Manipulation of conditional restrictions and error localization with the editrules package

Mark van der Loo and Edwin de Jonge

Discussion paper (201215)

Statistics Netherlands

The Hague/Heerlen, 2012

Explanation of symbols

.	data not available
*****	provisional figure
******	revised provisional figure (but not definite)
x	publication prohibited (confidential figure)
–	nil
–	(between two figures) inclusive
0 (0.0)	less than half of unit concerned
empty cell	not applicable
2011–2012	2011 to 2012 inclusive
2011/2012	average for 2011 up to and including 2012
2011/’12	crop year, financial year, school year etc. beginning in 2011 and ending in 2012
2009/’10–2011/’12	crop year, financial year, etc. 2009/’10 to 2011/’12 inclusive

Due to rounding, some totals may not correspond with the sum of the separate figures.

Publisher
Statistics Netherlands
Henri Faasdreef 312
2492 JP The Hague

Prepress
Statistics Netherlands
Grafimedia

Cover
Teldesign, Rotterdam

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

Where to order
E-mail: verkoop@cbs.nl
Telefax +31 45 570 62 68

Internet
www.cbs.nl

ISSN: 1572-0314

60083201215 X-10

Manipulation of conditional restrictions and error localization with the editrules package.

Mark van der Loo and Edwin de Jonge

Summary: The quality of statistical statements strongly depends on the quality of the underlying data. Since raw data is often inconsistent or incomplete, data editing may consume a substantial amount of the resources available for statistical analyses. Although R has many features for analyzing data, the functionality for data checking and error localization based on logical restrictions (edit rules, or edits) is currently limited. The editrules package is designed to offer a user-friendly toolbox for edit definition, edit manipulation, data checking, and error localization.

Previous versions of the package could handle either numerical or categorical datasets. In this paper we describe new functionality pertaining to mixed numerical and conditional data as well as functionality pertaining to *conditional restrictions*. Other additions to the package include the ability to read edits from free-form text files and faster error localization under certain conditions. This paper marks the release of editrules package version 2.5.

Keywords: Statistical data editing, error localization, Fellegi-Holt, backtracking, statistical software

Contents

1 Introduction

The quality of raw (survey) data is rarely sufficient to allow for straightforward statistical analyses. Indeed, it has been estimated that National Statistics Offices may devote as much as 40% of their resources to data editing activities (De Waal et al., 2011). For reasons of efficiency and reproducibility it is therefore highly desirable to automate data editing processes.

In the practice of (official) statistics, data records are often required to obey various restrictions, including sum rules, positivity demands or other linear inequalities and categorical restrictions that exclude certain value combinations. Such rules are called *edit rules* or *edits* in short, and a data record is called *inconsistent* when it violates one or more edits. The goal of data editing is to remove inconsistencies while leaving the reported data intact as much as possible.

Data editing is severely complicated by the fact that edit rules are often interrelated: a variable can occur in more than one restriction, and a restriction can contain multiple variables. For example, a variable in an account balance may occur in several sum rules, creating a dependency between those rules. It is common in data editing literature to distinguish between *linear restrictions*, *categorical restrictions* and *conditional restrictions*. Linear restrictions are linear (in)equality restrictions such as range restrictions and sum rules pertaining to numerical data. Categorical restrictions are rules that exclude invalid value combinations from a categorical dataset. We have discussed the implementation of linear rules previously in De Jonge and van der Loo (2012) and the implementation of categorical rules in Van der Loo and De Jonge (2011b). In this paper we discuss new functionality of the R extension package editrules pertaining to the third category: *conditional restrictions*.

Conditional restrictions are a generalization of numerical and categorical rules, and can therefore pertain to both numerical and categorical variables. As an example of such a restriction, consider the following demand on a business survey record:

> If the *legal form* (of a business) is self-employed, the *number of employees* must be zero.

