Manipulation of categorical data edits and error localization with the editrules package

Mark van der Loo and Edwin de Jonge

The views expressed in this paper are those of the author(s)
and do not necessarily reflect the policies of Statistics Netherlands

Discussion paper (201129)

Statistics Netherlands

The Hague/Heerlen, 2011

Explanation of symbols

.	= data not available
*	= provisional figure
**	= revised provisional figure
x	= publication prohibited (confidential figure)
–	= nil or less than half of unit concerned
–	= (between two figures) inclusive
0 (0,0)	= less than half of unit concerned
blank	= not applicable
2010–2011	= 2010 to 2011 inclusive
2010/2011	= average of 2010 up to and including 2011
2010/'11	= crop year, financial year, school year etc. beginning in 2010 and ending in 2011
2008/'09– 2010/'11	= crop year, financial year, etc. 2008/'09 to 2010/'11 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

60083201129 X-10

Manipulation of categorical data edits and error localization with the editrules package.

Mark van der Loo and Edwin de Jonge

Summary: Analyses of categorical data are often hindered by the occurrence of inconsistent or incomplete raw data. Although R has many features for analyzing categorical data, the functionality for error localization and error correction are currently limited. The editrules package is designed to offer a user-friendly toolbox for edit definition, manipulation, and error localization based on the generalized paradigm of Fellegi and Holt.

This is the second paper describing functionalities of the R editrules package and marks the completion of editrules version 2.0. The first paper (De Jonge and Van der Loo, 2011) describes methods and implementation for handling numerical data while this paper is concerned with handling categorical data.

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

Contents

1 **Introduction** 5

2 **Categorical data and edit rules** 5

2.1 Describing records and edit rules 6

2.2 The editarray object 7

2.3 Coercion, checking, redundancy and feasibility 11

3 **Manipulation of categorical restrictions** 14

3.1 Value substitution 14

3.2 Elimination of variables 16

4 **Error localization in categorical data** 17

4.1 A Branch and bound algorithm 17

4.2 Error localization with localizeErrors 20

4.3 Error localization with errorLocalizer 22

5 **Conclusions** 24

Index 25

List of Algorithms

1 isSubset(E) 13

2 substValue(E,k,v) 14

1 Introduction

Analyses of categorical data are often hindered by occurrences of incomplete or inconsistent raw data records. The process of locating and correcting such errors is referred to as *data editing*, and it has been estimated that National Statistics Institutes spend up to 40% of their resources on this process (De Waal et al., 2011). For this reason, considerable attention is paid to the development of data editing methods that can be automated. Since data are often required to obey many interrelated consistency rules, data editing can be too complex to perform manually. Winkler (1999) mentions practical cases where records have to obey 250, 300 or even 750 internal consistency rules. Although the R statistical environment has numerous facilities for analyzing categorical data [See *e.g.* Husson et al. (2010)], the options for error localization and record correction are currently limited.

This paper presents version 2.0 of R package editrules, which was developed to help closing the gap between raw data retrieval and data analysis with R. The main purpose of the editrules package is to provide a user-friendly environment for handling data restriction rules, to apply those rules to data, and to localize erroneous fields in data based on the generalized principle of Fellegi and Holt (1976). The package does not offer functionality for data correction. However, it does facilitate the identification of the set of solutions for an error correction problem.

Under the hood, the package contains several innovations with respect to the branch-and-bound algorithm for error localization in categorical data described in De Waal et al. (2011). The most important innovation is a new variable elimination algorithm of which allows for on-the-fly redundant rule removal. The algorithm itself will be reported in a forthcoming paper.

The current paper complements our previous paper on the treatment of numerical data (De Jonge and Van der Loo, 2011). We describe some of the algorithms underlying editrules' functionality and the internal representation of categorical data. Examples in R code are given throughout the text to assist new users in getting started with the package.

2 Categorical data and edit rules

The value domain of categorical data records is usually limited by rules interrelating these variables. The simplest examples are cases where the value of one variable excludes values of another variable. For example: if the age class of a person is “child”, then (by law) the marital status cannot be “married”. In survey or administrative data, violations of such rules are frequently encountered. Resolving such violations is an important step prior to data analysis and estimation.

In this section we describe the representation of edits and records as implemented in the editrules package and we report on the basic (low-level) edit rule manipulation functionality.

2.1 Describing records and edit rules

A record of n categorical variables can be written as an element of the cartesian product space D:

$$D = D_1 \times D_2 \times \cdots \times D_n, \tag{1}$$

where each D_k is the set of possible categories for variable k. The domain of values D is often referred to as the *data model* for a dataset. The number of categories for variable k is labeled d_k while the total number of categories is labeled d, given by

$$d = \sum_{k=1}^{n} d_k. \tag{2}$$

As an example, consider the domain of a record consisting of the variables *marital status*, *age* and *position in household*, so that $D = D_1 \times D_2 \times D_3$:

$$\begin{aligned} D_1 &= \{\mathsf{married}, \mathsf{unmarried}, \mathsf{widowed}, \mathsf{divorced}\} && (3)\\ D_2 &= \{\mathsf{under\text{-}aged}, \mathsf{adult}\} && (4)\\ D_3 &= \{\mathsf{spouse}, \mathsf{child}, \mathsf{other}\}. && (5)\end{aligned}$$

In total, there are $d_1 \cdot d_2 \cdot d_3 = 4 \cdot 2 \cdot 3 = 24$ records in D.

