: Example output from running `clean()` on the `toyData` dataset. First a summary of the full dataset is given and then type-dependent information on each variable is given in a table and a graph. Larger versions of the pages can be seen in Appendix A.

By default, an R markdown file and a rendered pdf overview document is produced, saved to the disc (in the working directory) and opened for immediate inspection. Turning to Figure 2, we see that such a data cleaning output document consists of two parts. First, an overview of what was done is presented under the title *Data cleaning summary*. Secondly, each variable in the dataset is presented in turn using (up to) three tools in the *Variable list*: A table summarizing key features of the variable, a figure visualizing its distribution and a list of flagged issues, if any. For instance, in the `numeric`-type variable `var2` from `toyData`, `clean()` has identified two values that are suspected to be miscoded missing values (`999` and `NaN`), while two values were also flagged as potential outliers that should be investigated more carefully.

Though the `clean()` function is very easy to use, it should not be mistaken to be inflexible. A large number of function arguments allows for the cleaning overview document to be molded according to the user's needs. The most commonly used arguments are summarized in Table 1 and they are grouped according to the part of the cleaning process they influence. In order to understand this distinction, a glimpse of the inner structure of `clean()` is shown in Figure 2. Below, we present a few examples on how to use the arguments from Table 1 to influence the output of a `clean()` call.

Argument	Description	Default value
Control input variables and summary		
`useVar`	What variables should be used?	`NULL` (corresponding to all variables)
`ordering`	Ordering of the variables in the data summary (as is or alphabetical)	`"asIs"`
`onlyProblematic`	Should only variables flagged as problematic be included in the *Variable list*?	`FALSE`
`listChecks`	Should an overview of what checks were performed be listed in the *Data cleaning summary*?	`TRUE`
`preChecks`	What check functions should be called to determine whether a variable is suitable for summarization, visualization and checking?	`c("isKey", "isEmpty")` Control summarize, visualize, and check steps
`mode`	What steps should be performed for each variable (out of the three possibilities *summarize*, *visualize*, *check*)?	`c("summarize", "visualize", "check")` (corresponding to all three)
`smartNum`	Should numerical values with only a few unique levels be flagged and treated as a factor variable?	`TRUE`
`maxProbVals`	Maximum number of problematic values to print, if any are found in data checks	`Inf`
`maxDecimals`	Maximum number of decimals to print for numeric values in the variable list	`2`
`twoCol`	Should the summary table and visualizations be placed side-by-side (in two columns)?	`TRUE`
Control output and post-processing		
`output`	Type of output file to be produced (html, or pdf)	`"pdf"`
`render`	Should the output file be rendered from markdown?	`TRUE`
`openResult`	If a pdf/html file is rendered, should it automatically open afterwards, and if not, should the `rmarkdown` file be opened?	`TRUE`
`replace`	Overwrite an existing files with the same name?	`FALSE`

Table 1: A selection of commonly used arguments to `clean()` separated into the parts they control.

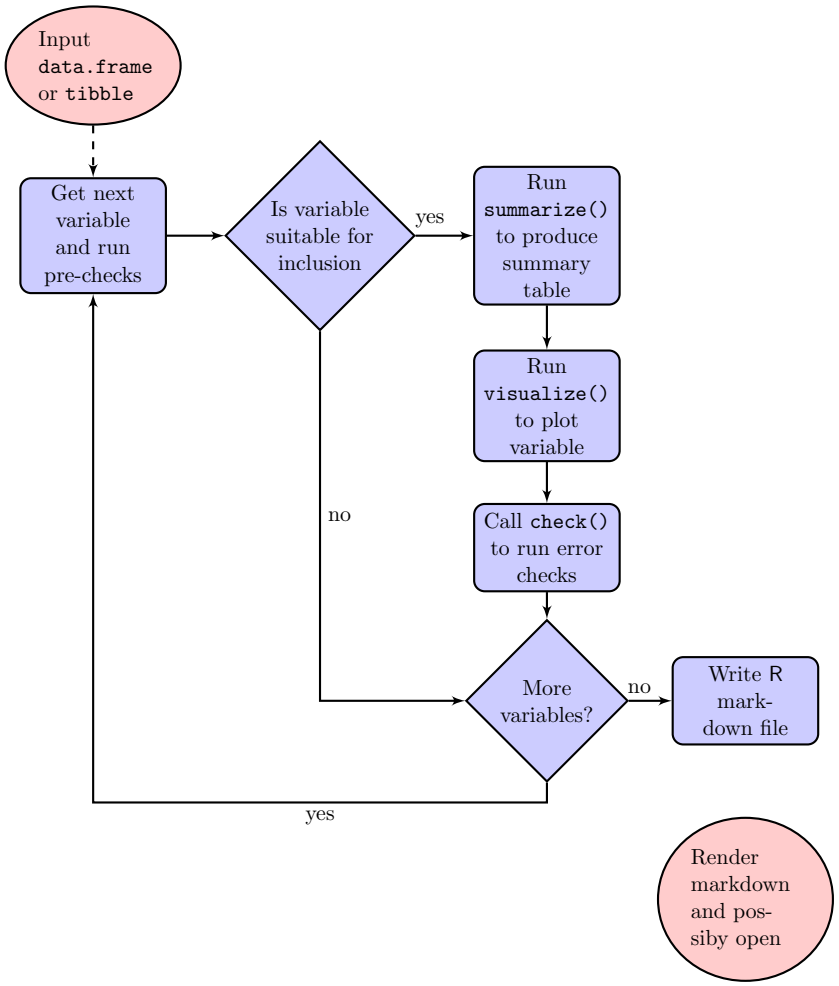


Figure 2: Schematic illustration of the stages undertaken when running `clean()`. Each variable is checked for eligibility before running `summarize()`, `visualize()`, and `check()`, and the resulting R markdown file may be rendered and opened.

2.1. Rolling out the arguments

We begin with an example that is intended as an illustration of how `clean()` might be used in the very first stages of data cleaning, where we are still uncertain about exactly what is needed and how much times should be allocated to data cleaning. At this stage, what is really needed, if a very rough idea of the extends of errors in the dataset. In this scenario, we might wish to obtain a summary document in html format that only contains the variables with potential problems, and with a limit of maximum 10 printed potential errors for each variable we can write (output not shown). Also, we can add the argument `replace=TRUE` in order to force `clean()` to overwrite any existing files produced by `clean()`, as we might run this function a few times in a row before deciding exactly which cleaning overview we want. Using the `toyData` dataset as a guinea pig, we type:

```
> clean(toyData, output = "html", onlyProblematic = TRUE, maxProbVals = 10,
```

```
        replace = TRUE)
```

The final rendering of the generated markdown file is controlled by the `render` and `openResult` arguments, which both default to `TRUE` Bliver det ved med at være sandt?. `render` determines if the R markdown file produced should be rendered using the **rmarkdown** package and `openResult` decides whether the output html or pdf file should be opened. The following command produces an R markdown file containing the data cleaning summary information, but without rendering nor opening the markdown file:

```
> clean(toyData, output="html", render=FALSE, openResult=FALSE)
```