This statement restricts the combined value range of a categorical variable (*legal form*) with that of a numerical variable (*number of employees*). The rule is called a *conditional* restriction since it is written in an if-then form where the first part ("If the legal form is self-employed") states the condition under which the demand in the second part ("the number of employees must be zero") must hold. In the rest of this paper we will refer to the first part as the *predicate* and to the second part as the *consequent*, as this appears to be the common terminology used in computer science.

De Waal (2003) and De Waal and Quere (2003) showed that every edit containing categorical as well as numerical restrictions can be written in such a form. That is, all restrictions on categorical data occur in the predicate and numerical restrictions occur in the consequent expression. The formulation encompasses the formulation of non-conditional numerical or categorical rules by choosing appropriate truth values for predicate or consequent. Additionally, rules connecting two or more numerical restrictions occur in practice as well, as shown by the following example.

> If the *number of employees* is positive, the *amount of salary* payed must be positive.

The latest version of editrules presented here (version 2.5) is now capable of managing conditional edits with a conditional as well as with a numerical predicate.

The rest of this paper is structured as follows. In Section 2 we describe which restrictions can be handled by the package, and introduce the central R object for storing and manipulating conditional restrictions: the editset object. In Section 3 we provide details on the most important edit manipulations that the package provides and Section 4 is devoted to error localization. Examples in R code are given throughout to help new users getting started with this functionality.

2 Mixed data and conditional edits

The term *mixed data* is used in this paper to indicate data containing both numerical and categorical data. We do not distinguish between integer and real numbers here: currently, both are handled as real numbers by editrules. We also do not distinguish between logical and categorical data: under the hood, editrules handles these data types as character, although a user need not consider this when specifying types (see also Van der Loo and De Jonge (2011b)).

2.1 Reading and writing edits

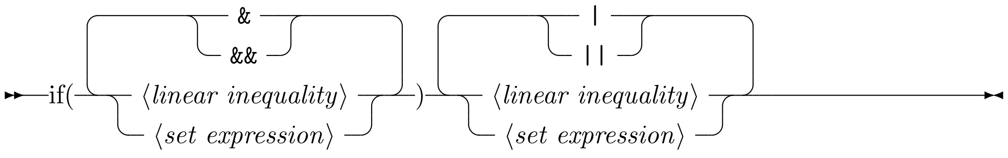
In previos versions of editrules, edits could be read from character vectors where each element contains a rule in textual form. For version 2.5, the parsing capabilities have been extended to allow for rules in free-form textfiles or in

```
# define category domains
    BOOL <- c(TRUE,FALSE)
    OPTIONS <- letters[1:4]

# (conditional) numerical edits
    x + y == z
    2*u  + 0.5*v == 3*w
    w >= 0
    if ( x > 0 ) y > 0
    if ( x > y ) z < 10
    x >= 0
    y >= 0
    z >= 0

# categorical and mixed data edits
    A %in% OPTIONS
    B %in% OPTIONS
    C %in% BOOL
    D %in% letters[5:8]
    if ( A %in% c('a','b') ) y > 0
    if ( A == 'c' ) B %in% letters[1:3]
    if ( !C == TRUE) D %in% c('e','f')
```

Figure 1. Example of a free-form text file defining numerical, categorical and conditional edits. The edits can be read into R *with the* editfile *function.*

expression vector form. From a user perspective, the free-form textfiles are the most convenient when working reproducibly with (large) rule sets. The capabilities for parsing expression vectors is usefull when defining rules *ad hoc* on the command line. An overview of edit reading and coercing functions is given in Table 1.

As an example, Figure 1 shows the contents of an example text file that is included with the package. Note that the domains of categorical variables (the *data model*) may be defined separate from the edits. This is convenient for domains which are reused over several variables or for large domains that need to be read from file. Numerical and categorical edits can be submitted as they would for objects of class editmatrix or editarray respectively, while conditional edits must follow the following syntax diagram.



Here, `&` and `&&` and `|` and `||` are synonyms. The symbol ⟨*set expression*⟩ is an expression indicating set membership for categorical values, for example `A %in% c("a","b")`. See Van der Loo and De Jonge (2011b) for a syntax diagram of possible set expressions. Comments, (preceded by a #) are allowed as well, and will be ignored by the parser.