To represent a record, we construct an index vector in $\{0, 1\}^d$ indicating the categories in a record. For example, the record:

$$(\mathsf{married}, \mathsf{adult}, \mathsf{spouse}). \tag{6}$$

can be represented as

$$(1, 0, 0, 0) \oplus (0, 1) \oplus (1, 0, 0) = (1, 0, 0, 0, 0, 1, 1, 0, 0), \tag{7}$$

where $\oplus$ is the direct vector sum. Note that the position of the 1's indicate which value each variable has. In this representation, a record with n variables has precisely n 1's and the rest 0's. We will refer to this representation of categorical records as the *boolean representation*, since the 1's and 0's can be interpreted as TRUE and FALSE respectively. We show below that it is useful to allow boolean operations (and, or, negation) on such vectors.

In practical cases, not every record in a domain, defined as in [Eq. (1)] will be valid. For example, we may want to exclude records such as

$$(\textsf{married}, \textsf{under-aged}, \textsf{spouse}), \tag{8}$$

since by law, under-aged people cannot be married.

In general, a data consistency rule, (called *edit rule* or *edit* in short) e is a subset of D, with the interpretation that if a record falls in e, it is invalid. Every edit can be written in the form

$$e = A_1 \times A_2 \times \cdots A_n, \tag{9}$$

where each $A_k \subseteq D$. For example, the rules which states:

$$\text{under-aged persons cannot be married,} \tag{10}$$

can be written as

$$\{\textsf{married}\} \times \{\textsf{under-aged}\} \times \{\textsf{spouse}, \textsf{child}, \textsf{other}\}. \tag{11}$$

Just like records, edits can be represented with boolean vectors where the 1s indicates which categories are a member of the edit. The edit in (11) can be represented as

$$(1, 0, 0, 0) \oplus (1, 0) \oplus (1, 1, 1) = (1, 0, 0, 0, 1, 0, 1, 1, 1). \tag{12}$$

In editrules, edits are represented as boolean vectors. A set of edits is represented as two-dimensional array, where each row represents a single edit.

2.2 The editarray object

In the editrules package, a set of categorical edits is storeed as an editarray object. We denote an editarray E for n categorical variables and m edits as (brackets indicate a combination of objects)

$$E = \langle \mathbf{A}, \mathbf{ind} \rangle, \text{ with } \mathbf{A} \in \{0, 1\}^{m\times d} \text{ and } d = \sum_{k=1}^{n} d_k, \tag{13}$$

Each row $\mathbf{a}$ of $\mathbf{A}$ contains the boolean representation of one edit, and the d_k denote the number of categories of each variable. The object $\mathbf{ind}$ is a nested list which relates columns of $\mathbf{A}$ to variable names and categories. Labeling variables with $k \in 1, 2, \ldots, n$ and category values with $v_k \in 1, 2, \ldots, d_k$, we use the following notations:

$$\mathbf{ind}(k, v_k) \quad = \quad \sum_{l<k} d_l + v_k \tag{14}$$

```
> E <- editarray(c(
+      "gender %in% c('male','female')",
+      "pregnant %in% c('yes','no')",
+      "if (gender == 'male') pregnant == 'no'"
+      )
+ )
> E

Edit array:
     levels
edits gndr:feml gndr:male prgn:no prgn:yes
   e1     FALSE      TRUE   FALSE     TRUE

Edit rules:
d1  :  gender %in% c('female', 'male')
d2  :  pregnant %in% c('no', 'yes')
e1  :  if( gender == 'male' ) pregnant != 'yes'

> datamodel(E)

  variable  value
1   gender female
2   gender   male
3 pregnant     no
4 pregnant    yes
```

Figure 1: Defining a simple editarray with the editarray function. The array is printed with abbreviated column heads, which themselves consist of variable names and categories separated by a colon (by default). When printed to screen, a character version of the edits is shown as well, for readability.

$$\mathbf{ind}(k) \quad = \quad \{\mathbf{ind}(k, v_k) \,|\, v_k \in D_k\}. \tag{15}$$

So $\mathbf{ind}(k, v_k)$ is the column index in $\mathbf{A}$ for variable k and category v_k and $\mathbf{ind}(k)$ is the set of column indices corresponding to all categories of variable k. The editarray is the central object for computing with categorical edits, just like the editmatrix is the central object for computations with linear edits [De Jonge and Van der Loo (2011)].

It is both tedious and error prone to define and maintain an editarray by hand. In practice, categorical edits are usually stated verbosely, such as: "a man cannot be pregnant", or "an under-aged person cannot be married". To facilitate the definition of edit arrays, editrules is equipped with a parser which takes R-statements in character and translates them to an editarray.