We will now move on to discuss how not only the structure of the cleaning summary is manipulated, but also its very contents. This is done through the summarize-visualize-check (SVC) step, as illustrated in Figure 2. **cleanR** uses three different types of functions for performing these steps, namely `summaryFunctions`, `visualFunctions` and `checkFunctions`. By default, `clean()` runs selected summary, visualization and check functions on each variable in the dataset, and the exact choice of these functions depends on the classes of the variables. For instance, though detection of outlier values might be interesting for numerical variables, it holds little meaning for factor variables, and therefore, numerical and factor variables need different checks. Table 2 lists all available SVC functions, but we can also use the `allSummaryFunctions()`, `allVisualFunctions()`, and `allCheckFunctions()` functions in **cleanR** to print overview lists in R. For example, the implemented `summaryFunctions` are:

```
> allSummaryFunctions()
```

```
-------------------------------------------------------------------------
name           description                    classes
-------------- ------------------------------ ----------------------------
centralValue   Compute median or mode         character, Date, factor,
                                              integer, labelled, logical,
                                              numeric

countMissing   Compute ratio of missing       character, Date, factor,
               observations                   integer, labelled, logical,
                                              numeric

minMax         Find minimum and maximum       Date, integer, numeric
               values

quartiles      Compute 1st and 3rd quartiles  Date, integer, numeric

uniqueValues   Count number of unique values  character, Date, factor,
                                              integer, labelled, logical,
                                              numeric

variableType   Data class of variable         character, Date, factor,
```

	Description	Variable classes						
		C	F	I	L	B	N	D
`summaryFunctions`								
`centralValue`	Compute median or mode	×	×	×	×	×	×	×
`countMissing`	Compute ratio of missing observations	×	×	×	×	×	×	×
`minMax`	Find minimum and maximum values			×			×	×
`quartiles`	Compute 1st and 3rd quartiles			×			×	×
`uniqueValue`	Count number of unique values	×	×	×	×	×	×	×
`variableType`	Data class of variable	×	×	×	×	×	×	×
`visualFunctions`								
`basicVisual`	Histograms and barplots using base R graphics	×	×	×	×	×	×	×
`standardVisual`	Histograms and barplots using ggplot2	×	×	×	×	×	×	×
`checkFunctions`								
`identifyCaseIssues`	Identify case issues	×	×					
`identifyLoners`	Identify levels with < 6 obs.	×	×					
`identifyMissing`	Identify miscoded missing values	×	×	×	×	×	×	
`identifyNums`	Identify misclassified numeric or integer variables	×	×					
`identifyOutliers`	Identify outliers			×		×	×	
`identifyOutliersTBStyle`	Identify outliers (Turkish Boxplot style)			×		×	×	
`identifyWhitespace`	Identify prefixed and suffixed whitespace	×	×		×			
`isCPR`	Identify Danish CPR numbers	×	×	×	×	×	×	×
`isEmpty`	Check if the variable contains only a single value	×	×	×	×	×	×	×
`isKey`	Check if the variable is a key	×	×	×	×	×	×	×

Table 2: List of all summary functions (used in the summary table for each variable in the output), visual functions (used for visualization of each variable), and check functions (used for data checks for each variable) currently implemented in **cleanR**. The variable classes C, F, I, L, B, N, and D, refer to `character`, `factor`, `integer`, `labelled`, `logical` (boolean), `numeric`, and `Date`, respectively.

```
                                                integer, labelled, logical,
                                                numeric
------------------------------------------------------------------------------
```

Thus we can see, for example, that for `numeric`, `integer`, or `Date` variables, **cleanR** provides functions for adding summary information about the minimum and maximum values, while all seven variable classes dealt with in **cleanR** have functions for central tendency summaries (i.e., mode or median).

We can control what summaries and checks are applied for each variable type through the `XXXSummaries` and `XXXChecks` arguments, where `XXX` represents a variable type, e.g., `factorChecks` for factors, `numericChecks` for numeric variables, etc. These arguments accept a vector of summary- or check function names that should be run for a particular variable type. The default values for each of these arguments can be obtained through the `defaultXXXChecks()` and `defaultXXXSummaries()` functions, where `XXX` again refers to the variable type.

For example, the default summaries being used for a factor variable is

```
> defaultFactorSummaries()

[1] "variableType" "countMissing" "uniqueValues" "centralValue"
```

We can change the summaries (and similarly the checks) by setting the corresponding arguments when calling `clean()`. For example, to get only the variable type and the central tendency listed in the summary table of each variable we write

```
> clean(toyData, characterSummaries = c("variableType", "centralValue"),
        factorSummaries = c("variableType", "centralValue"),
        labelledSummaries = c("variableType", "centralValue"),
        numericSummaries = c("variableType", "centralValue"),
        integerSummaries = c("variableType", "centralValue"),
        logicalSummaries = c("variableType", "centralValue"),
        dateSummaries = c("variableType", "centralValue"))
```

In this particular case, where we specify the same set of summary functions for each variable type, we can use the simpler argument `allSummaries` which overrides the summary functions for all variable types. Thus, the same result as above could be obtained with

```
> clean(toyData, allSummaries = c("variableType", "centralValue"))
```

Similarly, the checks applied are set with the `XXXChecks` arguments. The default checks being applied to a factor is

```
> defaultFactorChecks()

[1] "identifyMissing"    "identifyWhitespace" "identifyLoners"
[4] "identifyCaseIssues" "identifyNums"
```

Now, if we only wanted to apply the identify whitespace function for factor variables, then we would need to set the `factorChecks` accordingly

```
> clean(toyData, factorChecks = c("identifyWhitespace"))
```

or we could remove checks for factors altogether by setting the corresponding argument to `NULL`, in which case factor variables will not be checked for any potential errors:

```
> clean(toyData, factorChecks = NULL)
```

As with `summaryFunction`s, a complete list of available `checkFunction`s is obtained by calling `allCheckFunctions()`. Note however, that `checkFunction`s have a usage beyond the `XXXChecks` arguments, namely in the *pre-check* stage. In this stage, it is determined whether or not each variable is suitable for the SVC-steps. The functions used in the pre-check stage should be `checkFunction`s that are applicable to all variable classes. The default pre-checks, the functions `isKey` and `isEmpty`, check whether a variable has unique values for all observations or only a single value for all observations, respectively. If one of these statements are true, the variable will not be subjected to the SVC steps. We can allow empty variables to move on to the SVC step by only checking for keys in the pre-check step:

```
> clean(toyData, preChecks = "isKey")
```

Note that the data visualizations in the cleaning summary are also controllable, though only a single function can be provided for all variable types. If for instance, we wish to change the visualizations from the default **ggplot2** style histograms and barplots to base R histograms and barplots, we type

```
> clean(toyData, allVisuals = "basicVisual")
```

In summary, and as indicated in Figure 2, there are two stages where `clean()` applies functions to each of the variables:

1. In the pre-check stage
2. As part of the summarize/visualize/check (SVC) steps

Each of these stages are controllable using appropriate function arguments in `clean()`, and above we have shown examples of how to tweak them to modify the data cleaning outputs. However, if for instance the dataset at hand requires new, additional checks, then more control is needed, and Section 4 explains how to modify and expand the possibilities by producing new summary-, visual- and check functions.

