Data Validation Infrastructure for R

Mark P.J. van der Loo
Statistics Netherlands

Edwin de Jonge
Statistics Netherlands

Abstract

Checking data quality against domain knowledge is a common activity that pervades statistical analysis from raw data to output. The R package **validate** facilitates this task by capturing and applying expert knowledge in the form of validation rules: logical restrictions on variables, records, or data sets that should be satisfied before they are considered valid input for further analysis. In the **validate** package, validation rules are objects of computation that can be manipulated, investigated, and confronted with data or versions of a data set. The results of a confrontation are then available for further investigation, summarization or visualization. Validation rules can also be endowed with metadata and documentation and they may be stored or retrieved from external sources such as text files or tabular formats. This data validation infrastructure thus allows for systematic, user-defined definition of data quality requirements that can be reused for various versions of a data set or by data correction algorithms that are parameterized by validation rules.

Keywords: data checking, data quality, data cleaning, R.

Citation. MPJ van der Loo and E de Jonge (2019), *Data Validation Infrastructure for R*, Journal of Statistical Software (accepted for publication).

1. Introduction

Checking whether data satisfy assumptions based on domain knowledge pervades data analyses. Whether it is raw data, cleaned up data, or output of a statistical calculation, data analysts have to scrutinize data sets at every stage to ensure that they can be used for reporting or further computation. We refer to this procedure of investigating the quality of a data set and deciding whether it is fit for purpose as a 'data validation' procedure.

Many things can go wrong while creating, gathering, or processing data. Accordingly there are many types of checks that can be performed. One usually distinguishes between technical checks that are related to data structure and data type, and checks that are related to the topics described by the data. Examples of technical checks include testing whether an 'age' variable is numeric, whether all necessary variables are present, or whether the identifying variable (primary key) is unique across records. Examples of tests that relate to subject matter knowledge include range checks (the age of a person should be between 0 and say, 120), and consistency checks between variables (a person under the age of 18 can not be married). Some checks require simple comparisons, while others involve elaborate calculations. For example consider a check where we test that the mean profit to turnover ratio in a certain economic sector has not changed more than 10% between the current quarter and the same quarter last

year. To execute such a check we need data on profit and turnover for two quarters, compute their ratios (taking exceptions such as dividing by zero into account), average them, and then compare the relative absolute difference with a fixed value (0.1).

Since data validation is such a common and diverse task, it is desirable to use a dedicated tool that allows one to systematize and organize data validation procedures. There are several recent contributions to the R language (R Core Team 2019) that aim to support this in a systematic way. The **pointblank** package of Iannone (2018) allows users to test data against a number of hard-coded validation checks. For example, there are separate functions to determine whether values in a column are positive, nonnegative, or in a certain range. The **dataMaid** package (Petersen and Ekstrøm 2019) creates generic summary reports and figures that are aimed at detecting common problems with data. Detected problems include the presence of special values (such as `Inf`), empty strings and identifying potential outliers. Fischetti (2019) published the **assertr** package, which allows for checking data against a number of predefined (and parameterized) data checks called 'predicates'. It is specifically geared for use with a function composition operator such as the **magrittr** 'forward pipe' operator of Bache and Wickham (2014). Depending on the outcome of a predicate a custom function can be evaluated and the utility of **assertr** is therefore comparable to a try-catch mechanism. The **editrules** package (de Jonge and van der Loo 2018) allows users to specify their own checks (called 'edit rules') and apply them to data. The checks that can be specified are limited to simple in-record linear equalities and inequalities over the variables, and certain conditional checks (e.g., `IF age < 18 THEN married == FALSE`).

The **validate** package (van der Loo and de Jonge 2019) presented in this paper takes an approach that is more general than the packages mentioned above. Specifically, **validate** does not limit the type of checks by hard-coding them or by limiting them to certain rule types. Instead it provides a convenient domain specific language (DSL) in which the user can express any type of validation task in the form of any number of 'validation rules'. The package takes care of reading, parsing and applying the rules to the data and offers tools to analyze and visualize the results. Moreover, it treats these validation rules as first-class citizens so they can be organized, manipulated, and analyzed like any other data type.

Defining a language specifically for the definition of validation rules has several advantages. First, a formal language necessarily implements a strict demarcation of the concept of 'data validation'. A data validation language therefore incites users to separate their thinking about data quality requirements from the data processing work flow. Second, expressing data quality requirements in the form of a formal data validation language allows for unambiguous and clear communication of data quality or data quality requirements between data producers and consumers. Third, it allows for systematic development and maintainance of data quality requirements based on established techniques for developing source code. This includes practices like testing, reviewing, documenting, and version control. Fourth, it allows for the reuse of validation rules for purposes other than quality measurement. Important examples include the use of algorithms that can automatically find errors and adjust data to satisfy the validation rules (de Waal, Pannekoek, and Scholtus 2011; van der Loo and de Jonge 2018). Some examples of this are reported in Section 5. Finally, treating validation rules as enities on their own opens up interesting new avenues of inquiry including questions like: 'which rules were violated most often?' and 'what variables were involved in these rules?' Such statistics can provide valuable clues to issues related to data gathering and processing that occurred prior to data validation.

The current version of the package is limited to validating data that is represented as records in an R data frame. This means that in its current form other useful data types including networks, geographical data, time series or (high-dimensional) arrays are excluded. However, the package is extendable so it is possible to add such functionality without altering its associated workflow or syntax.

In the remainder of this paper we first give a more precise definition of data validation (Section 2). Next, in Section 3 we discuss how this definition is implemented in a domain specific language and demonstrate the basic data validation work flow with the package. We also demonstrate how **validate** treats validation as first-class citizens, allowing users to create, read, update, and delete (CRUD) rules interactively. Rules can also be endowed with metadata such as names and descriptions and we discuss how rules can be imported from, and exported to several formats. Section 3 also discusses options such has how to control the accuracy when testing numerical equalities during data validation, and how to analyse and visualise the results of a data validation. In Section 4 we provide some background and discuss the object model that is implemented by the package. In Section 5 we demonstrate how **validate** can be used to both control and monitor a small data cleaning task. We conclude with a summary and outlook in Section 6.

2. Data validation

Intuitively, data validation is a decision-making process where, based on data investigations, one decides whether the data set is fit for its intended purpose. This notion is made more precise in the following definition, which is currently the operational definition for the European Statistical System[1] (Di Zio, Fursova, Gelsema, Giessing, Guarnera, Ptrauskiene, von Kalben, Scanu, ten Bosch, van der Loo, and Walsdorfe 2015).

> Data validation is an activity in which it is verified whether or not a combination of values is a member of a set of acceptable value combinations.

In other words, one considers the collection of all datasets that might be observed, and defines data validation as a procedure that selects the datasets that are acceptable for further use. This definition is general enough to permit a precise formal definition but also includes the option to approve data by expert review.

To develop the formal side, the following definition is sufficient for our purpose (van der Loo and de Jonge 2020).

> A *data validation function* is a function v that accepts a data set and returns an element in $\{\texttt{TRUE}, \texttt{FALSE}\}$. There is at least one data set s for which $v(s) = \texttt{TRUE}$ and at least one data set s' for which $v(s') = \texttt{FALSE}$.

We say that s *passes* v if $v(s) = \texttt{TRUE}$ and that s *fails* v if $v(s) = \texttt{FALSE}$. The definition requires that validation functions are surjective on $\{\texttt{TRUE}, \texttt{FALSE}\}$ for the following reasons.

[1] The European Statistical System (ESS) is a partership between Eurostat (the European Union statistical authority) and statistical authorities of member states of the European Union, the European Free Trade Association (EFTA) and the European Economic Area (EEA). See https://ec.europa.eu/eurostat/web/european-statistical-system

On one hand, if $v(s) =$ `TRUE` for every possible data set s, it does not distinguish between valid and invalid data (a clear property demanded by the ESS definition). On the other hand, if $v(s) =$ `FALSE` for each possible data set s, then v must be a contradiction since no data can ever satisfy the demand expressed by v.

It is possible to define the concept of 'each possible data set' formally by defining data sets as collections of key-value pairs in a precise way. For a complete formal treatment of validation functions and some of their general properties we refer the reader to van der Loo and de Jonge (2020), Di Zio *et al.* (2015, Section 5) or van der Loo and de Jonge (2018, Chapter 6). For now, it is important to note that the definition does not pose any restriction on the form of data set that is under scrutiny. In the current definition a data set is viewed as an unstructured set of key-value pairs and we do not impose a topology such as relational or network structure.