Since the file of Figure 1 is included with the package it can be read as follows.

```
> myfile <- system.file("script/edits/myedits.txt",
+     package="editrules")
> (E <- editfile(myfile))

Data model:
dat1 : A %in% c('a', 'b', 'c', 'd')
dat2 : B %in% c('a', 'b', 'c', 'd')
dat3 : C %in% c(FALSE, TRUE)
dat4 : D %in% c('e', 'f', 'g', 'h')

Edit set:
num1  : x + y == z
num2  : 2*u + 0.5*v == 3*w
num3  : 0 <= w
num4  : 0 <= x
num5  : 0 <= y
num6  : 0 <= z
cat7  : if( A == 'c' ) B != 'd'
cat8  : if( C == FALSE ) !( D %in% c('g', 'h') )
mix9  : if( 0 < x ) y > 0
```

```
mix10 : if( y < x ) 10 > z
mix11 : if( A %in% c('a', 'b') ) y > 0
```

In the first line, the example file is located using R's built-in system.file command. The second line is where the actal work is done: the function editfile takes a filename (including the path) as argument, reads and parses the edits in the file and returns an object of class editset, here stored in variable E. The extra brackets around the second statement are only added to force R to print the result to screen.

When an editset is printed, the data model for categorical variables, as well as the textual representation of the edits are shown. For convenience, edits are named according to their type. Pure numerical edits are numbered with prefix num, pure categorical with prefix cat and conditional edits are prefixed with mix.

The function editfile has an optional type argument, allowing for extracting only the numerical (type="num"), categorical (type="cat") or conditional edits (type="mix") from the text file. When type="num" or type="cat", an editmatrix or editarray is returned respectively. Using these more specialized objects yields some performance enhancement for common operations such as value substitution and variable elimination. Under the hood, editfile parses the file, looks for assignments (by <- or =) and evaluates them in a separate R environment. Next, the edits are generated within that environment.

Edits can be selected with the bracket operator, using integer or logical indices, for example:

```
> E[c(7,10),]

Data model:
dat1 : A %in% c('a', 'b', 'c', 'd')
dat2 : B %in% c('a', 'b', 'c', 'd')
dat3 : C %in% c(FALSE, TRUE)
dat4 : D %in% c('e', 'f', 'g', 'h')

Edit set:
cat1 : if( A == 'c' ) B != 'd'
cat2 : if( y < x ) 10 > z
```

By default, the full data model is retained when selecting a subset of edits. The reduce function can be used to remove variables not occurring in any edit from an editset object.

To export edits, the most convenient way is to use either as.character to convert an editset to text or as.data.frame to convert it to a 2-column data.frame. One

Table 1. Functions for reading and coercing (conditional) edits.

Function	Description
editfile	read from free-form textfile
editset	read from character or expression vector
as.character	convert editset to character vector
as.data.frame	convert editset to two-column data.frame

can then use R's standard I/O functionality to store edits as a structured text file, or use one of the database interfaces to send edits to a database.

2.2 Edit checking, obvious redundancy and obvious infeasibility

Data can be checked against edits in an editset with the violatedEdits function. This function accepts an editset and a data.frame and returns a logical array (of class violatedEdits) where each row and column indicates which record violates what edit. A summary and plot method is available for violatedEdits objects so users can get a quick overview of edit violation frequencies. Internally, the violatedEdits method for editsets works by coercing the edits to logical character expression and using R's evaluation functionality to parse and evaluate these expressions in the context of the data.frame.

An edit in an editset is obviously redundant when it is the duplicate of another edit or when it has an easily recognizable form such as $0 < 1$. Such redundancies may arise after edit manipulations (value substitution, variable elimination). The isObviouslyRedundant method for editset returns a logical vector indicating which edit in an editset is redundant (TRUE) or not (FALSE). If the editset was separated in independent conditional editsets by disjunct, a list of boolean vectors is returned. For a detailed description of detecting obvious redundancies in numerical or categorical edits, we refer to De Jonge and Van der Loo (2011) and Van der Loo and De Jonge (2011b).