Figure 1 shows a simple example of defining an editarray with the editrules package. The first two edits in Figure 1 define the domain. The editarray function derives the datamodel based on the variable names and categories it finds in the edits, whether they are univariate (defining domains) or multivariate. This means that if all possible variables and categories are mentioned in the multivariate edits, the correct datamodel will be derived as well.

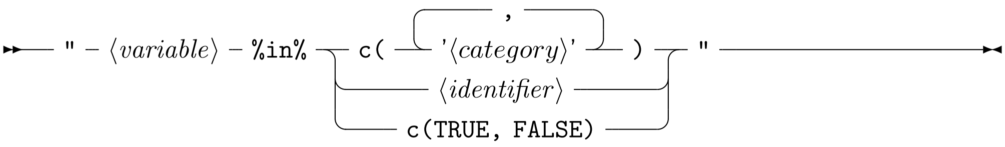
When printed to the screen, the boolean array is shown with column heads of the form

<abbreviated var. name><separator><abbreviated cat. label>

where both variable names and categories are abbreviated for readability, and the standard separator is a colon (:). The separator may not occur as a symbol in either variable or category name, and its value can be determined by passing a custom sep argument to editarray. For convenience, the function datamodel accepts an editarray as input and returns an overview of variables and their categories for easy inspection in the form of a data.frame.

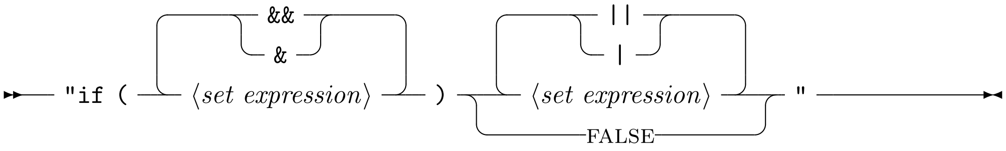
Internally, editarray uses the R internal parse function to transform a character expressions to a parse tree. This tree is subsequently traversed recursively to derive the entries of the editarray. The reverse operation is also implemented. The R internal function as.character has been overloaded to derive a character representation from a boolean representation. When printed to the screen, both the boolean and textual representation are shown.

Univariate edits define the domain of a single variable. Together, these domains form a data model. A domain can be defined with common R syntax using the %in% operator. If a domain is defined explicitly, the edit must follow the following syntax diagram.

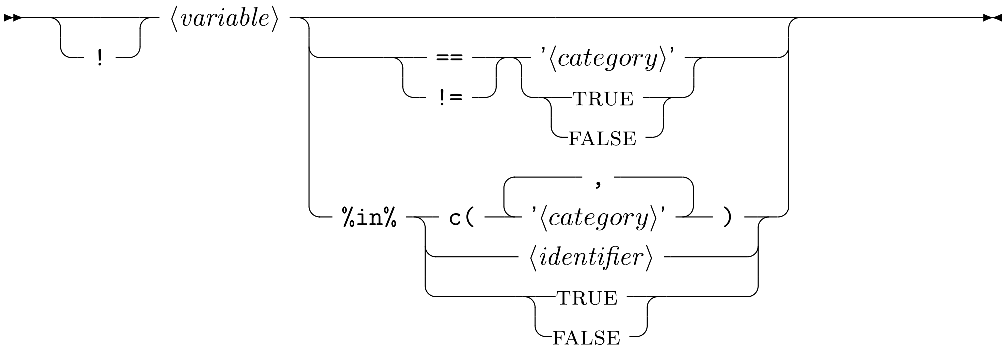


Here, ⟨*variable*⟩ is the name of a categorical variable, and ⟨*category*⟩ is a literal category name. Note that the category name is enclosed by single quotes while the entire statement is between double quotes. So here, the entire statement is offered in string format to editarray. The ⟨*identifier*⟩ is the name of a predefined character variable storing the unique categories for a variable. The suymbol ⟨*identifier*⟩ denotes a previously defined R character or factor vector listing categories for a variable. Since the ⟨*identifier*⟩ is evaluated as an expression, in principle any R expression evaluating to a character or factor vector may be used as well. However, we do not recommend such constructions since they clutter a clear definition of the datamodel.

Multivariate rules can be defined in two ways. The most useful and common way to define edits follows the following syntax diagram.



Where each ⟨*set expression*⟩ is a logical statement following



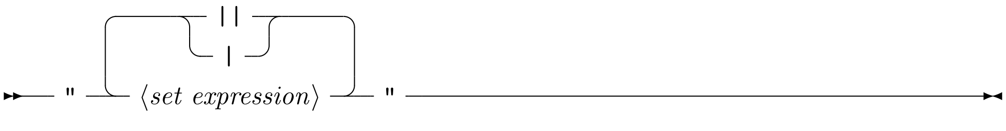
The reader can check that the examples given in Figure 1 follow this syntax. The example below illustrates the use of separately defined data models and boolean categories.

```
> xval <- letters[1:4]
> yval <- c(TRUE,FALSE)
> ( M <- editarray(c(
+      "x %in% xval",
+      "y %in% yval",
+      "if ( x %in% c('a','b') ) !y "
+      )) )

Edit array:
     levels
edits   x:a  x:b   x:c   x:d y:FALS y:TRUE
   e1 TRUE TRUE FALSE FALSE  FALSE   TRUE

Edit rules:
d1  :  x %in% c('a', 'b', 'c', 'd')
d2  :  y %in% c(FALSE, TRUE)
e1  :  if( x %in% c('a', 'b') ) y == FALSE
```

The second way to define multivariate edits is based on rewriting on the basic classical logic law $P \Rightarrow Q = \neg P \vee Q$. It involves the following syntax diagram.