2.1. Validation functions and validation rules

A validation function can be defined by fixing a set of restrictions on a data set. For example consider a data set with variables `age` and `employment`. Before using the data set in a statistical analysis we wish to ensure that age is in the range $[0, 120]$ and that persons under the age of 15 do not have a paid job (have no employment). Knowing to what population the data pertains, we also find it highly implausible to have an unemployment fraction exceeding 30%. These demands can be expressed as a rule set

$$\begin{cases} \texttt{age} \geq 0 \text{ for all ages} \\ \texttt{age} \leq 120 \text{ for all ages} \\ \texttt{IF employment} == \texttt{"employed" THEN age} \geq 15 \text{ for all age-employment combinations} \\ (\text{nr. of records with } \texttt{employment} == \texttt{"unemployed"})/(\text{nr. of records}) \leq 0.3, \end{cases}$$

The corresponding validation function evaluates each rule on a data set and returns the logical conjunction (`AND`) of their values.

Each restriction is ultimately a logical predicate, with propositions that possibly consist of complex calculations on the data. In **validate** this is implemented by allowing users to define a set of restrictions in the form of R expressions that evaluate to a scalar or vector of type `logical`. The package then takes care of confronting rules with data and of the administration of the rules and results.

3. Data validation with validate

A data validation task can be split up in three consecutive subtasks: loading the data and the validation rules, confronting the data with the rules, and transforming and analyzing the results. In what follows, we first demonstrate this workflow with an example. Next, detailed descriptions of the most important features of the package are given in Sections 3.1–3.7. An overview of the workflow is also given in Figure 1.

In the following we use the `retailers` data set included with the package. This data set consists of 60 records with information on supermarkets, such as the number of staff employed, several income amounts and several cost items.

```
R> library("validate")
R> data("retailers")
R> head(retailers[3:8], 3)
```

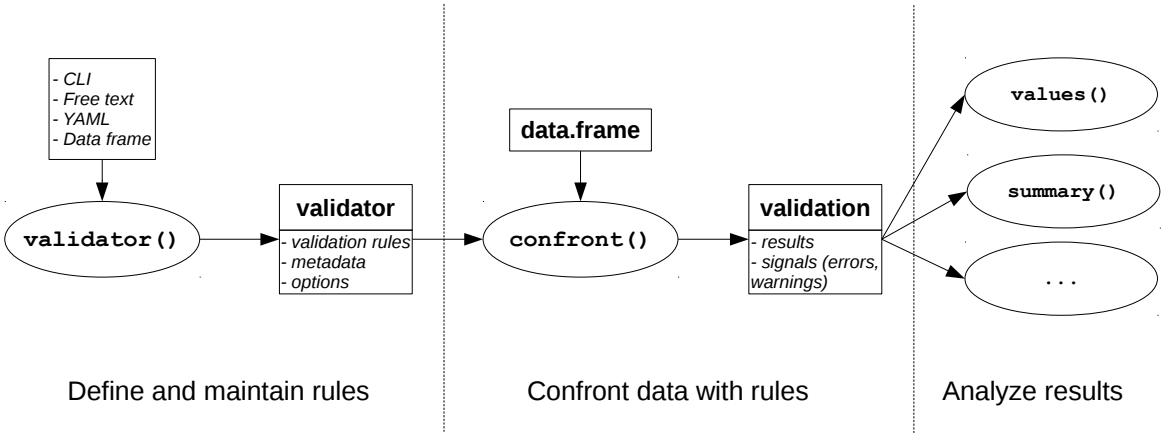

Figure 1: Overview of the main workflow and objects in **validate**.

```
  staff turnover other.rev total.rev staff.costs total.costs
1    75       NA        NA      1130          NA       18915
2     9     1607        NA      1607         131        1544
3    NA     6886       -33      6919         324        6493
```

```
R> retailers$id <- sprintf("RET%2d",1:nrow(retailers))
```

In the last expression we add a primary key to be better able to identify results later. Having access to a unique record identifier is not strictly necessary but it does make connecting validation results to original data easier later on in this example.

To test this data we first capture a set of restrictions in a '`validator`' object, using a function of the same name. We start with a few rules defined on the command-line.

```
R> rules <- validator(
+     st    = staff     >= 0
+   , to    = turnover  >= 0
+   , or    = other.rev >= 0
+   , st.cs = if (staff > 0 ) staff.costs > 0
+   , bl    = turnover + other.rev == total.rev
+   , mn    = mean(profit, na.rm=TRUE) >= 1
+ )
```

The first three rules are non-negativity restrictions. The fourth rule indicates that when a retailer has employees (staff), there must be associated costs. The fifth rule is an account balance check and the last rule expresses that perhaps some individual retailers have negative profit (loss), but the sector as a whole is expected to be profitable. Each rule is given a (very) short name. For instance the first rule is named `"st"`.

To test the data against these rules, we use the function `confront`.

```
R> confront(retailers, rules, key="id")
```

```
Object of class 'validation'
Call:
    confront(dat = retailers, x = rules, key = "id")

Confrontations: 6
With fails    : 2
Warnings      : 0
Errors        : 0
```

The resulting object holds validation results and some metadata on the data validation procedure. When printed, it shows the number of 'confrontations' performed (there are six rules) and number of confrontations that resulted in at least one failure. The number of warnings and errors do not refer to data failing a rule. Rather, they show whether trying to execute a rule raised a warning or error. An error occurs when a rule cannot be executed, for example when it refers to a variable that is not in the data set.

```
R> # We use a variable not occurring in the dataset
R> badrule <- validator(employees >= 0)
R> confront(retailers, badrule)

Object of class 'validation'
Call:
    confront(dat = retailers, x = badrule)

Confrontations: 1
With fails    : 0
Warnings      : 0
Errors        : 1
```

In section §3.6 we explain in more detail how to handle errors or warnings. Here, we return to the original rule set stored in `rules` and summarize the results.

```
R> check <- confront(retailers, rules, key="id")
R> summary(check)

   name items passes fails nNA error warning
1    st    60     54     0   6 FALSE   FALSE
2    to    60     56     0   4 FALSE   FALSE
3    or    60     23     1  36 FALSE   FALSE
4 st.cs    60     50     0  10 FALSE   FALSE
5    bl    60     19     4  37 FALSE   FALSE
6    mn     1      1     0   0 FALSE   FALSE
                                    expression
1                        (staff - 0) >= -1e-08
2                     (turnover - 0) >= -1e-08
3                    (other.rev - 0) >= -1e-08
4              !(staff > 0) | (staff.costs > 0)
```

```
5 abs(turnover + other.rev - total.rev) < 1e-08
6               mean(profit, na.rm = TRUE) >= 1
```

The `summary` method returns a data frame where each row represents one validation rule. The second column (items) lists how many results each rule returned, i.e., how many items were checked with the rule. Here, the first five rules are executed for each row in the data set so there are 60 items (each row is one item). The last rule returns a single value. Calculating this value involves comparing the mean profit with a positive number, so the 'profit' column is the single item under scrutiny.

The next two columns list how many items passed, or failed the test. The `nNA` column shows how many tests resulted in `NA` because one or more of the data points needed to evaluate the rule were missing. The columns named `error` and `warning` indicate whether an error or warning occurred during evaluation, and the last column shows the actual expression used to evaluate the rule. Depending on the rule, the original expressions are manipulated, for example to account for machine rounding errors (rule 1–3, and 5) or to vectorize the statement (rule 4). The choices made in these conversions can be influenced with parameters that will be detailed in Subsection 3.7.

The full set of results can be extracted, for example in the form of a data frame:

```
R> output <- as.data.frame(check)
R> tail(output,3)
```

```
       id name value                                      expression
299 RET59   bl    NA abs(turnover + other.rev - total.rev) < 1e-08
300 RET60   bl    NA abs(turnover + other.rev - total.rev) < 1e-08
301  <NA>   mn  TRUE               mean(profit, na.rm = TRUE) >= 1
```

The output is a record for each validation on each item, with columns identifying the item (if possible), the name of the validation rule, the result and the R expression used in obtaining the result.

A very short summary of the results can be extracted with `all`, which returns `TRUE` only when all tests are passed.

```
R> # passing all checks?
R> all(check)
```

```
[1] FALSE
```

```
R> # ignore missings:
R> all(check, na.rm=TRUE)
```

```
[1] FALSE
```

With `plot` we can quickly visualize the results. Here, we only visualize results of rules 1–5 because these are the rules rules that are applied to more than one item. The result is shown in Figure 2.