An edit in an editset is obviously infeasible when it contains an easily recognizable self-contradicting edit, such as $0 > 1$. The function isObviouslyInfeasible returns TRUE for editsets containing one or more obvious contradictions in numerical or categorical edits. Note that when isObviouslyInfeasible returns FALSE, this does not guarantee that the set of edits is consistent. Contradictions may still be implied by the edits. Finding out whether a set of edits is satisfiable can be far more computationally intensive. We will return to this problem in Section 3.2.

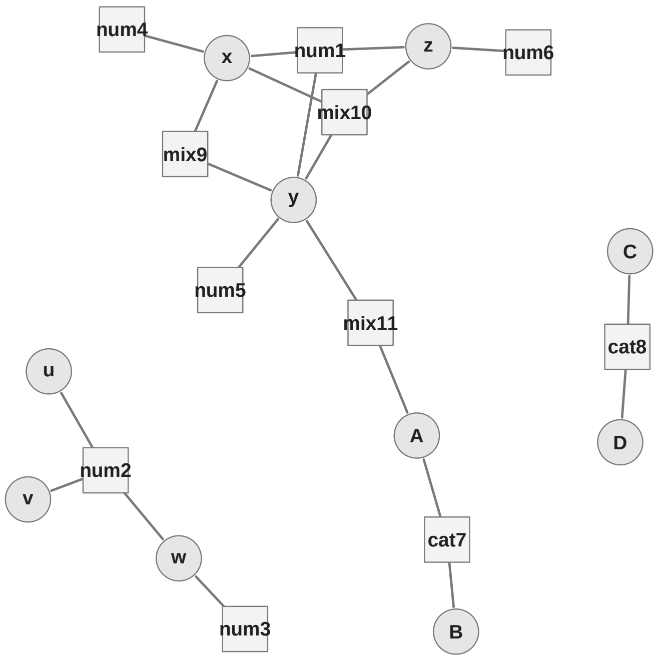


Figure 2. Dependency graph of edits defined in Figure 1. The squares represent edit rules, and the circles represent variables. An edge indicates that a variable occurs in an edit.

2.3 Visualizing and disentangling edits

As stated in the introduction, the fact that edits are entangled by shared variables severely complicates data editing: changing the value of a variable to solve an edit violation may cause the violation of another edit.

To make data editing more tractable, it is desired to break entangled sets of edits into smaller independent subsets as much as possible. Table 2 gives an overview of edit separation functions available in the editrules package. The most important ones will be discused below.

As an example, consider the dependency graph of the edits introduced in Figure 1. The graph can be generated by issuing the command

```
> plot(E)
```

and is depicted in Figure 2. A dependency graph represents the rules (squares) and variables (circles) that occur in an editset. A line is drawn between a square and a circle if the variable corresponding to the circle occurs in the rule represented by the square. Internally, the graph is generated by calling contains on E, which returns a logical matrix that indicates which edit contains which variables. Next, this matrix is converted to an igraph object and plotted with the igraph0 package (Csardi and Nepusz, 2006). The plot methods for