3. Using cleanR interactively

While overview documents are great for presenting and documenting the data cleaning checks, it may be useful to be able to work more interactively through the data cleaning process as well. Aside from the `clean()` function presented above, **cleanR** also provides more standard

R interactive tools, such as functions that print results to the console or returns the information as an object for later use. This section describes how to use the functions `check()`, `summarize()` and `visualize()` to work interactively with **cleanR**.

3.1. Data cleaning by hand: An example

Let's say we wish to look further into a certain variable from `toyData`, namely `var2`. The data cleaning summary found some issues in this variable, and we would like to recall what these issues were. This can be done using the `check()` command

```
> check(toyData$var2)

$identifyMissing
The following suspected missing value codes enter as regular values: 999, NaN.
$identifyOutliers
Note that the following possible outlier values were detected: 82, 999.
```

Note that the arguments specifying which checks to perform, as described in the previous section, are in fact passed to `check()`, and thus they can also be used here. For instance, if we only want to check for potentially miscoded missing values, we can use the relevant `XXXChecks` argument (e.g., `numericChecks`, `factorChecks`, etc. as described in Section 2) to provide a vector of the check functions that should be applied. Recall that Table 2 or an `allCheckFunctions()` call provide overviews of the available check functions. Moving forward, we limit the numeric checks to only identify miscoded missing values:

```
> check(toyData$var2, numericChecks = "identifyMissing")

$identifyMissing
The following suspected missing value codes enter as regular values: 999, NaN.
```

An equivalent way to call only a single, specific `checkFunction`, such as `identifyMissing`, is by using it directly on the variable, e.g.,

```
> identifyMissing(toyData$var2)

The following suspected missing value codes enter as regular values: 999, NaN.
```

The result of a `checkFunction` is an object of class `checkResult`. By using the structure function, `str()`, we can look further into its components:

```
> missVar2 <- identifyMissing(toyData$var2)
> str(missVar2)

List of 3
 $ problem      : logi TRUE
 $ message      : chr "The following suspected missing value codes enter as
                  regular values: \\\"999\\\", \\\"NaN\\\"."
 $ problemValues: num [1:2] 999 NaN
 - attr(*, "class")= chr "checkResult"
```

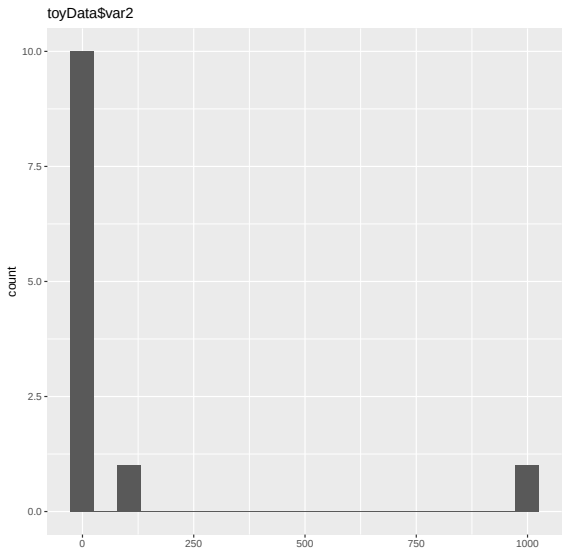


Figure 3: Output from running `visualize()` on the variable `var2` from the `toyData` dataset.

The most important thing to note here is that while the printed message is made for easy reading, the actual values of the variable causing the issue are still obtainable in the element `problemValues`. If we for instance decide that the values `999` and `NaN` in `var2` are in fact miscoded missing values, we can easily replace them with `NA`s:

```
> toyData$var2[toyData$var2 %in% missVar2$problemValues] <- NA
> identifyMissing(toyData$var2}

No problems found.
```

Similarly, the `visualize()` and `summarize()` functions can be used to run the corresponding visualizations and summaries for each variable. See Figure 3.1 for the visualization output.

```
> visualize(toyData$var2)
> summarize(toyData$var2)

     Feature                   Result
[1,] "Variable type"           "numeric"
[2,] "Number of missing obs."  "3 (20 %)"
[3,] "Number of unique values" "8"
[4,] "Median"                  "4.5"
[5,] "1st and 3rd quartiles"   "1.75; 6"
[6,] "Min. and max."           "1; 999"
```

As we saw with the `check()` function, the summary can be modified by providing the relevant `XXXSummaries` arguments. Setting the `numericSummaries` argument, we can control the summary output by providing a vector of function names to run for a particular summary. To only

get the median, the minimum and the maximum we set `numericSummaries=c("centralValue", "minMax")`:

```
> summarize(toyData$var2, numericSummaries = c("centralValue", "minMax"))

     Feature         Result
[1,] "Median"        "4.5"
[2,] "Min. and max." "1; 999"
```

Note that the `summarize()`, `check()` and `visualize()` functions are also available interactively for full datasets, by calling e.g. `summarize(toyData)`. However, this produces an extensive amount of output in the console, and therefore, we generally do not recommend it, unless working with very small datasets or subsets of datasets.

4. Extending cleanR

cleanR can be used as a user-friendly, self-contained package, as shown in the previous sections. However, **cleanR** is fully customizable and `clean()` is mainly a tool for formatting the results from various checking-, summary- and visualization functions. Thus, the actual work underlying a **cleanR** output file can be anything — depending on the arguments given to `clean()` — and user made functions are easily added to the summarize/visualize/check-function arguments, as discussed previously. However, the functions used in the SVC steps must adhere to specific structures in order to be called from these three steps, and therefore, we will now present how `summaryFunction`s, `visualFunction`s and `checkFunction`s are made.

This section consists of two parts. First, we describe how custom `summaryFunction`s, `visualFunction`s and `checkFunctions` can be made, including an introduction to a few convenient tools available in **cleanR** for aiding SVC-function construction. Table 3 serves as an overview of the internals of these three central function types. Secondly, we turn to a worked example of how to use custom made functions in practice. Here, four new SVC functions are defined and used, both interactively and in `clean()`.

4.1. Function templates

In order to construct a summary-, visual or check function, one needs to create a new function with a specific structure. This can be done with different levels of strictness. If the new custom function is only to be used as part of the SVC steps in `clean()`, then only the input/output structure of the function needs special attention. However, new user-defined functions can also be registered locally to be part of the full machinery of **cleanR**, and these function will be recognized and behave in the same way as the build-in functions in **cleanR**. The presentation below is given in the format of function templates, written in pseudo-code. These templates are designed for getting the full functionality, but note that Table 3 serves as a reference to the minimal requirements, while also presenting the ”full” versions of the function types.