Where each ⟨*set expression*⟩ is as in the syntax diagram above. In practice, a user will commonly not use this form since it is less readable. However, the as.character method for editarray can generate such statements by passing the argument useIf=FALSE, as shown below.

```
> as.character(M,useIf=FALSE)

                                      d1                                       d2
        "x %in% c('a', 'b', 'c', 'd')"              "y %in% c(FALSE, TRUE)"
                                      e1
"!( x %in% c('a', 'b') ) | y == FALSE"
```

The main advantage of this form is that contrary to the if() form, it allows for vectorized checking of edits, which is why it is used internally.

2.3 Coercion, checking, redundancy and feasibility

The editrules package is equipped with functions operating on sets of edits represented as an editarray. An overview is given in Table 1.

The datamodel function extracts the variables and categories from an editarray, and returns them as a two-column data.frame. With as.data.frame or as.character one can coerce an editarray so that it can be written to a file or database. Character coercion is also used when edits are printed to the screen. Coercing the data model to character form can be switched off by passing the option datamodel=FALSE to as.character. The result of as.data.frame has columns with edit names, a character representation of the edits and a column for remarks.

The function violatedEdits takes an editarray and a data.frame as input and returns a logical matrix indicating which record (rows) violate which edits (columns). It works by parsing the editarray to R-expressions and evaluating them as logical expressions within the data.frame environment. By default, the records are checked against the data model. This can be turned off by providing the optional argument datamodel=FALSE.

When manipulating edit sets, some edits may become redundant. We distinguish two redundancy situations. The first situation occurs when an edit e of the form in Eq. (9) has $A_k = \varnothing$ for at least one variable k. In this case, no record can ever be an element of e, making e obsolete. Such edits can be removed from a set of edits without harming any further processing or record checking. The second situation occurs when every record in an edit is also a member of a second edit. In this case, the first edit is redundant with respect to the second. The second case can easily be detected in the boolean representation, for if $\mathbf{a}$ and $\mathbf{b}$ are edits in the boolean representation, then $\mathbf{a}$ is redundant

Table 1. Functions for objects of class editarray*. Only mandatory arguments are shown, refer to the built-in documentation for optional arguments. The functions are described in the subsections indicated between square brackets.*

Function	description
datamodel(E)	get datamodel [2.3]
getVars(E)	get a list of variables
as.data.frame(E)	coerce edits to data.frame
contains(E)	which edits contains which variable
as.character(E)	coerce edits to character vector [2.2]
blocks(E)	Get list of independent blocks of edits
reduce(E)	Remove redundant variables and rows [3.1]
isObviouslyRedundant(E)	find redundancies, duplicates [2.3]
duplicated(E)	find duplicate edits [2.3]
isSubset(E)	find edits, subset of another edit [2.3]
isObviouslyInfeasible(E)	detect simple contradictions [2.3]
isFeasible(E)	check edit set consistency [2.3]
substValue(E,var,value)	substitute a value [3.1]
eliminate(E,var)	eliminate a variable [3.2]
violatedEdits(E,dat)	check which edits are violated by dat [2.3]
localizeErrors(E,dat)	localize errors [4.2]
errorLocalizer(E,x)	backtracker for error localization [4.3]
summary(E)	summarize the content of E
plot(E)	plot a graph of E (requires igraph package)

Algorithm 1 isSubset(E)

Input: An editarray $E = \langle \mathbf{A}, \mathbf{ind} \rangle$.

```
$\mathbf{s} \leftarrow (\text{FALSE})^m$
for $(\mathbf{a}^{(i)}, \mathbf{a}^{(i')}) \in \text{rows}(\mathbf{A}) \times \text{rows}(\mathbf{A})$ do
    if $\mathbf{a}^{(i)} \vee \mathbf{a}^{(i')} = \mathbf{a}^{(i')}$ then
        $s_i \leftarrow \text{TRUE}$
```

Output: Boolean vector s indicating which edits represented by $\mathbf{A}$ are a subset of another edit.

with respect to $\mathbf{b}$ if $\mathbf{a} \wedge \mathbf{b} = \mathbf{a}$, or equivalently when $\mathbf{a} \vee \mathbf{b} = \mathbf{b}$. Here, the logical operators $\wedge$ and $\vee$ work elementwise on the boolean vectors.

In editrules, the first type of redundancy can be detected in the boolean representation with isObviouslyRedundant. By default, this function also checks for duplicate edits, but this may be switched off with an extra parameter. Also, the standard R function duplicated has been overloaded to search for duplicate edits in an editarray directly. The second type of redundancy can be detected with the function isSubset. The pseudocode is given in Algorithm 1. In the actual R implementation, the only explicit loop is a call to R's vapply function. The other loops are avoided using R's indexing and vectorization properties.

