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 coding — but that process can be very time-consuming, expensive and error-prone by itself.

We describe an R package which implements an extensive and customizeable suite of quality assessment tools to 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 and the experimental design, but not necessarily knowledge of R. Thereby, **cleanR** aids the dialogue between data analysists 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. Hvad henvises der til med denne sætning?

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 provide an unsupervised

decision on what should be done to potential errors in the dataset, it can be an extremely useful tool in tracking down and flagging potential erroneous data and in providing information for humans to easier identify errors based on the context.

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 on the general researcher with a knowledge about the specific field and 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 build. This quality assessment is presented in an auto-generated overview document, readable by data analysts and field experts alike, thereby also contributing to a 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 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 such **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 proper ref, we discuss a number of examples of specific data cleaning challenges and how **cleanR** can be used to solve them.

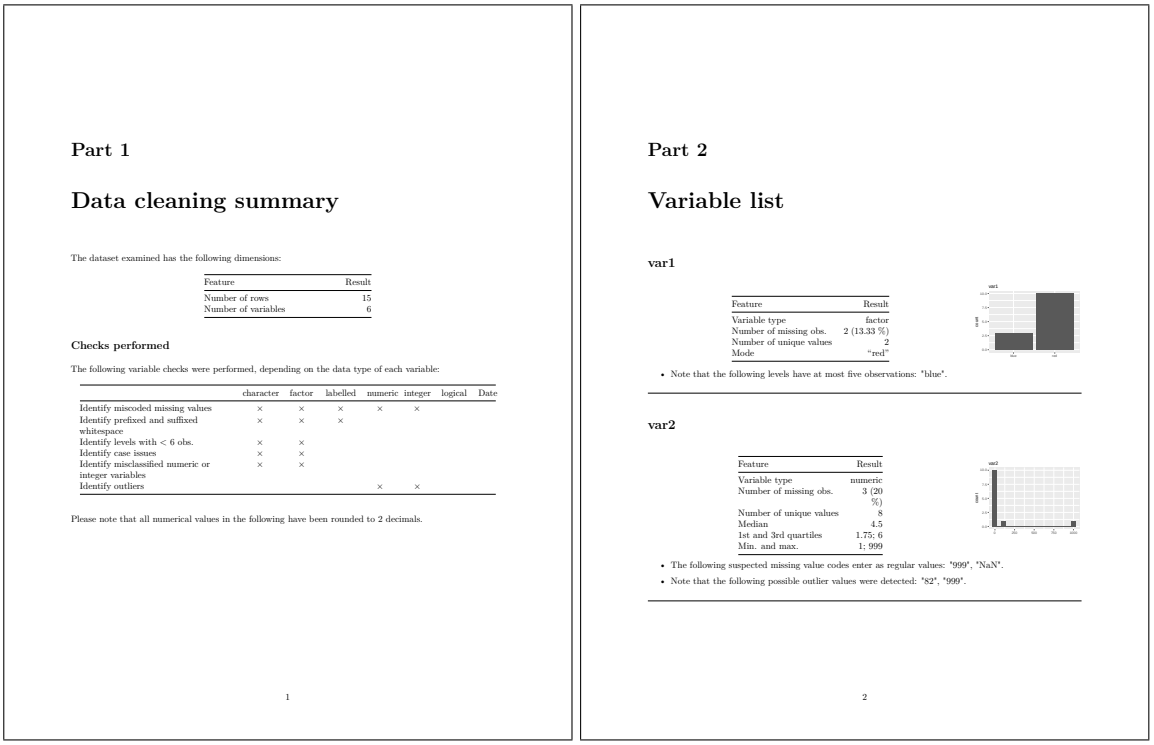
2. Creating a data cleaning overview

The `clean()` function is the primary workhorse of **cleanR** and this is the only function needed by the user if the standard battery of tests are used to output the data cleaning summaries. The data cleaning output 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 and 6 variables), artificial dataset, whose only purpose is 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, a 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 potentially also a list of flagged issues. 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. In the following we give examples on how to use these parameters to influence the output of `clean()` .

For example, to get a summary document that only contains the variables with potential problems, and with a limit of maximum 10 printed potential errors, we can write (output not shown)

Argument	Description	Default value
Control input variable 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 by listed in the *Data cleaning summary*?	`TRUE`
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")`
`labelled_as`	How should variables of class `labelled` be handled (as factors, is missing values or by ignoring labels)?	`"factor"`
`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?	`TRUE`

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

Input A dataset

- This should be of type `data.frame` or `tibble` or data.table?