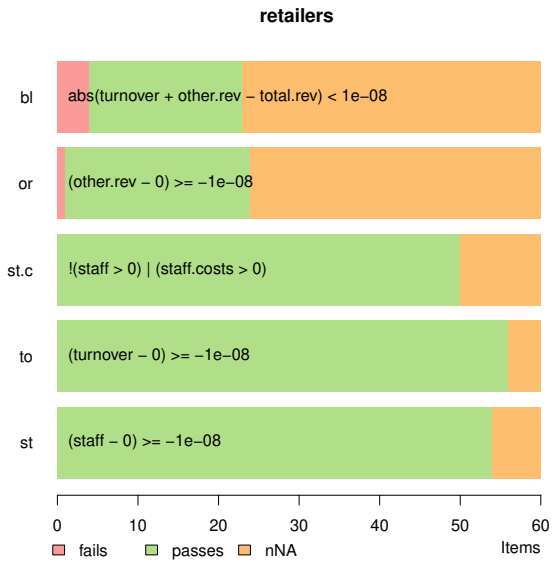

Figure 2: Barplot of validation results.

```
R> plot(check[1:5], main='retailers')
```

Until now we used `validator` to first create a rule set and then `confront`, to confront data with rules. In the following sections we shall sometimes use `check_that` which combines these steps.

```
R> ## this
R> out <- check_that(retailers, staff >= 0, other.rev >= 0)
R> ## is equivalent to
R> out <- confront(retailers, validator(staff>=0, other.rev >= 0))
```

3.1. Domain specific languages

According to Fowler (2010), a domain specific language (DSL) is "*a computer programming language of limited expressiveness focused on a particular domain*". DSLs are commonly divided into *standalone* DSLs that can be run within a specific software and *embedded* DSLs that are developed as a subset of expressions of an existing language (Gibbons 2013). Examples of *standalone* DSLs are the BLAISE scripting language for survey questionnaire design (Statistics Netherlands 2018) and the relatively new VTL language for data validation and transformation (VTL task force and SDMX technical working group 2018). A well-known example of a DSL in R is the set of expressions accepted by the `subset` function. Recall that if `d` is a data frame then `subset(d, <expression>)` returns the records in `d` satisfying `<expression>`. The DSL for `subset` is a set of R expressions that evaluate to a logical vector suitable for filtering records from `d`.

There are several reasons why R (R Core Team 2019) is a good choice for hosting an embedded DSL. First, it offers all the necessary technical tools out of the box: R expressions can be manipulated and investigated just like any other variable. In **validate** this is used to check

if an expression passed to `validator` can be interpreted as a validation rule. R also offers functions such as `eval`. This function accepts an unevaluated R expression and a data set and evaluates the expression using data in the data set. In **validate** such a construction is used to execute validation rules for a data set. Second, R and its packages offer an immense amount of well-defined and well-tested (statistical) functionality. A DSL that is built in R inherits all this functionality for free.

3.2. A domain specific language for data validation

The DSL implemented in **validate** consists of R expressions that evaluate to '`logical`', plus some special expressions that do not evaluate to '`logical`' but that make certain common data validation tasks easier. This section is devoted to the expressions that evaluate to a '`logical`'. Other syntax elements ('syntactic sugar') are discussed in the next section.

To ensure that an expression stored in a '`validator`' object results in a '`logical`', it is inspected when the user defines it. When a user passes an expression that is not part of our DSL, it is ignored with a warning.

```
R> rules <- validator(x > 0, mean(x))

Warning message:
  Invalid syntax detected, the following expressions have been ignored:
[002] mean(x)
```

Here, the first expression is recognized as part of the DSL because the final operation to be evaluated (the `>` comparison operator) must yield a '`logical`'. This is not true for the second expression which just computes a mean. For each expression defined by the user, the `validator` function compares the final operation with a list of allowed final operations. An overview of these operations is given in the table below. They can be separated into unary and binary operators or functions. Let us clarify their use with some examples.

Table 1: Basic operations and functions that define a validation rule.

Type	R function or operator
Unary operators or functions	`!, all, any, grepl`, any `is.` function.
Binary operators or functions	`<, <=, ==, identical, !=, >=, >, %in%, if, ~`
Helper functions	`is_unique`, `all_unique`, `is_complete`, `all_complete`

Example 1: checking for duplicates. The retailers dataset has an `id` variable. Requiering it to be unique can be expressed as follows.

```
R> checks <- check_that(retailers, !any(duplicated(id)))
R> all(checks)

[1] TRUE
```

Or, using one of the helper functions

```
R> all( check_that(retailers, all_unique(id)) )
```

```
[1] TRUE
```

The advantage of `all_unique` is that it accepts a comma-separated list of variable names so it is also possible to check for uniqueness of combinations of values.

Example 2: checking variable type. Any function starting with '`is.`' is accepted as a data validation function. Below we demand that `turnover` is a numeric variable and `size` is a categorical variable (`factor`).

```
R> checks <- check_that(retailers, is.numeric(turnover), is.factor(size))
R> all(checks)
```

```
[1] TRUE
```

The use of comparison operators (<,<=,...) was demonstrated in Section 2. The next example demonstrates how more complex validation rules can be built up by reusing functionality already present in R.

Example 3: correlation between variables. Suppose we wish to check whether the correlation between two variables `height` and `weight` in the `women` dataset is larger than 0.5.

```
R> correlation_check <- validator(cor(height,weight) > 0.5)
R> summary( confront(women, correlation_check) )
```

```
  name items passes fails nNA error warning                expression
1   V1     1      1     0   0 FALSE   FALSE cor(height, weight) > 0.5
```

The unary operators and functions also include R's `grepl` function which accepts a regular expression and a '`character`' vector, and returns `TRUE` for each element of the '`character`' vector that matches the regular expression.

Example 4: pattern matching a text variable. The following rule tests if the `size` variable always consists of a string starting with `"sc"`, followed by a number.

```
R> checks <- check_that(retailers, grepl("^sc[0-9]$", size))
R> all(checks)
```

```
[1] TRUE
```

Example 5: testing a variable against a code list. Since `size` is a categorical variable, we can use the binary `%in%` operator to check the values against an explicit list of allowed values.

```
R> checks <- check_that(retailers, size %in% c("sc0","sc1","sc2","sc3"))
R> all(checks)

[1] TRUE
```

The binary operations '`if`' and '`~`' (tilde) have special interpretations that need to be explained in more detail. The `if` function is interpreted as a logical implication that results in a truth value. In propositional logic, it is usually denoted with an arrow as in $P \Rightarrow Q$ (P implies Q), where P and Q are propositions that can be either TRUE or FALSE. The truth table for this operation is given below.

P	Q	$P \Rightarrow Q$
FALSE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

It is a classic result from propositional logic that $P \Rightarrow Q$ is equal to $\neg P \vee Q$ [(not P) or Q], in the sense that both expressions have the same truth table. This result is exploited by **validate** by translating rules of the form `if(P) Q` to `!(P) | Q`. This translation allows R to execute the test in a vectorized manner, resulting in a significant speed gain.

Example 6: conditional rules. When a company employs staff, there should be associated staff costs.

```
R> check <- check_that(retailers, if (staff > 0) staff.costs > 0)
R> all(check, na.rm=TRUE)

[1] TRUE

R> summary(check)

  name items passes fails nNA error warning
1   V1    60     50     0  10 FALSE   FALSE
                         expression
1 !(staff > 0) | (staff.costs > 0)
```

The summary shows that indeed the original expression has been rewritten. Comparing the rule with the truth table, we see that according to this rule it is acceptable for a company to employ no staff and have no staff costs; to have staff costs without employing staff; and to employ staff and have staff costs. It is not acceptable to employ staff and not have any staff costs. If it is reasonable to expect staff whenever there are staff costs, a second rule of the form `if ( staff.costs > 0 ) staff > 0` must be defined.