Manipulations may also lead to edits of the form $e = D$, in which case every possible record is invalid, and the editarray has become impossible to satisfy. The function isObviouslyInfeasible detects whether any such edits are present. The function isFeasible checks if the set of edits in its argument allows at least one valid record. This may yield results which are counterintuitive at first glance. For example, consider a set of edits on the domain $D = \{(x, y) \in \{\mathsf{a}, \mathsf{b}\} \times \{\mathsf{c}, \mathsf{d}\}\}$.

```
> M <- editarray(c(
+     "x %in% c('a','b')",
+     "y %in% c('c','d')",
+     "if ( x == 'a' ) y == 'c'",
+     "if ( x == 'a' ) y != 'c'"))
>
```

This set of edits is feasible, even though edits e_1 and e_2 seem to contradict each other:

```
> isFeasible(M)
```

```
[1] TRUE
```

The explanation is that e_1 and e_2 contradict each other only when $x = \mathsf{a}$, so

Algorithm 2 SUBSTVALUE(E,k,v)

```
Input: an editarray $E = \langle \mathbf{A}, \mathbf{ind} \rangle$, a variable index $k$ and a value $v$
$i \leftarrow \mathbf{ind}(k, v)$
$\mathbf{A} \leftarrow \mathbf{A} \backslash \{\mathbf{a} \in \text{rows}(\mathbf{A}) \,|\, a_i = \text{FALSE}\}$    ▷ Remove rows not involving $v$
$\mathbf{A} \leftarrow \mathbf{A} \backslash \{\mathbf{a}_j^t \in \text{columns}(\mathbf{A}) \,|\, j \in \mathbf{ind}(k) \backslash i\}$    ▷ Remove categories $\neq v$
Update $\mathbf{ind}$
Output: $\langle \mathbf{A}, \mathbf{ind} \rangle$ with $v$ substituted for variable $k$.
```

```
> isFeasible(substValue(M,'x','a'))

[1] FALSE
```

where the function substValue is discussed in the next section. One can check that the record ($x = \mathsf{b}, y = \mathsf{d}$) indeed satisfies all edits in M.

The feasibility check works by eliminating all variables in an editarray one by one until either no edits are left or an obvious contradiction is found. Variable elimination is discussed further in section 3.2.

3 Manipulation of categorical restrictions

The basic operations on sets of categorical edits are value substitution and variable elimination. The former amounts to adapting the datamodel underlying the edit set while the latter amounts to deriving relations between variables not involving the eliminated variable.

In the next subsection we give an example of value substitution with the editrules package, as well as some background. Subsection 3.2 we give an example of eliminating variables with the editrules package.

3.1 Value substitution

If it is assumed that in a record one of the variables takes a fixed value, that value may be substituted in the edit rules. In the boolean representation this amounts to removing all edits which exclude that value, since the record cannot violate those edits. Secondly, the columns related to the substituted variable but not to the substituted category are removed, thereby adapting the datamodel to the new assumption. Algorithm 2 gives the pseudocode for reference purposes.

In the editrules package, value substitution is performed by the substValue function, which accepts an editarray, a variable name and a category name. In the following example the editmatrix defined in Figure 1 is used.

```
> substValue(E,"gender","female")

Edit array:
      levels
edits  gndr:feml gndr:male prgn:no prgn:yes

Edit rules:
d1  :  gender %in% c('female', 'male')
d2  :  pregnant %in% c('no', 'yes')
```

In this case, the variable *gender* is substituted by the value female. The rules concerning *gender* = male may be deleted, so here only the datamodel is left without any multivariate rules. In fact, the datamodel itself may be reduced, which can be achieved by setting the option reduce=TRUE.

```
> substValue(E,"gender","female",reduce=TRUE)

Edit array:
      levels
edits  gndr:feml prgn:no prgn:yes

Edit rules:
d1  :  gender %in% 'female'
d2  :  pregnant %in% c('no', 'yes')
```

Here, the only option left for *gender* is included explicitly in the datamodel. For some operations it is necessary to remove such redundancies as well. The function reduce completely removes variables from an edit set for which the value has been fixed:

```
> reduce(substValue(E,"gender","female"))

Edit array:
<0 x 0 matrix>

Edit rules:
  :
```

The substitution renders the only edit rule obsolete. Since after the substitution, there are no limitations on the variable *pregnant*, (except for the datamodel) this variable is deleted from the editarray as well. The reduce function is used extensively in the deductive imputation routines of the deducorrect package (Van der Loo and De Jonge, 2011).

3.2 Elimination of variables

The purpose of the eliminate function is to derive all possible non-redundant edits from an edit set that do not contain a certain variable. For categorical data edits, this amounts to a series of logic resolution operations. As an example, consider the following syllogism:

$$\begin{array}{cl} P_1 & \text{An Under-aged person cannot be married} \\ P_2 & \text{A spouse has to be married} \\ \hline C & \text{An under-aged person cannot be a spouse.} \end{array}$$

Here, the conclusion C following from premises P_1 and P_2 does not contain the variable *marital status* anymore. That is, *marital status* is eliminated.