Create contents For each variable in the dataset, do the following:

Stage 1: Pre-checks

- Is the variable suitable for summarization, visualization and checks?

  Yes: Go to stage 2.

  No: Move on to the next variable.

Stage 2: SVC-steps

Summarize Call `summarize()` to produce a summary table describing the variable. What features enter this table depends on the data class of the variable.

Visualize Call `visualize()` to produce a plot visualizing the distribution of the variable.

Check Call `check()` to apply quality- and error checks to the variable. What checks are used depends on the data class of the variable.

Output Files for a overview document are saved to the disc and possibly also opened.

- Always a `rmarkdown` (.Rmd) file
- Possible also a html or pdf file

Figure 2: quite odd having a figure like this? maybe do a flowchart?

```
> clean(toyData, onlyProblematic=TRUE, maxProbVals=10)
```

The final rendering of the generated

By default, `clean()` runs all the summary, visualization, and check functions that are implemented. Table 2 lists these functions but we can also use the `allSummaryFunctions()`, `allVisualFunctions()`, and `allCheckFunctions()` function in R to obtain a list. For example the implemented summary functions are:

- Introduce the relevant `clean`-arguments and the `defaultWhateverSummaries` etc.-functions
- Introduce `allSummaryFunctions()` etc. and present a table corresponding to the output of this call
- Small examples:
  - Add a function to one of the XXXSummaries/XXXVisuals/XXXChecks-arguments (still calling default options)
  - Remove all but a single function from one of these arguments
  - Describe what happens if the argument is NULL

	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 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: Blabla, mention that C is character, F is factor, I is integer, L is labelled, B is logical (boolean), N is numeric and D is Date.
Also: Check that everything in here is correct (i.e. corresponds to the output of allSummaryFunctions() etc when makeXfunction has been fixed.

```
> 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       integer, numeric, Date
               values

quartiles      Compute 1st and 3rd quartiles  integer, numeric

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

variableType   Data class of variable         character, Date,
factor,
                                              integer, labelled,
                                              logical,
                                              numeric
-------------------------------------------------------------------------
```

In this section, we discuss how to control every step in `clean()` that actually involves a function being called on variables from the dataset. There are two stages in which this occurs, as mentioned in Figure 2, namely:

1. In the precheck functions
2. In the summarize/visualize/check (SVC) step

Each of these stages are controllable using appropriate function parameters in `clean`. In the above, we presented the default **cleanR** settings and how to tweak them into providing a slightly different data cleaning outputs. However, if for instance the dataset at hand requires completely different visualizations, more control is needed. **cleanR** uses three different types of functions for performing all stages in the above, namely `summaryFunctions`,

visualFunctions and checkFunctions. Something like "the avaialable options for functions used in the precheck- and check steps are obtainable by calling allCheckFunction(), blablabla, similarly with visualFunctions and summaryFunctions. Mention Section 4 and the fact that one can expand the possibilities of e.g. allVisualFunctions() by producing new functions quite easily.

3. Using cleanR interactively

While overview documents are great for presenting and documenting the data cleaning checks, it may be natural to work more interactively through the data cleaning process. **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 the result of the check for miscoded missing values, we write

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

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

The `numericChecks` argument XXX.. ... not defined

An equivalent way to call only a single, specific `checkFunction` such as `identifyMissing` (see 2 for a list of check functions) is by using it directly on the variable, i.e.

```
> 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"
```