The last operation we discuss here is the `~` (tilde) operator. This is used to indicate so-called functional dependencies. A functional dependency expresses a relation between multiple records in relational data. For example given two variables `street` and `postal_code` we could specify that 'if two records have the same street name, they must have the same postal code'. In **validate**, this would be expressed as

```
R> rules <- validator( street ~ postal_code )
```

If we also have a variable called `city`, we could state that if two records have the same city and street, they must have the same postal code. This would be expressed as

```
R> rules <- validator( city + street ~ postal_code )
```

Similarly, we can express the relation 'if two records have the same postal code, they must have the same values for city and street'.

```
R> rules <- validator( postal_code ~ city + street )
```

Functional dependencies are have been researched extensively in the field of database theory. Some interesting sources include Armstrong (1974); Beeri, Dowd, Fagin, and Statman (1984); Bohannon, Fan, Geerts, Jia, and Kementsietsidis (2007) and references therein.

3.3. Syntactic sugar

The term 'syntactic sugar' refers to elements of a syntax that are not strictly necessary but make common tasks easier. The **validate** DSL includes some syntactic sugar to refer to the dataset as a whole, to store and reuse intermediate results, and to apply the same rule to multiple variables. We discuss these syntax elements with a few examples while an overview is given in Table 2.

Syntactic sugar 1: inspecting metadata. An important class of data validation involves checking metadata, including the dimensions of a data set or the presence of certain variables. In the previous section it was shown how rules can refer to individual variables in a data set, but to test metadata, it is convenient to have access to the data set as a whole. This is implemented using '`.`'.

```
R> rules <- validator(
+   nrow(.) >= 100
+  ,"Species" %in% names(.) )
```

Here, the '`.`' refers directly to the data set (here, the `iris` data frame) passed to `confront`.

Syntactic sugar 2: reuse calculations. The `:=` operator can be used to store intermediate calculation for reuse. In the following rule set we check if the fraction of 'versicolor' is between 0.25 and 0.50. To do so, we first compute the fraction and reuse that to check the bounds.

Table 2: Syntactic sugar

Syntax element	Description
`.`	Refers to the whole dataset, e.g., `nrow(.) > 10`
`:=`	Store intermediate result, e.g., `med := median(x)`
`var_group()`	Create a list of variables to be reused in rules, e.g., `G := var_group(X, Y, Z)`

```
R> rules <- validator(
+   fraction := mean(Species == "versicolor")
+   , vc_upr =  fraction >= 0.25
+   , vc_lwr =  fraction <= 0.50)
```

The `:=` operator can be used as if it were the standard assignment operator in `R`. The main difference is that the right-hand-side of `A := B` is not evaluated immediately. Rather, in any rule after the definition of `A := B` the use of `A` is replaced by `B`. We can see this by confronting the rule set from the example with the `iris` data set and inspecting the expressions that have been evaluated.

```
R> as.data.frame( confront(iris, rules) )["expression"]
```

```
                             expression
1 mean(Species == "versicolor") >= 0.25
2  mean(Species == "versicolor") <= 0.5
```

Syntactic sugar 3: referring to external data. For some data validation purposes it is necessary or convenient to have access to external reference data. For example, we may want to retrieve the list of valid `Species` values from an external source. This can be done as follows.

```
R> codelist <- data.frame(
+      validSpecies =  c("versicolor", "virginica","setosa")
+    , stringsAsFactors=FALSE)
R> rules <- validator(Species %in% ref$validSpecies)
R> summary(confront(iris, rules, ref=codelist))[1:5]
```

```
  name items passes fails nNA
1   V1   150    150     0   0
```

Note that the reference data is passed via the `ref` argument of `confront`. There are several ways to pass reference data in this way, including data frames, lists, or environments.

Syntactic sugar 4: same rules, multiple variables. When many variables need to satisfy a similar set of rules, rule definition can become cumbersome. A common example occurs when many variables are measured on the same scale such as $[0, 1]$ or $\{1, 2, 3, 4, 5\}$, and

all these ranges need to be checked. Such cases are facilitated with the concept of 'variable groups'. This is essentially a list of variable names that can be reused inside a rule.

Explicitly, the following set of rules (defining that all of `x`, `y`, and `z` are between 0 and 1)

```
R> rules <- validator(x >= 0, y >= 0, z >= 0, x <= 1, y <= 1, z <= 1)
```

is equivalent to

```
R> rules <- validator( G := var_group(x, y, z), G >= 0, G <= 1 )
```

The rule sets are expanded when confronted with data. If multiple variable groups are used in a single validation rule, the expansion is based on the Cartesian product of the variables defined in the variable groups.

3.4. Validator objects

Every data validation rule expresses an assumption about a data set. When the number of such validation rules grows, it becomes desirable to be able to manage them with CRUD (create, read, update, delete) operations, to describe, analyze, and manipulate them. There are also benefits in having access to rule metadata, such as a rule name or description. Such information can then be included in automatically generated data quality reports or dashboards. The purpose of '`validator`' objects is to import, manipulate, and export validation rules and their metadata. In this section we demonstrate how to manipulate and investigate validation rules and their metadata, in the next section we focus on import and export of rule sets.

Objects of class '`validator`' behave much like an R `list`. They can be subsetted with the square bracket operator, and a single element can be extracted using double square brackets.

```
R> rules <- validator(minht = height >= 40, maxht = height <= 95)
R> # Subsetting returns a 'validator' object.
R> rules[2]

Object of class 'validator' with 1 elements:
 maxht: height <= 95
Rules are evaluated using locally defined options
```

A single element is an object of class '`rule`'. It holds an expression and some metadata.

```
R> rules[[1]]

Object of class rule.
 expr       : height >= 40
 name       : minht
 label      :
 description:
 origin     : command-line
 created    : 2019-12-20 11:53:49
 meta       : language<chr>, severity<chr>
```

Table 3: An overview of functions to investigate or manipulate 'validator' objects.

Function	description
summary	Summarize properties
plot	Matrix plot showing what variables occur in each rule
'['	Create new 'validator' with subset of rules
'[['	Extract a single rule with its metadata
length	Number of rules
as.data.frame	Store rules (as character) and metadata in data frame
variables	Which variables occur in a 'validator'?
names, names<-	get, set rule names
label, label<-	get, set rule labels
description, description<-	get, set rule description
origin, origin<-	get, set rule origin
created, created<-	get, set rule timestamp
meta, meta<-	get, set generic metadata
+	combine two validator instances

Here, label and description are short and long descriptions of the rule respectively. These have to be defined by the user. The fields origin and created are automatically filled when the rules are defined. The meta field allows the user to add extra metadata elements.

Metadata can be extracted and defined in ways that should be familiar to R users. For example, setting labels is done as follows.

```
R> label(rules) <- c("least height", "largest height")
R> rules

Object of class 'validator' with 2 elements:
 minht [least height]  : height >= 40
 maxht [largest height]: height <= 95
```

For each field there is a get and set function that works similar to the names function of R. The meta field can be manipulated with the meta function, which works similar to R's attr function. An overview of functions for manipulating 'validator' instances is given in Table 3.

For larger rule sets it is interesting to query which variables are covered and by which rules. This is done with variables, which returns an overview of variable coverage by the rule set.

```
R> variables(rules)

[1] "height"

R> variables(rules, as='matrix')

       variable
rule    height
  minht   TRUE
  maxht   TRUE
```

Table 4: Import and export of validation rules.

Function	Description
`validator(...)`	Define rules on the command-line
`validator(.file=)`	Read rules from file
`validator(.data=)`	Read rules from data frame
`export_yaml`	Export to YAML format
`as.data.frame`	Transform '`validator`' to '`data.frame`' (lossy)

One use is to automatically detect whether all variables in a data set are covered in by a rule set.

```
R> all(names(women) %in% variables(rules))

[1] FALSE

R> names(women)[!names(women) %in% variables(rules)]

[1] "weight"
```

There are other interesting questions to be asked about rule sets, for example whether there are any redundancies or even contradictions. For this more advanced type of functionality the reader is referred to the complementary **validatetools** package (de Jonge and van der Loo 2019).

3.5. Import and export of rule sets

Validation rules can be defined in several unstructured and structured formats. The unstructured formats include definition on the command line and definition via an unstructured text file (source code). Structured formats include data frames or YAML files (yaml.org 2015) where each rule can be endowed with metadata. Here we discuss some advantages and disadvantages of the methods for storing and exporting data validation rule sets (see also Table 4).

A rule set and its metadata can be exported to data frame using `as.data.frame` as usual in R. Reading from data frame is done using `validator(.data=)`. Import from and export to data frame is especially useful in situations where rule sets are stored in a centralized repository that is implemented as a data base. A disadvantage of using data frames is that metadata that pertains to the rule set as a whole will be lost when exporting to data frame. In particular, option settings particular to a rule set are not retained when exporting to data frame. For extensive rule sets an approach using structured or unstructured text files, or a combination thereof may better suit the needs.