Writing a summaryFunction

As mentioned above, **cleanR** provides a dedicated class for `summaryFunction`s. However, this does not imply that they are particularly advanced or complicated to create; in fact, they are

nothing but regular functions with a particular input/output-structure. Specifically, they all follow the template below:

```
mySummaryFunction <- function(v, ...) {
  res <- [result of whatever summary we are doing]
  summaryResult(list(feature = "[Feature name]", result = res))
}
```

The last function called here, `summaryResult()`, changes the class of the output, thereby making a `print()` method available for it. Note that `v` is a vector and that `res` should be either a character string or something that will be printed as one. In other words, e.g. integers are allowed, but matrices are not. Though a lot of different things can go into the `summaryFunction` template, we recommend only using it for summarizing the features of a variable, and leaving tests and checks for the `checkFunction`s (presented below).

Adhering to the template above is sufficient for using the freshly made `mySummaryFunction()` in `clean()`, but we recommend furthermore adding the new function to the overview of all summary functions by converting it to a proper `summaryFunction` object. This is done by calling the `summaryFunction()` creator with the user-defined function as the first argument, and additional arguments `description` (an explanatory text which will be added to the attributes of the function), and `classes` (a vector of variable classes the user-defined function is intended to be applied to, also stored as an attribute). In other words, a call on the following form should be made:

```
mySummaryFunction <- summaryFunction(mySummaryFunction,
  description = "[Text describing what the summaryFunction does]",
  classes = c([vector of data types that the function is intended for]))
```

which adds the new function to the output of an `allSummaryFunctions()` call. If the `description` argument is left unspecified, the name of the function (which in this case is `"mySummaryFunction"`) will be filled in and stored under the `description` attribute. What happens if the `classes` argument is not specified depends on the type of `mySummaryFunction`. If `mySummaryFunction` is a `S3` generic function with associated methods, the call to `summaryFunction()` will automatically produce a vector of the names of the classes for which the function can be called. If `mySummaryFunction` is not an `S3` generic and `classes` is left unspecified, the attribute will simply be left empty. Note that the helper function `allClasses()` might be useful for filling out the `classes` argument, as it simply lists all available classes in **cleanR**:

```
> allClasses()

[1] "character" "Date"      "factor"   "integer"   "labelled"
[6] "logical"   "numeric"
```

Writing a visualFunction

`visualFunction`s are the functions that produce the figures in a **cleanR** output document. Writing a `visualFunction` is slightly more complicated than writing a `summaryFunction`. This follows from the fact that `visualFunctions` need to be able to output standalone code for plots in order for `clean()` to build standalone **rmarkdown** files. We recommend using the following structure:

```
myVisualFunction <- function(v, vnam, doEval) {
  thisCall <- call("[the name of the function used to produce the plot]",
    v, [additional arguments for the plotting function])
  if (doEval) {
    return(eval(thisCall))
  } else return(deparse(thisCall))
}
```

In this function, `v` is the variable to be visualized, `vnam` is its name (which should generally be passed to `title` or `main` arguments in plotting functions) and `doEval` controls whether the output is a plot (if `TRUE`) or a character string of standalone code for producing a plot (if `FALSE`). Implementing the `doEval = TRUE` setting is not strictly necessary for a `visualFunction`'s use in `clean()`, but it makes it easier to assess what visualization options are available, and obviously, it is crucial for interactive usage of `myVisualFunction()`. In either case, it should be noted that all the parameters listed above, `v`, `vnam` and `doEval`, are mandatory, so they must be left as is, even if they are not in use (c.f. Table 3).

As with the summary function, we call `visualFunction()` to register our newly created function:

```
myVisualFunction <- visualFunction(myVisualFunction,
  description = "[Some text describing the visualFunction]",
  classes = c([data types that this function is intended for]))
  )
```

Now, `myVisualFunction()` will be available in a `allVisualFunctions()` call, just like the two build-in `visualFunctions`, `standardVisual` and `basicVisual`.

Writing a checkFunction

The last, but perhaps most important, **cleanR** function type is the `checkFunction`. These are the functions that flag issues in the data in the check step and control the overall flow of the data cleaning process in the precheck stage. A `checkFunction` follows one of two overall structures, depending on the type of check. Either, it tries to identify problematic values in the variable (as e.g., `identifyMissing()` does), or it performs a check concerning the variable as a whole (e.g. the functions used for prechecks and the function `identifyNums()`). We present templates for both types of `checkFunctions` below separately, but it should be emphasized that formally, they belong to the same class.

First, a template for the full-variable check function type, where we first define the function and subsequently register it as a check function using `checkFunction()`:

```
myFullVarCheckFunction <- function(v, ...) {
  [do your check]
  problem <- [is there a problem? TRUE/FALSE]
  message <- "[message describing the problem, if any]"
  checkResult(list(problem = problem,
    message = message,
    problemValues = NULL))
```

```
}

myFullVarCheckFunction <- checkFunction(myFullVarCheckFunction,
  description = "[Some text describing the checkFunction]",
  classes = c([the data types that this function is intended to be used for])
  )
```

Again, as with `summaryFunction`s and `visualFunction`s, the change of function class by use of `checkFunction()` is not strictly necessary. Note however, that if `myFullVarCheckFunction` is to be used in the summarize/visualize/check steps in `clean()`, the description attribute will be printed in the overview table in the *Data cleaning summary* part of the output document.

If problematic values are to be identified, the template from above should be expanded to follow a slightly more complicated structure:

```
myProbValCheckFunction <- function(v, nMax, maxDecimals, ...) {
        [do your check]
        problem <- [is there a problem? TRUE/FALSE]
        problemValues <- [vector of values in v that are problematic]
        problemStatus <- list(problem = problem,
                              problemValues = problemValues)

        problemMessage <- "[Message that is printed prior to listing
                           problem values in the cleanR output,
                           ending with a colon]"

        outMessage <- messageGenerator(problemStatus, problemMessage, nMax)

        checkResult(list(problem = problem,
                         message = outMessage,
                         problemValues = problemValues))
}

myProbValCheckFunction <- checkFunction(myProbValCheckFunction,
  description = "[Some text describing the checkFunction]",
  classes = c([the data types that this function is intended to be used for])
)
```

One comment should be devoted to the helper function, `messageGenerator`. This function's sole purpose is aiding consistent styling of all `checkFunction` messages. The function simply pastes together the `problemMessage` and the `problemValues`, with the latter being quoted and sorted alphabetically. If `nMax` is not `Inf`, only the first `nMax` problem values will be pasted onto the message, accompanied by a comment about how many problem values were left out (if any). Note that printing quotes in **rmarkdown** requires an extensive amount of character escaping, so opting for `messageGenerator()` really is the easiest solution.

In the template above, the argument `maxDecimals` is not in use. This argument should be used to round off the `problemValues` passed to `messageGenerator()`, if they are numerical. This can be done by adding an extra line of code after defining `problemStatus` in the template above:

```
myProbValCheckFunction <- function(v, nMax, maxDecimals, ...) {
 ...
  problemStatus <- list(problem = problem,
    problemValues = problemValues)

  if (!is.null(problemValues)) {
    problemStatus$problemValues <- round(problemValues, maxDecimals)
  }
 ...
}
```

Now, problematic values will be rounded in the outputted message, while they will still appear in their original format under the entry `$problemValues` in the outputted `checkResult`.