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.
```

- Do an example with visualize() and summarize(), like the one with check(). Especially visualize and to doEval = T thing needs a bit of special attention.
- Mention allCheckFunctions() etc. again here
- Mention check(), visualize() and summarize() modes for data.frames. Maybe also advice against it, at it will often produce a lot of information at once, and such large amounts of information really should be documented.

4. Extending cleanR

Though the discussion in the above paints a picture of cleanR as a user-friendly package which requires practically no knowledge of `R`, one should not be mistaken to think that it is not customizable. In fact, the main function of **cleanR**, `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 or nothing - depending on the arguments given to `clean`. Specfically, user made functions can be added to the SVC-function arguments discussed above, e.g. `factorSummaries`, `allVisuals` or `numericChecks`: All that is needed is to specify their names, just like the names of the build-in SVC functions are specified. However, not just any function can be called from these three steps, and therefore, we will now present how `summaryFunctions`, `visualFunctions` and `checkFunctions` are made. Mention something about the interactive mode here as well?

This section consists of two parts. First, we describe how the `clean` function can be extended by adding custom prechecks, summaries, checks and visualizations. In order to do this, one

	`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 (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: blabla. Fix formatting! Maybe use multirow and raggedRight stuff?

needs to produce small functions that obey to the syntax of **cleanR**. This can be done with different levels of strictness. If the custom functions are only to be used by `clean()`, only the input/output structures of the functions need to be restricted. However, by adding a few extra word? to these functions, the full machinery of **cleanR** becomes available for the new, user-made functions, just as it is for the build-in functions. 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 please note that Table 3 serves as a reference to the minimal requirements, while also presenting the "full" versions of the function types.

After this abstract, templatic overview of the internals of the three SVC functions is given, we turn to a worked example of how to use custom made functions in practice in ref to worked example section. Here, four new SVC functions are defined and used, both interactively and in `clean()`.

4.1. Function templates

metatext, mention Table 3 again.

Writing a summaryFunction

As mentioned above, **cleanR** provides a dedicated class for `summaryFunctions`. However, this does not imply that they are particularly advanced or complicated to create; in fact, they are nothing but regular functions with a certain 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 `checkFunctions` (presented below).

Though adhering to the template above is sufficient for using the freshly made `mySummaryFunction()` in `clean()`, we recommend furthermore adding it to the overview of all summary functions by converting it to a proper `summaryFunction` object. This is done by writing

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

which adds the new function to the output of an `allSummaryFunctions()` call. One comment should be devoted to the two attributes of a `summaryFunction`. If the `description` argument is left unspecified, the name of the function (in this case, `"mySummaryFunction"`) will be filled in. 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 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"
```

Write something here, don't end paragraph with code. Also, maybe move the allClasses() stuff somewhere else, it doesn't really belong under this header. Not sure where to, though.

Writing a visualFunction

`visualFunction`s are the functions that produce the figures of 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 to the plotting function])
  if (doEval) {
    return(eval(thisCall))
  } else return(deparse(thisCall))
}

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

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.

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` can follow 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 `checkFunction`s 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:

```
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 SVC step in `clean()`, the description attribute will be printed in the overview table in the *Data cleaning summary*.

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 <- "[The message that should be 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])
  )
```

In this template, 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 is done by substituting the `problemStatus` assignment above with the following code:

```
problemStatus <- list(problem = problem,
  problemValues = round(problemValues, maxDecimals))
```

Another noteworthy component of the template is the usage of the helper function `messageGenerator()`, which aids consistent styling of all `checkFunction` messages. This function simply pastes together the `problemMessage` and the `problemValues`, with the latter being quoted and sorted

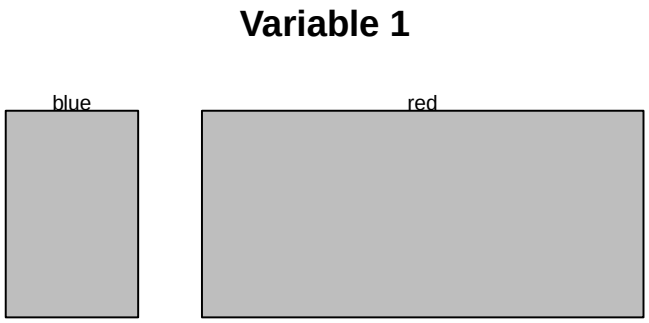


Figure 3: something, note that code for producing this plot is available in ./latex/codeForArticle.R

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.

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 so-called 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 e.g. 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 in the variable, 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 = "")
  if (class(v) %in% setdiff(allClasses(), c("logical", "Date"))) {
    v <- as.character(v)
    lengths <- c(nchar(v))
    if (all(lengths > 10) & length(unique(lengths)) == 1) {
      out$problem <- TRUE
      out$message <- "Warning: This variable seems to contain ID codes."
    }
  }
  out
}
```

Mention somewhere that `mosiacplot()` is a base-R (`graphics`) function? Otherwise, it might not be completely clear, that we are passing a function name in the above... 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

We will define this function such that it gets the full **cleanR** functionality. This can be done using the code

```
mosaicVisual <- function(v, vnam, doEval) {
  thisCall <- call("mosaicplot", table(v), main = vnam, xlab = "")
  if (doEval) {
   return(eval(thisCall))
  } else return(deparse(thisCall))
}
```

This function can now be called directly or used in `clean()`. We will return to its usage in `clean()` below. Depending on the argument `doEval`, either a text string with code or a plot is produced. The plot resulting from the following call is found in Figure 4.2:

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

remove $. 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 a `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. I'm repeating myself here, but I think it is quite a neat feature, so maybe that's okay?

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
               graphics

standardVisual Histograms and barplots using
               ggplot2
-------------------------------------------------------------------------
```

Redo this output table when makeXFunction issues are fixed.