Text files offer the most flexible way of defining rule sets in **validate**. Indeed, there are benefits in treating a rule set as a type of source code that can be managed under version control and provided with comments. The **validate** package also supports advanced features such as file inclusion, where one rule file can refer to another rule file. We will demonstrate this in the following example, which also shows how structured and unstructured formats may be used together.

For this example we imagine the following use case. We have a questionnaire that consists of two parts: a general part that is sent to all participants of a study, and a specific part of which the contents depends on the participant type. Here, the general part is related to retailers, while the specific part is related to the subdomain of supermarkets. The idea is that we maintain two files: one with general rules, that is to be reused accross different subdomains and one with specific rules for supermarkets.

Figure 3 shows the general rules in YAML format. At the top, there is a section with general options pertaining to the rule set as a whole. Next, all the rules and their metadata are defined one by one. The rules are stored in a file named `general_rules.yml`. The specific rules are stored in a simpler text file called `rules.txt` with the following content.

```
---
include:
  - general_rules.yml
---

# a reasonable profit
profit/total.rev <= 0.6

# We expect that the supermarket sector
# is profitable on average
mean(profit) >= 1
```

At the top there is an optional block, demarcated with dashes, where we can define properties for the whole rule set or as in this case indicate that the file `general_rules.yml` must be read before the current file. Both files can now be read as follows.

```
R> rules <- validator(.file="rules.txt")
R> rules

Object of class 'validator' with 5 elements:
 G1 [nonnegative staff] : staff >= 0
 G2 [nonnegative income]: turnover >= 0
 G3 [Balance check]     : profit + total.costs == total.rev
 V1                     : profit/total.rev <= 0.6
 V2                     : mean(profit) >= 1
```

A user is able to trace the origins of each rule interactively since the '`validator`' remembers where each rule came from.

```
R> origin(rules)

                  G1                    G2                    G3
"./general_rules.yml" "./general_rules.yml" "./general_rules.yml"
                  V1                    V2
          "rules.txt"           "rules.txt"
```

```
---
options:
  raise: none
  lin.eq.eps: 1.0e-08
  lin.ineq.eps: 1.0e-08
---
rules:
- expr: staff >= 0
  name: 'G1'
  label: 'nonnegative staff'
  description: |
    'Staff numbers cannot be negative'
  created: 2018-06-05 14:44:06
  origin:
  meta: []
- expr: turnover >= 0
  name: 'G2'
  label: 'nonnegative income'
  description: |
    'Income cannot be negative (unlike in the
     definition of the tax office)'
  created: 2018-06-05 14:44:06
  origin:
  meta: []
- expr: profit + total.costs == total.rev
  name: 'G3'
  label: 'Balance check'
  description: |
    'Economic profit is defined as the
     total revenue diminished with the
     total costs.'
  created: 2018-06-05 14:44:06
  origin:
  meta: []
```

Figure 3: Contents of the file `general_rules.yml`. Several global options and metadata fields have been filled in.

Finally we point out that file inclusion works recursively so included files can include other files. Mistakes such as cyclic inclusion are detected and reported when they occur. A complete description can be found in a dedicated vignette that is included with the package.

3.6. Confronting data with rules

At the beginning of Section 3 we have seen how data can be checked against a rule set using `confront`. It was also highlighted that several default choices were made, for instance how

Table 5: Options for confronting data with rules. Default values in brackets.

Option	Values	Description
`na.value`	`[NA], TRUE, FALSE`	Value when rule results in `NA`
`raise`	`["none"], "error", "all"`	What exceptions to raise
`lin.eq.eps`	`[1e-8]`; positive number	Tolerance for checking equalities
`lin.ineq.eps`	`[1e-8]`; positive number	Tolerance for checking inequalities

machine rounding and exceptions (errors, warnings) are handled, and the fact that missing values in data lead to missing data validation results. In this section we demonstrate how those options can be manipulated both globally and locally.

There are three places where options can be set, with three different ranges of influence. The first place is while calling `confront`. In this case the options only pertain to the current call. For example, we can count missing validation results as `FALSE` (fail) as follows (we suppress the last column of the summary for brevity)

```
R> data("retailers")
R> rules <- validator(turnover >= 0, turnover + other.rev == total.rev)
R> summary( confront(retailers, rules) )[-8]

  name items passes fails nNA error warning
1   V1    60     56     0   4 FALSE   FALSE
2   V2    60     19     4  37 FALSE   FALSE

R> summary( confront(retailers, rules, na.value=FALSE) )[-8]

  name items passes fails nNA error warning
1   V1    60     56     4   0 FALSE   FALSE
2   V2    60     19    41   0 FALSE   FALSE
```

The second place to control options is by setting options for a specific '`validator`' instance. This is done with the `voptions` function. For example, if we know that all our variables are integer we can ignore the possibility of spurious fails caused by machine rounding in linear equalities and linear inequalities (we ignore columns 6 and 7 in the output for brevity).

```
R> voptions(rules, lin.eq.eps=0, lin.ineq.eps=0)
R> rules

Object of class 'validator' with 2 elements:
 V1: turnover >= 0
 V2: turnover + other.rev == total.rev
Rules are evaluated using locally defined options

R> summary( confront(retailers, rules) )[-(6:7)]

  name items passes fails nNA                       expression
1   V1    60     56     0   4                     turnover >= 0
2   V2    60     19     4  37 turnover + other.rev == total.rev
```

The expressions in the `expression` column are now exactly equal to the user-defined rules and there is no tolerance that allows for some machine rounding.

The third and final place where options can be set is globally. This is done by calling `voptions` with `option = value` pairs. This will affect all `validator` objects, except those that already have their options adjusted. Recall that by default, errors are caught and stored.

```
R> data("retailers")
R> out <- check_that(retailers, employees >= 0) # the error is caught.
```

However, we can set `raise="error"` so execution stops when an error is raised.

```
R> voptions(raise = "error") # set global option.
R> out <- check_that(retailers, employees >= 0)
Error in fun(...) : object 'employees' not found
```

Raising errors or warnings immediately rather then just registering them can be useful for example when developing or debugging a large rule set.

To summarize, any option listed in Table 5 can be set at three levels: globally, for individual '`validator`' objects, and during a single call to '`confront`'.

3.7. Validation objects and analyzing results

The return value of `confront` is a '`validation`' object. It holds the data validation results and some information on the data validation procedure. This information can be extracted and summarized with the functions listed in Table 6. Below we demonstrate some of them and point out some issues related to the dimensionality of data validation results.

Like '`validator`' objects, '`validation`' objects can be subsetted by rule using single square brackets.

```
R> rules <- validator(other.rev >= 0, turnover >= 0,
+   turnover + other.rev == total.rev)
R> check <- confront(retailers, rules)
R> summary(check[1:2])
```

```
  name items passes fails nNA error warning                expression
1   V1    60     23     1  36 FALSE   FALSE (other.rev - 0) >= -1e-08
2   V2    60     56     0   4 FALSE   FALSE  (turnover - 0) >= -1e-08
```

Using `values` all results can be gathered in an array.

```
R> head(values(check), n=3)
```

```
        V1   V2    V3
[1,]    NA   NA    NA
[2,]    NA TRUE    NA
[3,] FALSE TRUE FALSE
```

Table 6: Investigating 'confrontation' objects.

Function	description
summary	Summarize results per rule
all	Check if all validations result in TRUE
any	Check if any validation resulted in FALSE
as.data.frame	Gather results in a data frame
values	Gather results in a (list of) array(s)
aggregate	Aggregate results by record or by rule
sort	As aggregate, but with sorted results
plot, barplot	Create barplot(s) of results
errors	List of error signals
warnings	List of warning signals
'['	Select subset of confrontations (by rule)
length	Number of rules evaluated.

By aggregating and sorting results by rule or by record, interesting information can be gathered on which records are the 'worst violators', or which rules are violated most often.

```
R> sort(check, by="rule")
```

```
   npass nfail nNA  rel.pass   rel.fail     rel.NA
V3    19     4  37 0.3166667 0.06666667 0.61666667
V1    23     1  36 0.3833333 0.01666667 0.60000000
V2    56     0   4 0.9333333 0.00000000 0.06666667
```

Here, we aggregated results by rule and sorted the totals by increasing number of passes per rule. In this case the balance restriction (rule V3) is passed the least number of times.