Jeg forstår ikke helt, hvad der var uklart her ang. messageGenerator. Jeg har nu stort set bare byttet om på rækkefølgen af de to sidste paragraphs her, hjælper det?

4.2. A worked example

We will now build four new functions and show both how they can be used interactively and how they can be integrated with the `clean()` function. These four new functions are:

`isID` A new `checkFunction` intended for use in the precheck-stage. This function checks whether a variable consists exclusively of long (> 10 characters/digits) entries that are all of equal length, as this might be personal identification codes that we do not wish to print out in the data summary.

`mosaicVisual` A new `visualFunction` that produces mosaic plots. This function will be used in the *visualize* step of `clean()`.

`countZeros` A new `summaryFunction` that counts the number of occurrences of the value `0` in a variable. This function will be used in the *summarize* step of `clean()`.

`identifyColons` A new `checkFunction` that flags variables in which values have colons that appear before and after alphanumerical characters. This is practical for identifying autogenerated interaction effects. This function will be used in the *check* step of `clean()`.

These functions are defined in turn below, and afterwards, an example of how they can be called from `clean()` is provided.

isID — a new checkFunction without problem values

First, let's define the `isID()` function. As this function is not supposed to list problematic values, it falls within the category of `checkFunctions` represented by `myFullVarCheckFunction()` in the above. We do not particularly wish to use this function interactively, so we will stick to the minimal requirements of a `checkFunction` used in `check()` (see Table 3). The function can then be defined by

```
isID <- function(v, nMax = NULL, ...) {
  out <- list(problem = FALSE, message = "")
```

```
  # Should work for all but logical and Date variables
  if (class(v) %in% setdiff(allClasses(), c("logical", "Date"))) {
    v <- as.character(v)
    lengths <- c(nchar(v))
    # Check that all lengths are the same and greater than 10 characters
    if (all(lengths > 10) & length(unique(lengths)) == 1) {
      out$problem <- TRUE
      out$message <- "Warning: This variable seems to contain ID codes."
    }
  }
  out
}
```

This is essentially all we need to do in order to include this function as a precheck-function in `clean()`, so we will leave it as is and move on to the next function, namely `mosaicVisual()`.

mosiacVisual — a new visualFunction

R contains a function, `mosiacplot()`, which produces mosaic plots. We intend to use this existing high-level plotting function as part of the new visualization function. We will define the new function such that it gets the full **cleanR** functionality. This can be done using the following code.

```
mosaicVisual <- function(v, vnam, doEval) {
  # Setup the call using the existing function mosaicplot
  thisCall <- call("mosaicplot", table(v), main = vnam, xlab = "")
  if (doEval) {                    # Return the graphics
   return(eval(thisCall))
  } else return(deparse(thisCall))  # Else return the code for the call
}
```

This function can now be called directly or used in `clean()`, as will presented in an example below. Depending on the `doEval` argument, either a text string with code or a plot is produced. The plot resulting from the following call is found in Figure 4:

```
> mosaicVisual(toyData$var1, "variable 1", doEval = TRUE)
```

Even though `mosaicVisual()`, as written above, follows the style of a `visualFunction`, it is not yet truly one and therefore, it will not appear in an `allVisualFunctions()` call. In order to get this functionality, we need to change its object class. This can be done by writing

```
mosaicVisual <- visualFunction(mosaicVisual,
                        description = "Mosaic plots using graphics",
                              classes = allClasses())
```

Here, we use the function `allClasses()` to quickly obtain a vector of all the seven variable classes addressed in **cleanR**. Note that if `mosaicVisual()` were an S3 generic function, this argument could have been left as `NULL` and then the classes for which methods are available would be added automatically.

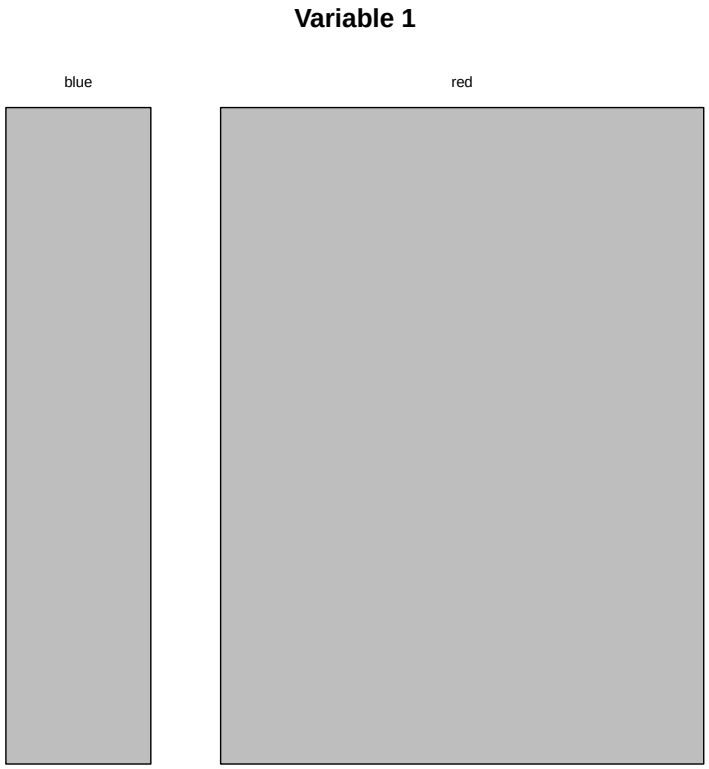


Figure 4: Mosaic plot generated by the user-defined visualization function `mosaicVisual`.

As `mosaicVisual` is now a full-blooded `visualFunction`, it will also be included in the `allVisualFunctions()` output table:

```
> allVisualFunctions()
```

```
-------------------------------------------------------------------------
name           description                    classes
-------------- ------------------------------ -------------------------
mosaicVisual   Mosaic plots using graphics    character, Date, factor,
                                              integer, labelled, logical,
                                              numeric

basicVisual    Histograms and barplots using  character, Date, factor,
               graphics                       integer, labelled, logical,
                                              numeric
standardVisual Histograms and barplots using  character, Date, factor,
               ggplot2                        integer, labelled, logical,
                                              numeric
-------------------------------------------------------------------------
```

Now that we are done with the definition of `mosaicVisual()` we can turn to the next function in line, `countZeros`.

countZeros — a new summaryFunction

This `summaryFunction` is defined in the following lines of code:

```
countZeros <- function(v, ...) {
        res <- length(which(v == 0))
        summaryResult(list(feature = "No. zeros", result = res, value = res))
}
```

As this function computes an integer (the number of zeros), there is no difference between the entries `$result` and `$value`. If, on the other hand, the result had been a character string, extra formatting might be required in the `$result` entry (such as escaping of quotation marks), and in this scenario, the two entries would have differed. As the result is returned as a `summaryResult` object, a printing method is automatically called when `countZeros` is used interactively:

```
> countZeros(c(rep(0, 5), 1:100))
```

```
No. zeros: 5
```