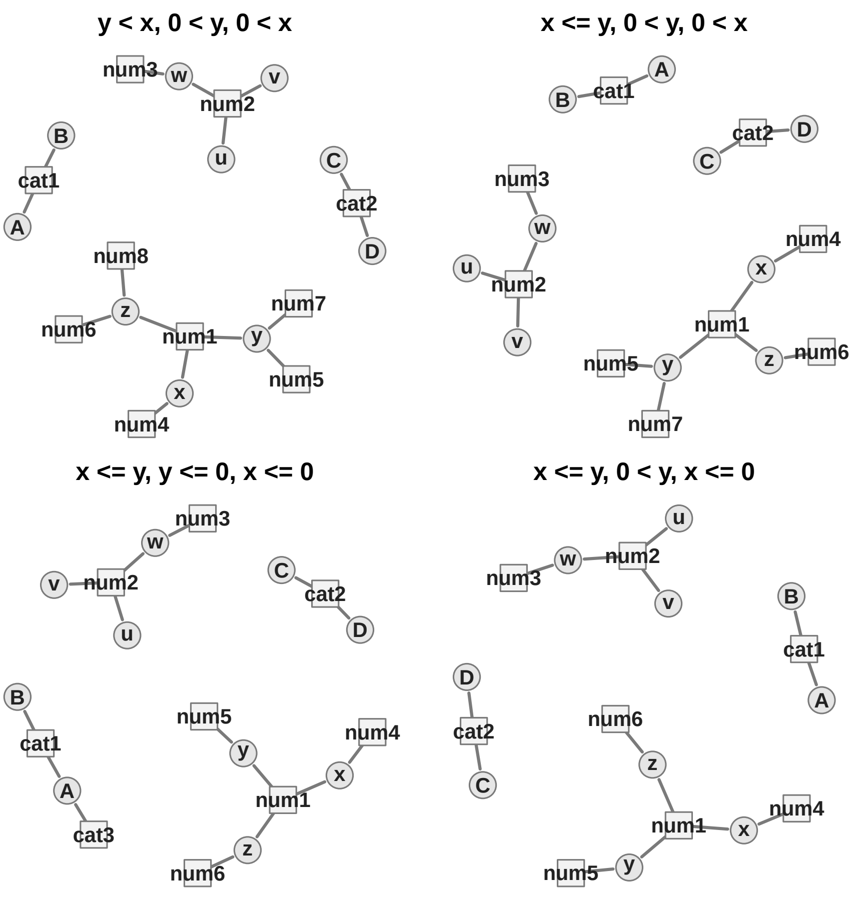


Figure 3. Dependency graphs of the conditional editsets *generated from the edits of Figure 1. There are no paths from numerical variables (x, y, z, u, v, w) to categorical variables (A, B, C, D) anymore. The titles of the subplots indicate the predicates for each* editset*. Edits with the same name contain the same variables but not necessarily the same condition on those variables across subplots.*

editset, editmatrix and editarray have several options for coloring violated edits or erroneous variables. Refer to the built-in documentation of the package for an extensive overview and examples.

The dependency graph clearly shows that our example set can be split into three unrelated blocks. These blocks (corresponding to columns of data) can therefore be treated separately when performing edit manipulations. Since important manipulations such as variable elimination have exponential complexity in the number of edits, recognizing such blocks can considerably enhance performance.

The higher-level error localization function that will be discussed in section 4.1, in fact does detect and exploit this block structure, so users need not concern themselves with it directly. To facilitate edit rule investigation and maintenance, the lower-level blocks function is also exported to user space. This function returns a list of the independent editsets, as illustrated by the following example.

```
> sapply(blocks(E),nedits)
```

```
[1] 8 2 1
```

Here, nedits counts the number of edits in an editset and sapply makes sure that nedits is applied to each member of the list returned by blocks. Clearly, the three independent blocks with 8, 3 and 1 edits (Figure 2) have been found.

The largest cluster of edits in Figure 2 connects numerical variables with categorical variables. Operations such as variable elimination are difficult to implement for such edit sets. However, it is possible to split up such a set further by working out what happens when we assume statements in the predicate or consequent to be TRUE or FALSE.

Consider again the edits on page 9. As an example, assume that $x > 0$ in `mix9`. We then know that $y > 0$ must hold. This means that `num5` becomes redundant and `mix11` reduces to $y > 0$ and becomes therefore redundant. On the other hand, when we assume $x \leq 0$, then `mix9` can be dropped, since the value of y has become unimportant for that edit. Combined with `num4`, assuming that $x \leq 0$ this means that $x = 0$.

The assumption $x > 0$ and $x \leq$ exclude each other. Working out their consequences therefore yield two different edit sets which cannot be obeyed fully by a record at the same time. If we continue making assumptions for the numerical statements in conditional edits recursively and work out their consequences, we get a list of *conditional* editsets where for each editset, the dependencies between categorical and numerical edits have been severed.