One issue that hampers analysis of data validation results is that the outcome of different validation rules may have different dimensions. As an example consider the following rules for the retailers dataset.

```
R> rules <- validator(staff >= 0, turnover >= 0, mean(profit) >= 1)
R> check <- confront(retailers, rules)
```

The first two rules evaluate to sixty results: one for each record in the retailers data frame. The third rule evaluates to a single result for the whole data set. This means that values cannot meaningfully combine all results in a single array. For this reason values works much like R's native sapply function. An array is returned when all results fit in a single array. If not, a list is returned with an array for each subset of results that fit together in an array.

```
R> c( class(values(check[1:2])), class(values(check)) )
```

```
[1] "matrix" "list"
```

The default behavior can be controlled with a simplify argument. It has default value TRUE, again similar to sapply.

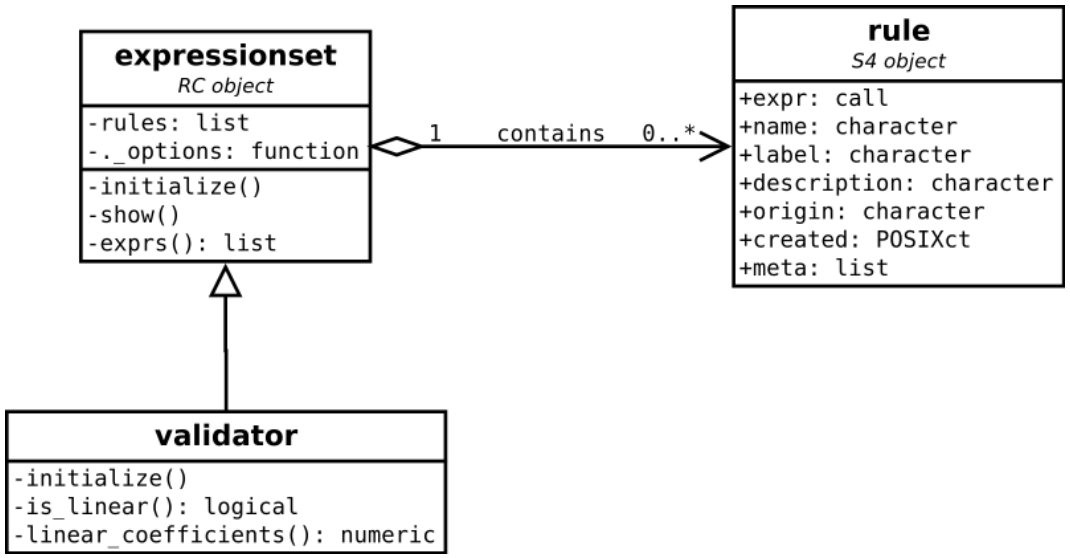

Figure 4: UML diagram demonstrating part of the '`validator`' object model. See Rumbaugh *et al.* (2004) for a description of UML.

4. Object model and implementation

The **validate** package is designed with extensibility in mind. The main objects and methods are based on the `S4` system (see e.g., Chambers (2016)) since this is the most flexible object system currently available in base R. In particular, the fact that `S4` supports multiple dispatch was an important driver to choose it over `S3`. This allows us to easily add `confront` methods for different types of rules or different data types.

Figures 4 and 5 give a high-level overview of the object model used in **validate** in UML diagrams. The central object for storing expressions is a (not user-visible) class called '`expressionset`'. An expressionset contains an option settings manager and a list of '`rule`' objects. A '`rule`' is an R expression endowed with with some standard metadata. The object type that users see and manipulate directly is called '`validator`'. This is a specialization of '`expressionset`' with some methods that only apply to data validation rules. For example, the constructor of '`validator`' checks whether the imported rules fall within the Domain Specific Language described in Subsection 3.2. The constructor of '`expressionset`' does not perform such checks. There are also a few methods for extracting properties of validation rules. These are currently not publicly documented and should be avoided by users. The reason they are implemented as methods of `validator` objects is that it allows us to reuse them in packages depending on **validate** (for example **validatetools**).

The user-visible return value of `confrontation` is an object of class '`validation`' (Figure 5). This is again a specialization of a more general object called '`confrontation`'. A '`confrontation`' object contains the call that created it, the evaluated expressions and their results. Currently the specialization in '`validation`' is limited to the `show` method.

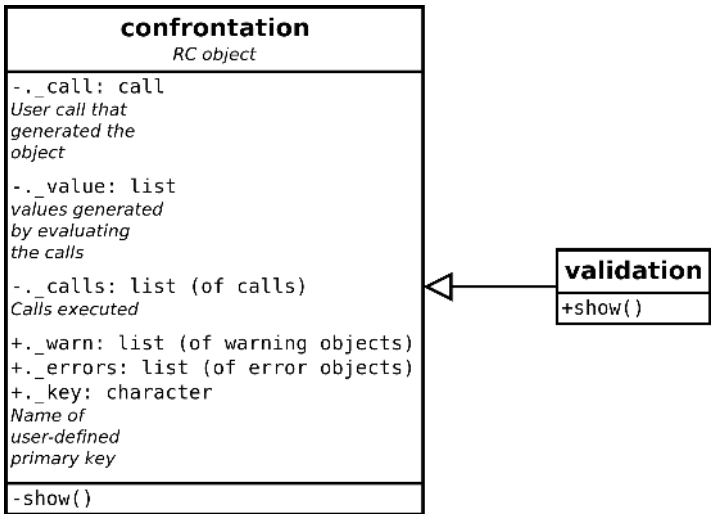

Figure 5: UML diagram showing contents of the '`confrontation`' and '`validation`' object classes.

5. Demo: cleaning up a small data set

In this Section we demonstrate how **validate** can be integrated in a data cleaning workflow. We also introduce two new functions called `cells` and `compare`, that allow for comparing two or more versions of the same dataset.

The purpose in this example is to define a set of rules for the `retailers` data set and to process it step by step until all fields are completed and all rules are satisfied. Moreover, we wish to monitor changes in the quality of the dataset as it gets processed. To this end we use the following R packages (to be discussed and referenced below).

```
R> library("validate")
R> library("dcmodify")
R> library("errorlocate")
R> library("simputation")
R> library("rspa")
```

The task is to clean up measured attributes of the `retailers` dataset. This excludes the first two columns (representing size class and sample inclusion probability) so we first create a working copy for the example.

```
R> data("retailers")
R> dat_raw <- retailers[-(1:2)]
```

We have defined a set of 18 validation rules for this data set in a text file that is available and documented in the supplementary materials. The rules include account balances, non-negativity constraints and sanity constraints on ratios between variables. For brevity only a few examples are displayed below. We relax the condition on equalities somewhat to accommodate a method that will be introduced further on.

```
R> rules <- validator(.file="rules.R")
R> voptions(rules, lin.eq.eps=0.01)
R> rules[c(1,4,11)] # show a few examples

Object of class 'validator' with 3 elements:
 V01: staff >= 0
 V04: turnover + other.rev == total.rev
 V11: profit <= 0.6 * turnover
Rules are evaluated using locally defined options
```

The `retailers` set consists of staff numbers and financial variables of 60 supermarkets. Respondents have been asked to submit financial amounts in multiples of EUR 1000, but in some cases they are submitted in Euro. For example, this is obviously the case in record 15.

```
R> dat_raw[15,c('staff','turnover','staff.costs')]

   staff turnover staff.costs
15     3    80000       40000
```

Taking this record at face value would be equal to believing that a retailer with a staff of 3 generates 80 million Euros turnover and pays 40 million to three employees. If we assume that `staff` is a reliable variable, such errors can often be detected and repaired using rule-based processing. For example, `IF` *the ratio between* `turnover` *and* `staff` *exceeds 500* `THEN` *divide* `turnover` *by 1000*. The **dcmodify** package (van der Loo and de Jonge 2018) allows users to formulate such rules and apply them to data, much like the way validation rules are defined and applied in the **validate** package. We defined 12 of such rules, a few of which are printed here for brevity. The complete set is available in the supplementary materials.

```
R> modifiers <- dcmodify::modifier(.file="modifiers.R")
R> modifiers[c(1,4,11)]

Object of class modifier with 3 elements:
M01:
  if (other.rev >= 500 * vat) other.rev <- other.rev/1000

M04:
  if (abs(profit) >= 100 * vat) profit <- profit/1000

M11:
  if (staff.costs >= 500 * staff) staff.costs <- staff.costs/1000
```

The first and fourth statement compare a financial variable with the value of the Value-Added-Tax amount (which is considered reliable). Just like in **validate**, the if-statement is executed for each record of the data set. In cases where the condition evaluates to `TRUE` the variable is to be replaced with the same value divided by one thousand. We can apply the rules to the dataset as follows.