To generalize this operation, note that using Eqs. (3)-(5), the above syllogism can be written as

$$\begin{array}{ccccc} \{\mathsf{married}\} & \times & \{\mathsf{under\text{-}aged}\} & \times & D_3 \\ \{\mathsf{unmarried}, \mathsf{widowed}, \text{divorced}\} & \times & D_2 & \times & \{\mathsf{spouse}\} \\ \hline D_1 & \times & \{\mathsf{under\text{-}aged}\} & \times & \{\mathsf{spouse}\}. \end{array} \tag{16}$$

Note that the resulting edit is derived by taking the set union in the first variable and the set intersection in the other variables. A generalized version of this operation was first described in the context of error localization by Fellegi and Holt (1976). They also devised a combinatorial algorithm that generates every edit from an edit set that does not contain some chosen variable. The elimination algorithm in editrules is different, and is based on repeated application of the binary operation shown in Eq. (16). The details of this algorithm will be published in a forthcoming paper.

In editrules the above operation can be performed as follows. We first define a data model and edit rules:

```
> E <- editarray(c(
+     "age %in% c('under-aged','adult')",
+     "maritalStatus %in% c('married','not married')",
+     "positionInHousehold %in% c('spouse','child','other')",
+     "if (age == 'under-aged') maritalStatus != 'married'",
+     "if (positionInHousehold == 'spouse') maritalStatus == 'married'"
+ ))
```

We may derive the conclusion by eliminating the *marital status* variable:

```
> eliminate(E,'maritalStatus')
```

```
Edit array:
     levels
edits age:adlt age:und- mrtS:mrrd mrtS:ntmr psIH:chld psIH:othr psIH:spos
   e1    FALSE     TRUE      TRUE      TRUE     FALSE     FALSE      TRUE

Edit rules:
d1  :  age %in% c('adult', 'under-aged')
d2  :  maritalStatus %in% c('married', 'not married')
d3  :  positionInHousehold %in% c('child', 'other', 'spouse')
e1  :  if( age == 'under-aged' ) positionInHousehold != 'spouse'
```

This indeed yields the right conclusion. Alternatively, we may eliminate *age*:

```
> eliminate(E,'age')

Edit array:
     levels
edits age:adlt age:und- mrtS:mrrd mrtS:ntmr psIH:chld psIH:othr psIH:spos
   e1     TRUE     TRUE     FALSE      TRUE     FALSE     FALSE      TRUE

Edit rules:
d1  :  age %in% c('adult', 'under-aged')
d2  :  maritalStatus %in% c('married', 'not married')
d3  :  positionInHousehold %in% c('child', 'other', 'spouse')
e1  :  if( maritalStatus == 'not married' ) positionInHousehold != 'spouse'
```

This deletes the only rule actually involving *age*. That is, no new rules not involving *age* can be derived.

4 Error localization in categorical data

4.1 A Branch and bound algorithm

The editrules package implements an error localization algorithm, based on the branch-and-bound algorithm of De Waal and Quere (2003). The algorithm has been extensively described in De Waal (2003) and De Waal et al. (2011). The algorithm is similar to the branch-and-bound algorithm used for error localization in numerical data in the editrules package as described in De Jonge and Van der Loo (2011), except that the elimination and substitution subroutines are implemented for categorical data.

In short, a binary tree is created with the full set of edits and an erroneous record at the root node. Two child nodes are created. In the first child node the first variable of the record is assumed correct, and its values is substituted in the edits. In the second child node the variable is assumed incorrect and it is eliminated from the set of edits. The tree is continued recursively until choices are made for each variable. Branches are pruned when they cannot lead to a solution, leaving a partial binary tree where each path from root to leaf represents a solution to the error localization problem. Computational complexity is reduced further by pruning branches leading to higher-weight solutions then solutions found earlier.

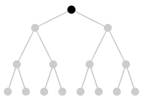
Recall the datamodel of Eqs. (3)-(5), with variables *marital status*, *age* and *position in household*. We define the following two edits:

$e^{(1)}$	An under-aged person cannot be married
$e^{(2)}$	A spouse has to be married

As an example we treat the following record with the branch-and-bound algorithm to localize the errors:

$$(\mathsf{married}, \mathsf{under\text{-}aged}, \mathsf{spouse}). \tag{17}$$

At the beginning of the algorithm, only the root node is filled. The situation may be represented as follows:

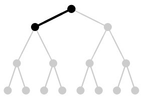


	mar. stat.				*age*		*pos. in hh*		
$\mathbf{v}$	1	0	0	0	1	0	1	0	0
$\mathbf{a}^{(1)}$	1	0	0	0	1	0	1	1	1
$\mathbf{a}^{(2)}$	0	1	1	1	1	1	1	0	0

Root node, $w = 0$,

where $\mathbf{v}$ is the boolean representation of the record, and $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(2)}$ are the boolean representations of $e^{(1)}$ and $e^{(2)}$ respectively. The weight w counts the number of variables that are assumed to be incorrect, which at the root node is zero.