As with `mosaicVisual()`, we change the class of this function in order to make it appear in `allSummaryFunctions()` calls. But now we wish to emphasize that the function is not intended to be called on all variable types, as zeros have different roles in `Date`s and in `logical` variables:

```
> countZeros <- summaryFunction(countZeros,
        description = "Count number of zeros",
        classes = c("character", "factor", "integer",
                    "labelled", "numeric"))
```

identifyColons - a new checkFunction with problem values

The last function mentioned above is `identifyColons()`. We define it using the helper function `messageGenerator()` to obtain a properly escaped message, and we use `checkResult` to make its output print neatly:

```
identifyColons <- function(v, nMax = Inf, ... ) {
  v <- unique(na.omit(v))
  problemMessage <- "Note: The following values include colons:"
  problem <- FALSE
  problemValues <- NULL

  # Use regular expressions to identify colons between two words
  problemValues <- v[sapply(gregexpr("[[:xdigit:]]:[[:xdigit:]]", v),
                            function(x) all(x != -1))]

  if (length(problemValues) > 0) {
    problem <- TRUE
  }

  problemStatus <- list(problem = problem,
                        problemValues = problemValues)
  # Use the messagegenerator function to produce the output
  # message from the problemStatus and problemMessage
  outMessage <- messageGenerator(problemStatus, problemMessage, nMax)

  checkResult(list(problem = problem,
                   message = outMessage,
                   problemValues = problemValues))
}

identifyColons <- checkFunction(identifyColons,
    description = "Identify colons surrounded by alphanumeric characters",
    classes = c("character", "factor", "labelled"))
```

As with the previous two functions, we also change its class. Note, however, that for `checkFunctions`, the function description will appear in the document produced by `clean()` (in the *Data cleaning summary* section of the output), so now this is not only done for the sake of the `allCheckFunctions()` output.

Calling the new summarize/visualize/check functions from clean()

Now, we are ready to use these new functions in a `clean()` call. The extended **cleanR** output

document should have the following modifications, relative to the standard **cleanR** output:

- We want to add the new pre-check function, `isID`, to the already existing pre-checks.
- We wish to change the plot type for all variables to the new mosaic plot.
- We want the new summary function, `countZeros`, to be added to the summaries performed on all variable types but `Date` and `logical`.
- We want the new check function, `identifyColon`, to be added to the checks performed on `character`, `factor` and `labelled` variables.

These options are specified as follows: Jeg synes ikke testData egner sig her. Der er ingen koloner og CPRvar er et mærkeligt, internt navn, hvis ikke man kender danske CPR-numre. Langt de fleste variable har desuden ingen nuller. Måske skal der laves et nyt datasæt?

```
> data(testData)
> clean(testData,
      # Add the new pre-check function
      preChecks = c("isKey", "isEmpty", "isID"),
      # Change the visual
      allVisuals = "mosaicVisual",
      # Add the new summary for specified data types
      characterSummaries = c(defaultCharacterSummaries(), "countZeros"),
      factorSummaries = c(defaultFactorSummaries(), "countZeros"),
      labelledSummaries = c(defaultLabelledSummaries(), "countZeros"),
      numericSummaries = c(defaultNumericSummaries(), "countZeros"),
      integerSummaries = c(defaultIntegerSummaries(), "countZeros"),
      # Add the new check for specific data types
      characterChecks = c(defaultCharacterChecks(), "identifyColons"),
      factorChecks = c(defaultFactorChecks(), "identifyColons"),
      labelledCheck = c(defaultLabelledChecks(), "identifyColons"))
```

The outputted document is found in Appendix B.

5. Zooming in on cleaning challenges

Lidt poppet titel, men den gamle ("Using cleanR in specific situations") fik mig til flabel at tænke "i modsætning til uspecifikke situationer?". Jeg er dog meget åben for en bedre titel.

Finally, we present a few examples of how to make **cleanR** solve specific issues related to data cleaning. First, we discuss the challenges related to cleaning large datasets, particularly in terms of memory use and computation speed. Next, we show how **cleanR** can be used for problem-flagging. Lastly, we discuss how the **cleanR** output document can be included in other **rmarkdown** documents as a way to produce clear and concise documentation of a dataset.

5.1. Cleaning large datasets

If the dataset becomes very large, the standard use of `clean()` outlined above might not be ideal. If there is a vast number of variables, creation of the **rmarkdown** document might be

quite slow, while a large number of observations will generally affect the rendering time of the document. In this section, we give a few practical examples of ways to deal with large data, while wishing to still produce (potentially very long) data cleaning overview documents. Note that the interactive tools of **cleanR** can be used as usual or sequentially in small subsets of the large dataset, if no such overview documents are needed.

Handling the figures

Though figures give a nice overview of each variable, they are also quite heavy objects in terms of memory allocation. Therefore, it might be beneficial to not include figures in the **cleanR** outputs for very large datasets. This is controlled via the `mode` argument:

```
> clean(toyData, mode = c("summarize", "check"))
```

If figures are indeed needed, a different approach is to choose the less memory heavy standard R figure style instead of the **ggplot2** figures that are the default option in `clean()`. This can be done by setting the `allVisuals = "basicVisual"` argument:

```
> clean(toyData, allVisuals = "basicVisual")
```

Of course, even less heavy plots might be achieved by writing new `visualFunction`s, using the guidelines from section 4.1. For instance, a future extension of **cleanR** might be the inclusion of ASCII plots, as e.g. represented in the R package **txtplot**.

Economic memory use

Another solution, which is especially relevant to Windows users due to the unfortunate combination of memory control in this operating system and RStudio, And also just R, right? Du har udkommenterert denne kommentar, men jeg mener stadig at det er tilfældet, og jeg mener desuden ikke vi kan skrive det uden en præcision af hvad det går ud på. Hvad tænker du? is simply splitting the two steps performed by `clean()`, namely producing the **rmarkdown** file and rendering it afterwards. If the `rmarkdown` file is very long, as it will typically be for very large datasets, keeping this file open in memory wastes precious memory capacities. Therefore, we advice users to instead split the two steps. This can be done in the following manner:

```
> clean(toyData, render = FALSE, openResult = FALSE}
> render("cleanR_toyData.Rmd", quiet = FALSE)
```

This also deals with the fact that **cleanR** can produce **rmarkdown** files that supersedes the upper size limit. dette giver ingen mening uden en præcision? ”upper size limit” af hvad? Se udkommenteret kommentar.