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

countZeros - a new summaryFunction

This `summaryFunction` in spe 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))
}
```

Note that as this function computes an integer (the number of zeros), there is no difference between the entires `$result` and `$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 then differ. 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"))
```

more? don't end on code.

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

  problemValues <- v[sapply(gregexpr("[[:xdigit:]]:[[:xdigit:]]", v),
                            function(x) all(x != -1))]

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

  problemStatus <- list(problem = problem,
                        problemValues = problemValues)
  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 `checkFunction`s, the function description will appear in the document produced by `clean()` (in the *Data cleaning summary* section), so now this is not only done for the sake of the

`allCheckFunctions()` output.

Calling the new SVC functions from clean()

Now, we are ready to use these new functions in a `clean()` call. do this... what dataset should we use here? Include output in appendix, maybe.. 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:

```
> clean(dataSet,
      preChecks = c("isKey", "isEmpty", "isID"),
      allVisuals = "mosaicVisual",
      characterSummaries = c(defaultCharacterSummaries(), "countZeros"),
      factorSummaries = c(defaultFactorSummaries(), "countZeros"),
      labelledSummaries = c(defaultLabelledSummaries(), "countZeros"),
      numericSummaries = c(defaultNumericSummaries(), "countZeros"),
      integerSummaries = c(defaultIntegerSummaries(), "countZeros"),
      characterChecks = c(defaultCharacterChecks(), "identifyColons"),
      factorChecks = c(defaultFactorChecks(), "identifyColons"),
      labelledCheck = c(defaultLabelledChecks(), "identifyColons"))
```

The outputted document is found in Appendix NUMTWO. Comment. Make appendix. Remember to change dataSet to something else, when we decide what data to use here.

5. Something like examples

Finally, we will 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 mean to produce clear and concise documentation of a dataset. I feel like there should be more topics here, but I'm all out of ideas...

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, production of the `rmarkdown` document might

be quite slow, while an extensive amount of observations generally affects the rendering time of this 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.

Attacking 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`reference? figures that are the default option in `clean()`. This can be done using the `allVisuals` 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`reference here? It's on CRAN..

I really feel like we should do some benchmarking here, maybe just on toyData, both in terms of speed and memory use. I would make the recommendations more trustworthy and serious.

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? There's got to be a nice reference on this..?, 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 in very large datasets, having this file opened in memory waists 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 of RStudio, which is currently find number GBs (using RStudio version 1.0.44). Is this maybe too editor specific? On the other hand, a lot of people do use RStudio....

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 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, `checkResults`). 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 the final variable, `var6`, for which all observations have the value `"Irrelevant"`, is problem-free. drop this last bit? too technical?

5.3. Include cleanR document in other files

Sometimes, a **cleanR** document might be a useful addition to a more general overview document, including also e.g. 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 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'}
```
```

Use proper formatting here. How do we do non-R code?

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)
```

don't end on code.

6. Conclusion/Concluding remarks/summary/?

Is this a thing in this journal? Otherwise, we might want to make some final remarks in the previous sections. Feels awkward to end with a bunch of code and some super specific examples...

A. Appendix Something

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				×	×		

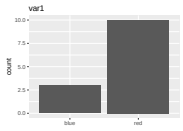
Please note that all numerical values in the following have been rounded to 2 decimals.

Part 2

Variable list

var1

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

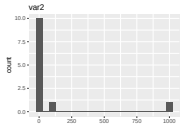


- Note that the following levels have at most five observations: "blue".

---

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

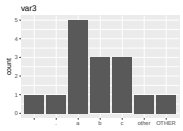


- The following suspected missing value codes enter as regular values: "999", "NaN".
- Note that the following possible outlier values were detected: "82", "999".

---

var3

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

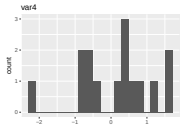


- The following suspected missing value codes enter as regular values: " ", ".".
- The following values appear with prefixed or suffixed white space: " ".
- Note that the following levels have at most five observations: " ", ".", "a", "b", "c", "other", "OTHER".
- Note that there might be case problems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

B. Appendix NUMTWO

Data cleaning with user supplied extensions here

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