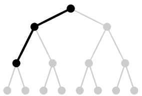
The tree is traversed in depth-first fashion. In the first step, we substitute married in *marital status*, yielding



$\mathbf{v}$	1				1	0	1	0	0
$\mathbf{a}^{(1)}$	1				1	0	1	1	1

Subst. mar. stat., $w = 0$.

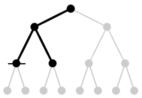
Here, $\mathbf{a}^{(2)}$ is removed, since it has no meaning for $\mathbf{v}$ anymore. The positions for the categories unmarried, widowed and divorced are left empty here to signify that the datamodel has a fixed marital status now. The dark part of the tree on the left shows which nodes have been treated. Continuing we find



v	1				1		1	0	0
$\mathbf{a}^{(1)}$	1				1		1	1	1

Subst. age., $w = 0$.

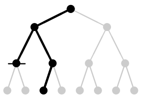
At this point we have fixed the value for *marital status* and *age*. It can be seen from the value of $\mathbf{a}^{(1)}$ for *position in household* that no matter what value is chosen for that field in $\mathbf{v}$, the resulting record will always fall in $\mathbf{a}^{(1)}$. This shows that this path will never lead to a valid solution. We therefore prune the tree here, go up one node and turn right.



v	1						1	0	0

Elim. age, $w = 1$.

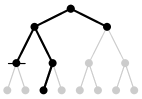
Eliminating the *age* variable yields an empty edit set. We may continue down and substitute the value spouse for *position in household*.



v	1						1		

Subst. hh. pos., $w = 1$.

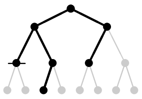
This yields the first solution: only the age variable needs to be changed. In search for more solutions, we move up the tree and try to eliminate *position in household*. However, since eliminating *position in household* would increase the weight to 2 we will prune the tree at this point. Moving up to the root node and eliminating *marital status* gives



v					1	0	1	0	0
$\mathbf{a}^{(3)}$	1	1	1	1	1	0	1	0	0

Elim. mar. stat., $w = 1$.

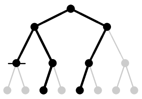
where $\mathbf{a}^{(3)}$ is the derived edit. It is interpreted as the rule that under-aged people cannot be a partner in the household (no matter what the value of *marital status* is). Creating the next child node by substituting *age*, we get



v					1		1	0	0
$\mathbf{a}^{(3)}$	1	1	1	1	1		1	0	0

Subst. age, $w = 1$.

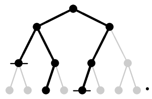
Going down the tree and substituting *position in household* yields



v					1		1		
$\mathbf{a}^{(3)}$	1	1	1	1	1		1		

Subst. hh. pos., $w = 1$.

However, whatever value we would choose for *marital status*, it would always result in an erroneous record since $\mathbf{a}^{(3)}$ has TRUE on all categories of that variable. Therefore, we go up one step in the tree. Eliminating *position in household* would increase the weight to 2, but since we already found a solution with weight equal to 1, this path need not be followed. We go up another node and bound on the fact that eliminating *age* would yield the same problem. The final tree may be represented as follows:



Here, every evaluated node is colored black, and a node is crossed when a bound condition was encountered. The only (minimal) solution created is represented by the path substitute *marital status* → eliminate *age* → substitute *position in household*. This corresponds to the solution where *age* has to be altered to fix the record, and indeed changing *age* from under-aged to adult in Eq. (17) will make the record fully valid. Note that the branch-and-bound algorithm reduced the number of nodes to be evaluated from 15 to 8 in this example.

4.2 Error localization with localizeErrors

The function localizeErrors applies the branch-and-bound algorithm to determine the minimal weight error location for every record in a data.frame. The columns may be in character or factor format. The function has an identical interface for numerical data under linear edits and categorical data under categorical edits. It is implemented as an S3 generic function, accepting either an editmatrix or an editarray as the first argument and a data.frame as the second argument. Further arguments are a vector of variable weights, a maximum search time (in seconds) to spend on a single record, a maximum weight and the maximum number of variables which may be changed. The latter two arguments introduce extra bound conditions in the branch-and-bound algorithm.

Even when variables are weighted, the solution to the error localization problem may not be unique. In those cases localizeErrors will draw uniformly from the set of lowest-weight solutions. The degeneracy (number of equivalent solutions found) is reported in the output.

The result of a call to localizeErrors is an object of class errorLocation. It contains a boolean matrix with error locations for each record as well as a status report containing degeneracies, solution weights run times and whether the maximum runtime was exceeded. It also contains a timestamp (in the form of a Date object) and the name of the user running R. Table 2 gives an overview of the slots involved.