5.2. Using cleanR for problem flagging

If the data is large, but memory issues and computation time are less of an issue than the human time it takes to look through the data cleaning document, a viable solution might be not to include all information about all variables. Or even for more reasonably sized datasets,

sometimes a brief overview of the most pressing issues can be useful. This can be achieved by using the `onlyProblematic` argument in `clean()`. By specifying `onlyProblematic = TRUE`, only variables that raise a flag in the checking steps will be summarized and visualized. But perhaps we are not even interested in obtaining general information about these variables, but only in getting a quick overview of the problems they might have. This can be done by also controlling the `mode` argument:

```
> clean(toyData, onlyProblematic = TRUE, mode = c("check"))
```

Now only the checking results are printed, and only for variables where problems were identified. An even more minimal output can be generated by also leaving out the checking results — then `clean()` essentially just produces a list of the variable names that should be investigated further:

```
> clean(toyData, onlyProblematic = TRUE, mode = NULL)
```

Of course, this can also be done without generating an overview document, by direct, interactive use of the `check()` function. When called on a `data.frame`, this function produces a list (of variables) of lists (of checks) of lists (or rather, `checkResult`s). Thus, the overall problem status of each variable can easily be unravelled using the list manipulation function `sapply()`:

```
> toyChecks <- check(toyData)
> foo <- function(x) {
>   any(sapply(x, function(y) y[["problem"]]))
> }
> sapply(toyChecks, foo)

 var1  var2  var3  var4  var5  var6
 TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

and we find that only a single variable in `toyData`, `var6` (for which all observations have the value `"Irrelevant"`), is problem-free.

5.3. Including cleanR documents in other files

Sometimes, a **cleanR** document might be a useful addition to a more general overview document, including also for example pairwise association plots, time series plots, or exploratory analysis results. To this end, it is possible to produce a **cleanR** document that can readily be included in other **rmarkdown** files. This is done by using the `standAlone` argument in `clean()`, which removes the preamble from the outputted **rmarkdown** file. Please note, that it is still necessary to indicate which `rmarkdown` type is being created; the pdf and html **rmarkdown** styles are unfortunately not identical.

If it is important that the embedded **cleanR** document can be rendered to either of these two file types, we recommend setting `twoCols = FALSE` and `output = html` in `clean()`, thereby essentially removing almost all output type specific formatting code from the generated **rmarkdown** file.

On the other hand, if a pdf document is to be produced, a few extra lines need to be added to the preamble of the master **rmarkdown** document — otherwise, the two-column layout code will produce an error. The following is an example of how such a master document preamble YAML might look like and how the `cleanR_toyData.Rmd` file can then be included:

```
---
output: pdf_document
documentclass: report
header-includes:
  - \renewcommand{\chaptername}{Part}
  - \newcommand{\fullline}{\noindent\makebox[\linewidth]{\rule{\textwidth}{0.4pt}}}
  - \newcommand{\bminione}{\begin{minipage}{0.75 \textwidth}}
  - \newcommand{\bminitwo}{\begin{minipage}{0.25 \textwidth}}
  - \newcommand{\emini}{\end{minipage}}
---

```{r, child = 'cleanR_toyData.Rmd'}
```
```

In the this example, the `cleanR_toyData.Rmd` file could have been created as follows:

```
> clean(toyData, standAlone = FALSE)
```

and the more minimal, html-style **rmarkdown** file described above can be produced using