The function disjunct implements this procedure. It speeds up computation by detecting whenever contradictory assumptions have been made. The function returns a list of conditional editsets or optionally, an R environment containing those editsets. The conditions pertaining to each editset can be retrieved using the condition function. Figure 3 shows the dependency graphs of the four conditional editsets resulting after calling

```
> disjunct(E)
```

The conditional editsets form an equivalent representation of the original set and have the advantage that operations such as variable elimination (and therefore error localization) can be performed separately for each set. Note that in Figure

Table 2. Edit separation functions. Each function accepts an editset *as input.*

Function	Description
contains	Detects which edit contains which variable
plot	Plot the dependency graph
blocks	Splits an editset in independent edits not sharing any variables
disjunct	Splits an editset in disjunct sets, not containing mixed edits
condition	Returns the editmatrix holding the conditions for an editset generated by disjunct
separate	Uses blocks, simplifies the results, and calls disjunct on the remaining editsets

3, there are no paths running from numerical to categorical variables anymore. The downside is that for large, strongly connected edit sets, separation into disjunct sets can be a computationally daunting task, growing exponentially in the number of edits. We will shortly return to this problem in section 4. More background on manipulation of categorical edits will be descibed in a forthcoming paper (Van der Loo and de Jonge, 2012). Here, we just note that there are four numerical edits in the example set of Figure 1, yielding $2^4 = 16$ possible assumptions for their respective truth values. In principle this means that sixteen editset objects should be derived. However, because some assumptions conflict, only four subsets are generated. For example, the reader may veryfy that assuming that $x \leq 0$ and $y > 0$ and $y < x$ are contradictory demands.

Finally, we note that the utility function separate performs both the block decomposition based on variable occurrence and calls, when appropriate, the disjunct function on conditional edits. The results are simplified as much as possible and returned in a list.

3 Manipulation of conditional edits

The two basic operations on any set of restrictions, either numerical, categorical, or conditional, are value substitution and variable elimination. Methods for the pure numerical and pure categorical situations are fairly straightforward and have been implemented in the editrules package before. Operations on conditional edits require a bit more care, which will be detailed in the next two subsections.

3.1 Value substitution

Assigning a value to a variable occurring in an editset can be done with the substValue function. The consequences of substituting a value in conditional edits require a bit more care than for simple linear or categorical edits. Recall the truth table for the logical implication of q by p, denoted $p \rightarrow q$:

p	q	$p \rightarrow q$
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	TRUE
FALSE	FALSE	TRUE

When either predicate or consequent of a conditional edit resolves to a truth value after substituting a variable, the remaining edits must be processed according to this table. As a demonstration, consider the following simple editset, consisting of a single conditional edit.

```
> X <- editset("if ( x > 0 )  y > 0")
```

Substituting a value for x so that the predicate holds, obviously yields a numerical edit.

```
> substValue(X,'x',1)


Edit set:
num1 : 0 < y
```

Substituting a value so that the predicate does not hold yields an empty editset since the condition in the consequent only has to be obeyed when the predicate holds.

```
> substValue(X,'x',-1)


Edit set:
NULL :
```

The same happens when we enter a value for y so that the consequent holds:

```
> substValue(X,'y',1)


Edit set:
NULL :
```

since in that case, the value of the predicate is unimportant. On the other hand, when the restriction in the consequent resolves to FALSE, the predicate cannot be TRUE and must be inverted.

```
> substValue(X,'y',-1)


Edit set:
num1 : x <= 0
```

Observe that the substValue function recognizes that the remaining edits are purely numerical.

3.2 Variable elimination and satisfiability

Variable elimination is the mechanism by which edits, logically implied by a set of (user-defined) edits, are derived. Variable elimination amounts to deriving all implicit edit rules from a set of edits which do not contain the eliminated variable anymore. For an editset object, it is executed by first separating the edits in disconnected sets as described in Section 2.3. Next, variables are eliminated from the separate numerical or categorical parts of every separate conditional editset.