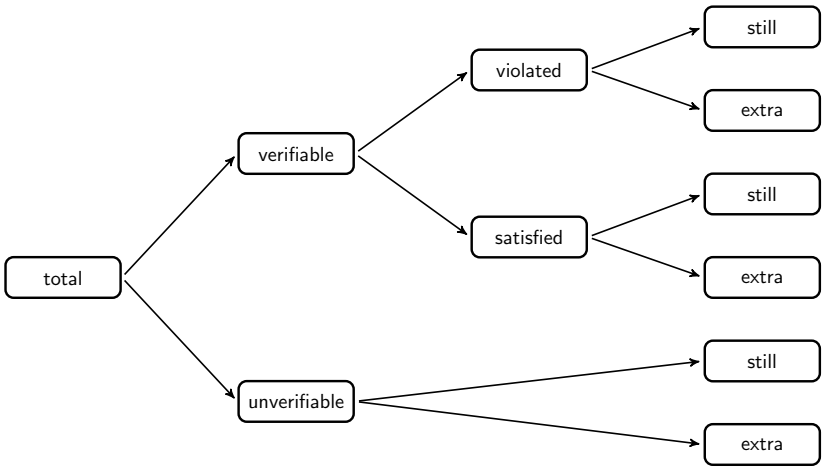

Figure 6: Partitioning the result of comparing validation results for two versions of a data set. The number of cells in a parent node is the sum of the number of cells in its child nodes (reproduced from van der Loo and de Jonge (2018)).

```
R> dat_mod <- dcmodify::modify(dat_raw, modifiers)
```

We are now interested in the effect of this step by comparing the validation results before and after executing these data modifying rules. Figure 6 shows how we can decompose the validation results. The total number of validation results can be separated into those that are verifiable (i.e., yielding `TRUE` or `FALSE`) and those that are not (yielding `NA`). The values that are verifiable can be further separated into those that are `TRUE` or `FALSE`, and for each one we can check whether they had a similar value in the previous version of the data set (still) or if it has changed (extra). Similarly, we can check for unverifiable validations in the current version whether they were also unverifiable in the previous version (still) or not (extra).

The **validate** package exports a function that computes the number of transitions for each instance shown in Figure 6. The `compare` function accepts a '`validator`' and an arbitrary number of data sets that can be compared sequentially or against the first data set.

```
R> compare(rules, raw = dat_raw, modified=dat_mod)
```

```
Object of class validatorComparison:

   compare(x = rules, raw = dat_raw, modified = dat_mod)

                    Version
Status                raw modified
  validations        1080     1080
  verifiable          789      789
  unverifiable        291      291
  still_unverifiable  291      291
  new_unverifiable      0        0
  satisfied           746      755
  still_satisfied     746      741
```

```
new_satisfied         0       14
violated             43       34
still_violated       43       29
new_violated          0        5
```

We see that out of 1080 validations (= 60 records times 18 rules), 746 are satisfied in the raw data while 755 are satisfied in the modified data. This seems an improvement, but a closer look reveals that actually 14 cases were resolved while 5 new violations were introduced. So the assumptions laid down in the modifying rules may have been too strong in some cases. Or, it is possible that one or more records had all financial variables a factor of thousand too high (thus consistent in terms of balance checks) but not all variables were altered, hence introducing inconsistency. For this example we will not dive deeper into this but instead move on to the next cleaning step.

We now clean up the data in three steps. First, knowing what rules each record violates, we need to figure out which fields to alter so that all rules can ultimately be satisfied. This procedure is called 'error localization'. Next, we impute the fields that are empty or deemed erroneous during the error localization procedure. Since our imputation method will not take validation rules into account, imputation will generally not yield a dataset that satisfies all validation rules[2]. So in a last step we adjust the imputed values minimally (in a sense to be defined) such that all constraints are satisfied.

For the first step we apply the principle of Fellegi and Holt (1976). In this approach, it is assumed that errors are both rare and independently distributed over the variables. The idea is to find for each record, the minimal (weighted) number of fields that can be emptied such that imputation consistent with the rules is possible in principle. The chief difficulty is that variables typically occur in more than one rule, so changing a variable to fix one rule, may cause violation of another rule. There are several numerical approaches to solving this (de Waal *et al.* 2011). The **errorlocate** package implements a mixed integer programming (MIP) approach that both localizes and removes erroneous values. In the following step we set higher weights for `staff` and `vat`, so **errorlocate** will not replace them unless it cannot be avoided. This means we judge variables with heigher weight to be of higher quality.

```
R> weights = setNames(rep(1, ncol(dat_raw)), names(dat_raw))
R> weights[c("staff","vat")] <- 10
R> dat_el <- errorlocate::replace_errors(dat_mod, rules, weight = weights)
R> colSums( summary(confront(dat_el, rules))[3:5] )

passes  fails    nNA
   677      0    403
```

The output of `confront` confirms that there are no more violated rules. Of course many cannot be checked anymore because a number of cells either were empty or have been emptied by the error localization routine.

For the imputation step we use the **simputation** package (van der Loo 2017) to estimate the missing values using classification and regression trees (CART, Breiman (1984)).

[2]For some work on imputing data under restrictions see de Waal (2017); Vink (2015) and Tempelman (2007) and references therein.

```
R> dat_imp <- simputation::impute_cart(dat_el, . ~ .)
R> colSums(summary(confront(dat_imp, rules))[3:5])

passes  fails    nNA
  1003     77      0
```

As expected, we see many fails but no more missing values.

Third and finaly, we adjust the imputed values to fit the constraints. For this, we use the **rspa** package (van der Loo 2019) which implements the successive projection algorithm. The idea is to minimize the weighted Euclidean distance between the current imputed values and adjusted values while satisfying all restrictions (this algorithm is limited to systems of linear equalities and inequalities). To account for differences of scale between the variables, we use the reciprocal of the current values as weight. It was shown by Zhang and Pannekoek (2015) that this preserves the ratio between the original values to first order.

```
R> # Compute weights for weighted Euclidean distance
R> W <- t(t(dat_imp)) # convert to numeric matrix
R> W <- 1/(abs(W)+1)  # compute  weights
R> W <- W/rowSums(W)  # normalize by row
R> # adjust the imputed values.
R> dat_adj <- rspa::match_restrictions(dat_imp, rules
+    , adjust = is.na(dat_el), weight=W, maxiter=1e4)
```

We can now verify that all rules are satisfied.

```
R> all(confront(dat_adj, rules))

[1] TRUE
```

Since we stored all intermediate results, we can visualize the effect of each intermediate step. We use `compare` and `cells` to follow the changes in the data set as it is treated step by step. The `cells` function is also part of `validate`. It measures the number of changes by decomposing the number of cells in a data set into those that are available or not, and separating those again into which are still available, have been removed, or are still missing when compared to a previous version of the data set. (See Figure 7).

```
R> plot(compare(rules, raw= dat_raw, modified = dat_mod
+   , errors_located = dat_el, imputed = dat_imp
+   , adjusted = dat_adj, how="sequential"))

R> plot(cells(raw= dat_raw, modified = dat_mod
+   , errors_located = dat_el, imputed = dat_imp
+   , adjusted = dat_adj, compare="sequential"))
```

These `plot` methods can provide some quick first insight into the effect that each step has on the dataset, thus providing potentially useful information for improving the process. For example, in the left panel we see that across the procedure, the number of violations (thick red