In Figure 2 an example of the use of localizeErrors is given. The data model and rules are as in subsection 4.1. The records are given by

```
> E <- editarray(c(
+     "age %in% c('under-aged','adult')",
+     "maritalStatus %in% c('unmarried','married','widowed','divorced')",
+     "positionInHousehold %in% c('spouse', 'child', 'other')",
+     "if( age == 'under-aged' ) maritalStatus == 'unmarried'",
+     "if(positionInHousehold == 'spouse') maritalStatus == 'married'"
+     )
+ )
> (dat <- data.frame(
+     maritalStatus=c('married','unmarried','widowed' ),
+     age = c('under-aged','adult','adult' ),
+     positionInHousehold=c('child','spouse','other')
+ ))

  maritalStatus        age positionInHousehold
1       married under-aged               child
2     unmarried      adult              spouse
3       widowed      adult               other

> set.seed(1)
> localizeErrors(E,dat)

Object of class 'errorLocation' generated at Mon Nov 28 11:28:18 2011
call : localizeErrors(E, dat)
slots: $adapt $status $call $user $timestamp

Values to adapt:
      adapt
record maritalStatus   age positionInHousehold
     1         FALSE  TRUE               FALSE
     2         FALSE FALSE                TRUE
     3         FALSE FALSE               FALSE

 Status:
  weight degeneracy  user system elapsed maxDurationExceeded
1      1          2 0.008      0   0.007               FALSE
2      1          2 0.008      0   0.008               FALSE
3      0          1 0.004      0   0.002               FALSE
```

Figure 2: Localizing errors in a data.frame of records. The data model is as defined in Eqs. (3)-(5). The randseed is set before calling localizeErrors to make results reproducible. The third record has degeneracy 2, which means that the chosen solution was drawn uniformly from two equivalent solutions with weight 1.

Table 2. Slots in the errorLocation *object*

Slot	description.
$adapt	boolean array, stating which variables must be adapted for each record.
$status	A data.frame, giving solution weights, number of equivalent solutions, timings and whether the maximum search time was exceeded.
$user	Name of user running R during the error localization
$timestamp	date() at the end of the run.
$call	The call to localizeErrors

```
  maritalStatus         age positionInHousehold
1       married under-aged               child
2     unmarried      adult              spouse
3       widowed      adult               other
```

Clearly, the first and third record disobey at least one rule while the second record is valid. The first record can be repaired by adapting age and the second record can be made consistent by changing either *position in household* or *marital status*. In the latter case, both solutions have equal weight and localizeErrors has drawn one solution.

4.3 Error localization with errorLocalizer

The function errorLocalizer gives more control over the error localization process since it allows to parameterize the search separately for each record. This can be useful, for example when reliability weights are calculated for each record. Since errorLocalizer is described extensively in De Jonge and Van der Loo (2011), here we just discuss the example shown in Figure 3.

The data model and edits are defined in Eqs. (3)-(5). For errorLocalizer, a record must be offered as a named character vector. A call to errorLocalizer generates a backtracker object which contains all information necessary to start searching the binary tree. After calling $searchNext() the weight and first found solution are returned, while the backtracker object stores some meta-information about the process, most significantly the duration of the search. The parameter $maxDurationExceeded indicates whether a solution was found within the maximum time allowed for the search. A second call yields an equivalent solution and the third call returns NULL, indicating that all minimal weight solutions have been found. Finally, we note that with $searchAll(), all solutions encountered during the branch-and-bound procedure are returned.

```
> record <- c(
+     age = 'under-aged',
+     maritalStatus='married',
+     positionInHousehold='child'
+ )
> el <- errorLocalizer(E,record)
> el$searchNext()

$w
[1] 1

$adapt
                age       maritalStatus positionInHousehold 
              FALSE                TRUE               FALSE 

> el$duration

   user  system elapsed 
  0.000   0.000   0.003 

> el$maxdurationExceeded

[1] FALSE

> el$searchNext()

$w
[1] 1

$adapt
                age       maritalStatus positionInHousehold 
               TRUE               FALSE               FALSE 

> el$searchNext()

NULL
```

Figure 3: Finding errors with errorLocalizer. The data model and edits in E are as in Figure 2.

5 Conclusions

This paper describes the theory and implementation of categorical edit manipulation in R package editrules. Categorical restrictions may be defined textually in standard R syntax. Edits can be manipulated by variable elimination and value substitution. Many other functionalities such as feasibility checks, block detection and redundancy removal are implemented as well. Moreover, the package offers functionality to check records against rules and can determine the location of errors based on the generalized principle of Fellegi and Holt.

Future work will include performance enhancements of error localization by restating the problem as a mixed-integer problem, the treatment of mixed data and handling interdependent edits.

References

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 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.

Husson, F., S. Lê, and J. Pagès (2010). *Exploratory Multivariate Analysis by Example Using R.* Computer Sciences and Data Analysis. Chapman & Hall/CRC.

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

Winkler, W. E. (1999). State of statistical data editing and current research problems. In *Working paper no. 29. UN/ECE Work Session on Statistical Data editing*, Rome.

Index

boolean representation
  of edits, 7
  of records, 6

data model, 6
domain, 6

editarray, 7
  feasibility, 13
  functions, 11
  redundancy, 13
  value substitution, 14
  variable elimination, 16
error localization
  with errorLocalizer, 22
  with localizeErrors, 20