As an example, we will eliminate variable y from the following set of conditional edits.

$$G = \begin{cases} \text{if } x \geq 0 \text{ then } y \geq 0 \\ \text{if } x \geq 0 \text{ then } x \leq y \\ \text{if } x < 0 \text{ then } y < 0 \\ \text{if } x < 0 \text{ then } x - y < -2. \end{cases} \qquad (1)$$

Observe that these edits are all connected, since all of them contain both x and y. In Figure 4, the valid areas in the xy-plane defined by these edits are shown in grey. Because of their conditional structure, the edits define two disjunct subregions. Informally, if $x < 0$, a record (x, y) must be in the left grey region and if $x \geq 0$, it must be in the right grey region.

Recall that geometrically, eliminating a variable from a set of linear (in)equalities amounts to a projection along the corresponding axis. Here, eliminating y amounts to projecting the grey areas along the y-axis, yielding two separate line segments, shown in bold in Figure 4.

Figure 5 shows how to perform the elimination with the editrules package. In the first line, the edits are defined, using an expression vector as input. In the second line, variable y is eliminated by calling eliminate. The result is an object

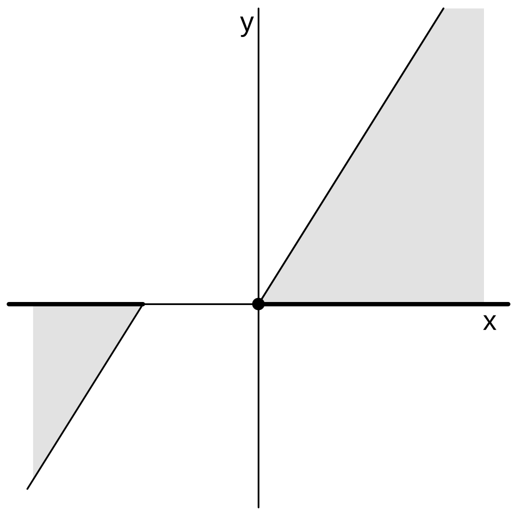


Figure 4. Graphical representation (in gray) of the valid areas defined by the edits of Eq. (1)*. Depicted are sections of the bordering lines $y = x$ and $y = x+2$. The bold lines indicate the projections of the gray areas along the y-axis.*

```
> G <- editset(expression(
+     if ( x >= 0 ) y >= 0,
+     if ( x >= 0 ) x <= y,
+     if ( x < 0 ) y < 0,
+     if ( x < 0 ) x - y < -2
+ ))
> eliminate(G,"y")

editsets:

Set 1  conditions:
0 <= x

Edit set:
NULL :

Set 2  conditions:
x < 0

Edit set:
num1 : x < -2
```

Figure 5: Eliminating a variable from an object of class editset results in a number of editsets which are stored in an editlist object.

of class editlist, containing two editsets. The first editset holds when $x \geq 0$ and holds no further restrictions: the restriction $x \geq 0$ corresponds exactly with the projection of the right grey area of Figure 4 on the x-axis. The second editset holds when $x < 0$, and imposes that in that case $x < -2$.

4 Error localization in mixed data

Given that a record violates a number of edit rules, the problem remains to point out which fields cause the error. When there is no clear evidence pointing out which variables cause the violations, one can resort to adapting as few variables as possible. Adapting as few variables as possible, without violating any new, implied rules is referred to as the principle of Fellegi and Holt (1976), who described the first systematic approach to error localization.

4.1 Error localization with localizeErrors and errorLocalizer

In editrules, errors can be localized according to Felligi and Holt's principle using the localizeErrors function. This function has been introduced earlier for purely numerical data (De Jonge and Van der Loo, 2011) and for purely categorical data (Van der Loo and De Jonge, 2011a), and has now been extended to handle conditional edits.