```
> clean(toyData, standAlone = FALSE, output = "html", twoCols = FALSE)
```

Note that with the latter option, no special YAML preamble is needed in the `rmarkdown` document.

6. Concluding remarks

In this paper we have introduced the R package **cleanR** for performing reproducible error detection and data cleaning summaries. The package provides a general and extendable framework for identifying potential errors and for creating human-readable summary documents that will help investigators to identify possible errors in the data.

While the the current release is stable, the authors have an interest in further developing the functionality of **cleanR** by providing more summary, visual, and check function as part of the default package. We are also currently considering adding options to handle repeated measurement, where the visualizations — in particular — might be improved by visualizing measurements over time. In addition, an online **shiny** application where users that are not R-savvy can upload their data and get a data cleaning document is currently planned.

Alle pakker. Andet?

A. Appendix A: cleaning the toyData data frame

Part 1

Data cleaning summary

The dataset examined has the following dimensions:

Feature	Result
Number of rows	15
Number of variables	6

Checks performed

The following variable checks were performed, depending on the data type of each variable:

	character	factor	labelled	numeric	integer	logical	Date
Identify miscoded missing values	×	×	×	×	×		
Identify prefixed and suffixed whitespace	×	×	×				
Identify levels with < 6 obs.	×	×					
Identify case issues	×	×					
Identify misclassified numeric or integer variables	×	×					
Identify outliers				×	×		

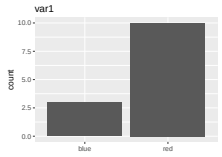
Please note that all numerical values in the following have been rounded to 2 decimals.

Part 2

Variable list

var1

Feature	Result
Variable type	factor
Number of missing obs.	2 (13.33 %)
Number of unique values	2
Mode	“red”

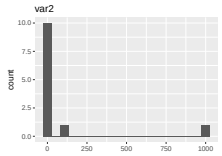


- Note that the following levels have at most five observations: "blue".

---

var2

Feature	Result
Variable type	numeric
Number of missing obs.	3 (20 %)
Number of unique values	8
Median	4.5
1st and 3rd quartiles	1.75; 6
Min. and max.	1; 999

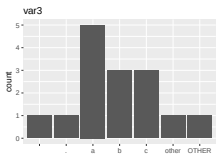


- The following suspected missing value codes enter as regular values: "999", "NaN".
- Note that the following possible outlier values were detected: "82", "999".

---

var3

Feature	Result
Variable type	factor
Number of missing obs.	0 (0 %)
Number of unique values	7
Mode	“a”

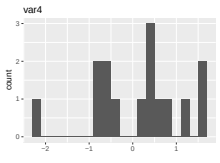


- The following suspected missing value codes enter as regular values: " ", ".".
- The following values appear with prefixed or suffixed white space: " ".
- Note that the following levels have at most five observations: " ", ".", "a", "b", "c", "other", "OTHER".
- Note that there might be case problems with the following levels: "other", "OTHER".

---

var4

Feature	Result
Variable type	numeric
Number of missing obs.	0 (0 %)
Number of unique values	15
Median	0.33
1st and 3rd quartiles	-0.62; 0.66
Min. and max.	-2.21; 1.6



- Note that the following possible outlier values were detected: "1.12", "1.51", "1.6".

---

var5

- The variable is a key (distinct values for each observation).

---

var6

- The variable only takes one (non-missing) value: "Irrelevant". The variable contains 0 % missing observations.

This report was created by cleanR v0.5.1.

B. Appendix B: User-extended cleaning of testData

Part 1

Data cleaning summary

The dataset examined has the following dimensions:

Feature	Result
Number of rows	15
Number of variables	15

Checks performed

The following variable checks were performed, depending on the data type of each variable:

	character	factor	labelled	numeric	integer	logic
Identify miscoded missing values	×	×	×	×	×	
Identify prefixed and suffixed whitespace	×	×	×			
Identify levels with < 6 obs.	×	×	×			
Identify case issues	×	×	×			
Identify misclassified numeric or integer variables	×	×	×			
Identify non-suffixed nor -prefixed colons	×	×	×			
Identify outliers				×	×	

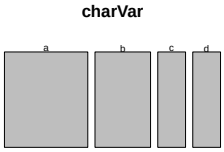
Please note that all numerical values in the following have been rounded to 2 decimals.

Part 2

Variable list

charVar

Feature	Result
Variable type	character
Number of missing obs.	1 (6.67 %)
Number of unique values	4
Mode	“a”
No. zeros	0

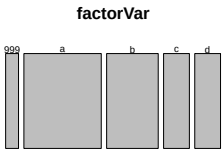


- Note that the following levels have at most five observations: "b", "c", "d".

---

factorVar

Feature	Result
Variable type	factor
Number of missing obs.	0 (0 %)
Number of unique values	5
Mode	“a”
No. zeros	0

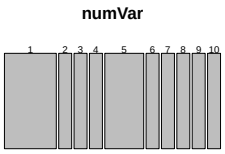


- The following suspected missing value codes enter as regular values: "999".
- Note that the following levels have at most five observations: "999", "b", "c", "d".

---

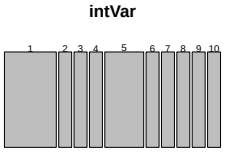
numVar

Feature	Result
Variable type	numeric
Number of missing obs.	0 (0 %)
Number of unique values	10
Median	5
1st and 3rd quartiles	1.5; 6.5
Min. and max.	1; 10
No. zeros	0



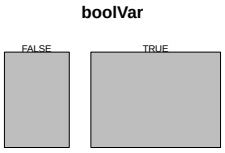
intVar

Feature	Result
Variable type	integer
Number of missing obs.	0 (0 %)
Number of unique values	10
Median	5
1st and 3rd quartiles	1.5; 6.5
Min. and max.	1; 10
No. zeros	0



boolVar

Feature	Result
Variable type	logical
Number of missing obs.	3 (20 %)
Number of unique values	2
Mode	“TRUE”



keyVar

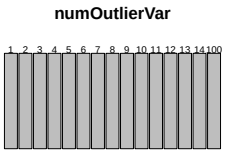
- The variable is a key (distinct values for each observation).

emptyVar

- The variable only takes one (non-missing) value: "1". The variable contains 0 % missing observations.

numOutlierVar

Feature	Result
Variable type	numeric
Number of missing obs.	0 (0 %)
Number of unique values	15
Median	8
1st and 3rd quartiles	4.5; 11.5
Min. and max.	1; 100
No. zeros	0



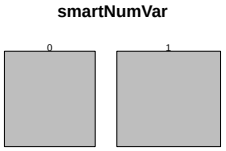
- Note that the following possible outlier values were detected: "100".

---

smartNumVar

- Note that this variable is treated as a factor variable below, as it only takes a few unique values.

Feature	Result
Variable type	numeric
Number of missing obs.	0 (0 %)
Number of unique values	2
Mode	“1”
No. zeros	7



---

cprVar

- Warning: This variable seems to contain ID codes!

---

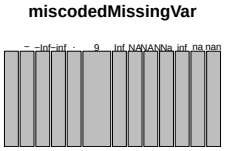
cprKeyVar

- The variable is a key (distinct values for each observation).
- Warning: This variable seems to contain ID codes!

---

miscodedMissingVar

Feature	Result
Variable type	character
Number of missing obs.	0 (0 %)
Number of unique values	14
Mode	“9”
No. zeros	0

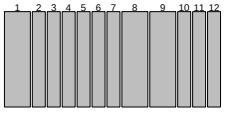


- The following suspected missing value codes enter as regular values: "", "-", "-Inf", "-inf", ".", "9", "Inf", "NA", "NAN", "Na", "NaN", "inf", "na", "nan".
- Note that the following levels have at most five observations: "", "-", "-Inf", "-inf", ".", "9", "Inf", "NA", "NAN", "Na", "NaN", "inf", "na".
- Note that there might be case problems with the following levels: "-Inf", "-inf", "Inf", "NA", "NAN", "Na", "NaN", "inf", "na", "nan".

---

misclassifiedNumVar

Feature	Result
Variable type	factor
Number of missing obs.	0 (0 %)
Number of unique values	12
Mode	“1”
No. zeros	0

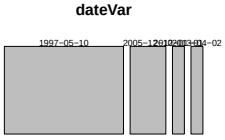


- Note that the following levels have at most five observations: "1", "10", "11", "12", "2", "3", "4", "5", "6", "7", "8", "9".
- Note: The variable consists exclusively of numbers and takes a lot of different values. Is it perhaps a misclassified numeric variable?

---

dateVar

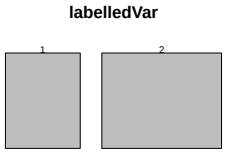
Feature	Result
Variable type	Date
Number of missing obs.	0 (0 %)
Number of unique values	4
Mode	“1997-05-10”
Min. and max.	1997-05-10; 2013-04-02
1st and 3rd quartiles	1997-05-10; 2005-12-10



---

labelledVar

Feature	Result
Variable type	labelled
Number of missing obs.	2 (13.33 %)
Number of unique values	2
Mode	"2"
No. zeros	0



This report was created by cleanR v0.6.2.

Affiliation:

Claus Thorn Ekstrøm
Biostatistics, Department of Public Health
University of Copenhagen
Denmark
E-mail: ekstrom@sund.ku.dk
URL: http://staff.pubhealth.ku.dk/~ekstrom/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd

	`summaryFunction`	`visualFunction`	`checkFunction`
Input (required)	`v` - a variable vector `...`	`v` - a variable vector `vnam` - the variable name (as character string) `doEval` - a logical (`TRUE`/`FALSE`) controlling the output type of the function	`v` - a variable vector `nMax` - an integer (or `Inf`), controlling how many problematic values are printed, if relevant `...`
Input (optional)	`maxDecimals` - number of decimals printed in outputted numerical values.	-	`maxDecimals` - number of decimals printed in outputted numerical values.
Purpose	Describe some aspect of the variable, e.g. a central value, its dispersion or level of missingness.	Produce a distribution plot.	Check a variable for a specific issue and, if relevant, identify the values in the variable that cause the issue.
Output (required)	A list with entries: `$feature` - a label for the summary value (as character string); `$result` - the result of the summary (as character string)	A character string with `R` code for producing a plot. This code should be standalone, i.e. should include the data if necessary.	A list with entries: `$problem` - a logical identifying whether an issue was found; `$message` - a character string (possibly empty) decribing the issue that was found, properly escaped and ready for use in `rmarkdown`
Output (recommended)	A `summaryResult` object, i.e. an attributed list with entries `$feature`, `$result` and `$value`, the latter being the values from `$result` in their original format).	If `doEval` is `TRUE`: A plot that will be opened by the graphic device in `R`. / If `doEval` is `FALSE`: A text string with `R` code, as described above.	A `checkResult` object, i.e. an attributed list with entries `$problem`, `$message` and `$problemValues`, the latter being either `NULL` or the problem causing values, as they were found in `v`, whichever is relevant.
Tools available for producing the function	`summaryResult()`	-	`messageGenerator()` `checkResult()`

Table 3: Reference information for creating new functions to be used as part of the summarize-, visualize-, and check steps. The three columns correspond to each of the three function types. I think the formatting here is still sort of chaotic. Maybe include horizontal lines everywhere? Is it allowed in JSS? It's impossible to read what's in which row right now.