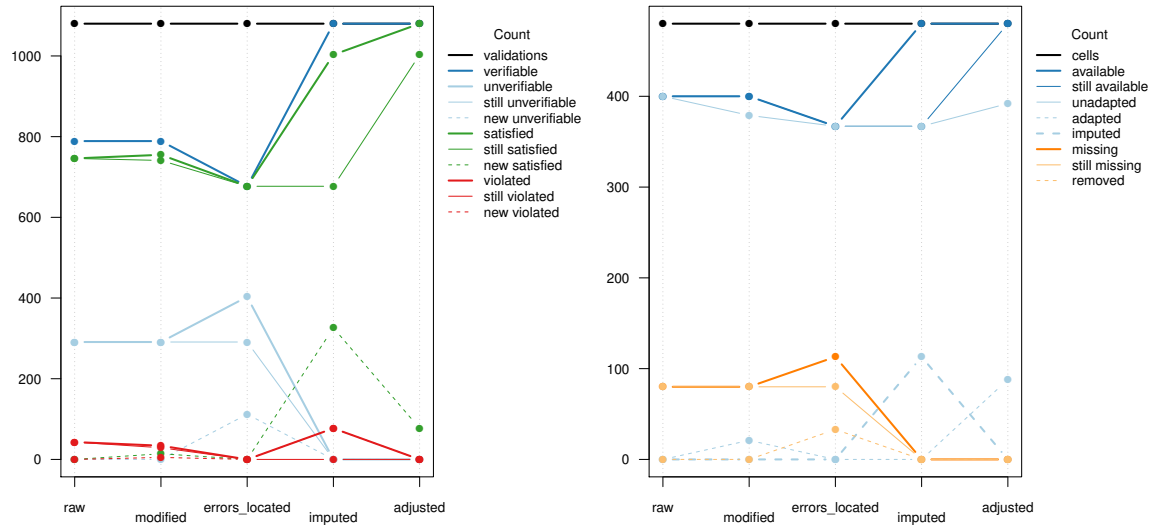

Figure 7: Left: following the performance of a dataset with respect to rule satisfaction as it gets processed step by step. Right: follow changes in cell values as a data set gets processed step by step.

line) first decreases to zero after error localization, then errors get introduced by imputing without regard of the restrictions, and finally all constraints are satisfied by adjusting the imputed values. In the right panel we see that the number of missings (thick yellow line) increases at error localization, reduces to zero after imputation (everything can be imputed) and stays zero afterwards. Such displays can help when assessing the performance of a data cleaning process, for example by switching imputation methods or altering the modifying rules. For cases where there is no natural ordering in a sequence of data sets, the `barplot` method has also been overloaded for objects of classes

Summarizing, we see that **validate** plays an integral role in the data cleaning procedure demonstrated here. Not only is it used to define validation rules and to monitor data quality as a function of processing step, it also provides control parameters for two crucial data cleaning steps: error localization and adjusing numerical values.

6. Summary and outlook

The **validate** package demonstrated in this paper serves as a data validation infrastructure in several aspects. First, it is grounded in a strict definition of the concept of data validation functions. This definition is supported in the form of an R-embedded domain specific language that allows users to express restrictions on data (validation rules) with great flexibility. Second, the package treats validation rules as first class citizens, implementing full CRUD (create, read, update, delete) functionality on validation rule sets along with rudimentary rule inspection functionality. Third, a flexible file-based interface makes rule sets and their metadata elements maintainable as source code, complete with recursive file inclusion functionality. The ability to reading rules and their metadata from a data frame also allows storage and

retrieval from relational databases. Fourth, and finally, the package is implemented using a fairly simple, extendable object-oriented framework. This allows extensions in the direction of different data sources and different rule set implementations.

Future work on this package will likely include improved support for subsetting validation rule sets based on metadata elements, support for user-defined extensions of the validation DSL, more helper functions for complex validation tasks, and improved support for summarization and reporting of data validation results.

References

Armstrong W (1974). "Dependency Structures of Data Base Relationships." In *IFIP Congress*, pp. 580–583. IFIP.

Bache SM, Wickham H (2014). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5, URL https://CRAN.R-project.org/package=magrittr.

Beeri C, Dowd M, Fagin R, Statman R (1984). "On the Structure of Armstrong Relations for Functional Dependencies." *Journal of the ACM (JACM)*, **31**(1), 30–46.

Bohannon P, Fan W, Geerts F, Jia X, Kementsietsidis A (2007). "Conditional Functional Dependencies for Data Cleaning." In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 746–755. IEEE.

Breiman L (1984). *Classification and Regression Trees*. Taylor & Francis Group, New York.

Chambers JM (2016). *Extending R*. The R Series. Chapman & Hall.

de Jonge E, van der Loo M (2018). *editrules: Parsing, Applying, and Manipulating Data Cleaning Rules*. R package version 2.9.3, URL https://CRAN.R-project.org/package=editrules.

de Jonge E, van der Loo M (2019). *validatetools: Checking and Simplifying Validation Rule Sets*. R package version 0.4.6, URL https://CRAN.R-project.org/package=validatetools.

de Waal T (2017). "Imputation Methods Satisfying Constraints." In *Work Session on Statistical Data Editing*, Working Paper 5. United Nations Economic Commission for Europe.

de Waal T, Pannekoek J, Scholtus S (2011). *Handbook of Statistical Data Editing and Imputation*, volume 563. John Wiley & Sons.

Di Zio M, Fursova N, Gelsema T, Giessing S, Guarnera U, Ptrauskiene J, von Kalben LQ, Scanu M, ten Bosch K, van der Loo M, Walsdorfe K (2015). "Methodology for Data Validation." *Technical Report deliverable No.*, ESSNet on validation. URL https://ec.europa.eu/eurostat/cros/content/methodology-data-validation-handbook-final_en.

Fellegi IP, Holt D (1976). "A Systematic Approach to Automatic Edit and Imputation." *Journal of the American Statistical Association*, **71**(353), 17–35.

Fischetti T (2019). *assertr: Assertive Programming for R Analysis Pipelines*. R package version 2.6, URL https://CRAN.R-project.org/package=assertr.

Fowler M (2010). *Domain-Specific Languages*. Pearson Education.

Gibbons J (2013). "Functional Programming for Domain-Specific Languages." In V Zsók, Z Horváth, L Csató (eds.), *Central European Functional Programming - Summer School on Domain-Specific Languages*, volume 8606 of *LNCS*, pp. 1–28. Springer-Verlag. doi:10.1007/978-3-319-15940-9_1. URL http://link.springer.com/chapter/10.1007/978-3-319-15940-9_1.

Iannone R (2018). *pointblank: Validation of Local and Remote Data Tables*. R package version 0.2.0, URL https://CRAN.R-project.org/package=pointblank.

Petersen AH, Ekstrøm CT (2019). "dataMaid: Your Assistant for Documenting Supervised Data Quality Screening in R." *Journal of Statistical Software*, **90**, 1–38.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rumbaugh J, Jacobson I, Booch G (2004). *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education. ISBN 0321245628.

Statistics Netherlands (2018). *BLAISE*. Version 5.2, URL https://blaise.com.

Tempelman C (2007). *Imputation of Restricted Data*. Ph.D. thesis, University of Groningen.

van der Loo M (2017). *simputation: Simple Imputation*. R package version 0.2.3, URL https://CRAN.R-project.org/package=simputation.

van der Loo M (2019). *rspa: Adapt Numerical Records to Fit (in)Equality Restrictions*. R package version 0.2.4, URL https://CRAN.R-project.org/package=rspa.

van der Loo M, de Jonge E (2018). *dcmodify: Modify Data Using Externally Defined Modification Rules*. R package version 0.1.2, URL https://CRAN.R-project.org/package=dcmodify.

van der Loo M, de Jonge E (2018). *Statistical Data Cleaning with Applications in R*. John Wiley & Sons.

van der Loo M, de Jonge E (2019). *validate: Data Validation Infrastructure*. R package version 0.9.2, URL https://CRAN.R-project.org/package=validate.

van der Loo M, de Jonge E (2020). "Data Validation." *Wiley StatsRef*. Accepted for publication.

Vink G (2015). *Restrictive Imputation of Incomplete Survey Data*. Ph.D. thesis, Utrecht University. https://dspace.library.uu.nl/bitstream/handle/1874/308699/vink.pdf.

VTL task force, SDMX technical working group (2018). *Validation and Transformation Language*. Version 2.0, URL https://sdmx.org.

yamlorg (2015). "YAML Aint Markup Language." http://yaml.org/ (accessed 2015-08-13).

Zhang LC, Pannekoek J (2015). "Optimal Adjustments for Inconsistency in Imputed Data." *Survey methodology*, **41**(1), 127–144.

Affiliation:

Mark van der Loo, Edwin de Jonge
Research and Development
Statistics Netherlands
Henri Faasdreef 312
2492JP Den Haag, The Netherlands
E-mail: m.vanderloo@cbs.nl, e.dejonge@cbs.nl