The interface of localizeErrors is exactly the same as described in the references above. The minimal input is an editset and a data.frame and the output is an object of class errorLocalizer, which holds error locations as well as some logging info. Below we give a simple demonstration.

```
> E <- editset(expression(
+     if ( x > 0 ) y > 0,
+     x + y == 10
+ ))
> dat <- data.frame(x = 1, y = -5)
> el <- localizeErrors(E,dat)
> el$adapt

     x     y
1 TRUE FALSE

> el$status

  weight degeneracy  user system elapsed maxDurationExceeded
1      1          2 0.012      0   0.015               FALSE
```

In the first line we define an edit set demanding that when $x > 0$, then $y > 0$ and that x and y must add up to 10. The second line defines a record where

($x = 1, y = -5$). Obviously, this record violates both restrictions. The errors may be resolved by either adapting y or x and y. The former solution is the minimal case and this is what localizeErrors returns in line 3: the array el$adapt indicates with a boolean value which variable in which record must be changed. The data.frame el$status gives information on the total solution weight, the number of equivalent solutions (degeneracy) and the amount of time it took to compute the solution.

4.2 Some details on implementation

The error localization problem consists of finding the least (weighted) number of fields in a record that can be adapted or imputed, such that no edits are violated anymore. De Waal (2003) developed a branch-and-bound algorithm which computes solutions to the localization problem by systematically building and testing partial solutions. The computational time necessary for the algorithm to complete is exponential in the number of variables. Therefore, partial solutions which cannot lead to a solution are abandoned (pruning) as much as possible. To decrease the number of variables entering the branch-and-bound algorithm, edits sets are separated in independent blocks (using the blocks function). Since version 2.5 of editrules, error localization is accelerated further by adding variables which violate univariate constraints to the solution set prior to entering the branch-and-bound algorithm.

Because for conditional edits, an editset must be separated by the disjunct function described earlier, error localization using the branch-and-bound approach can become computationally expensive when many entangled conditional edits are involved. For this reason a second algorithm based on a mixed-integer formulation of the problem has been implemented as well. This algorithm avoids explicit variable elimination and value substitution and will be reported upon in a separate paper (De Jonge and Van der Loo, 2012).

5 Conclusions

We described new functionality of the editrules package, pertaining to conditional restrictions and mixed data edits. All existing edit manipulation functions have been extended (overloaded) to handle the new editset object and several new edit manipulation features have been added. Most notably the possibility to read edits from a free-form textfile and the option to split sets of edits in disjunct sets that do not contain any mixed data edits anymore. Also, the branch-and-bound error localization methods have been accelerated by taking care of range edit violations prior to multivariate error localization.

Future work on the package may include the extension to *soft edits*, where not only the violation of an edit is weighed in the process of error localization but also the *amount* of violation can be taken into account during error localization.

References

Csardi, G. and T. Nepusz (2006). The igraph software package for complex network research. *InterJournal Complex Systems*, 1695.

De Jonge, E. and M. Van der Loo (2011). Manipulation of linear edits and error localization with the editrules package. Technical Report 201120, Statistics Netherlands, The Hague.

De Jonge, E. and M. van der Loo (2012). *editrules: R package for parsing and manipulating of edit rules and error localization.* R package version 2.5.

De Jonge, E. and M. Van der Loo (2012). Error localization as a mixed-integer problem with the editrules package. Technical Report 2012XX, Statistics Netherlands, The Hague. forthcoming.

De Waal, T. (2003). *Processing of erroneous and unsafe data.* Ph. D. thesis, Erasmus University Rotterdam.

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

De Waal, T. and R. Quere (2003). A fast and simple algorithm for automatic editing of mixed data. *Journal of Official Statistics 19*, 383–402.

Fellegi, I. P. and D. Holt (1976). A systematic approach to automatic edit and imputation. *Journal of the Americal Statistical Association 71*, 17–35.

Van der Loo, M. and E. De Jonge (2011a). Deductive imputation with the deducorrect package. Technical Report 201126, Statistics Netherlands.

Van der Loo, M. and E. De Jonge (2011b). Manipulation of categorical data edits and error localization with the editrules package. Technical Report 201129, Statistics Netherlands.

Van der Loo, M. and E. de Jonge (2012). Algorithms of the editrules package. Technical report, Statistics Netherlands. Forthcoming.