

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

package version 1.6.0

Mark van der Loo and Edwin de Jonge

October 6, 2011

Abstract

*This vignette is far from finished. Version 2.0 fo the package will have the full vignette. At the moment, **functionality for treating categorical data has beta status** so bugs are likely.*

Refer to the accompanying paper De Jonge and Van der Loo (2011) for manipulation of linear edits.

Contents

1	Introduction	3
2	Defining and checking categorical constraints	4
2.1	Boolean representation of records and edits	4
2.2	The <code>editarray</code> object	6
2.3	Coercion, checking, redundancy and infeasibility	8
3	Manipulation of categorical restrictions	10
3.1	Value substitution	10
3.2	Variable elimination by category resolution	11
3.3	An example with <code>eliminate</code>	12
4	Error localization in categorical data	12
4.1	A Branch and bound algorithm	12
4.2	Error localization with <code>localizeErrors</code>	15
4.3	Error localization with <code>errorLocalizer</code>	16
5	Conclusions	16
A	Notation	18

List of Algorithms

1	<code>ISSUBSET(E)</code>	9
2	<code>SUBSTVALUE(E, k, v)</code>	10
3	<code>ELIMINATE(E, k)</code>	12

Reading guide. This paper contains a fair amount of background on the algorithms and methods behind the `editrules` package. Readers who want to get started without going through the theory can read the following subsections: $1 \rightarrow 2.2 \rightarrow 2.3 \rightarrow 3.1 \rightarrow 3.4 \rightarrow 4.2$.

1 Introduction

The value domain of categorical data records is usually limited by domain 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.

A categorical data record r with n variables may be defined as an element of the cartesian product space D :

$$D = D_1 \times D_2 \times \dots \times D_n, \quad (1)$$

where each D_k is a finite set of d_k possible categories for variable k . We label the categories as follows:

$$D_k = \{c \in D_k \mid c = 1, 2, \dots, d_k\}. \quad (2)$$

Each restriction e is a subset of D and we say that that *if* $r \in e$ *then* r *violates* e . Conversely, when $r \notin e$ we say that r *satisfies* e . In data editing literature, such rules are referred to as *edit rules* or *edits*, in short. In the context of contingency tables they are referred to as *structural zeros* since each rule implies that one or more cells in the $d_1 \times d_2 \times \dots \times d_n$ contingency table must be zero. A record is *valid* if it satisfies every edit imposed on D .

Large, complex surveys may consist of hundreds of interrelated rules and variables, which impedes resolution of edit violations and renders manual manipulation infeasible. Winkler (1999) mentions practical cases where statistical offices handle 250, 300 or 750 categorical edit rules for surveys.

The R package `editrules` offers functionality to define, manipulate and maintain sets of edit rules with relative ease. It also implements error localization functionality based on the generalized principle of (Fellegi and Holt, 1976), which states that one should find the smallest (weighted) number of variables whose values can be adapted such that all edits can be satisfied. Fellegi and Holt’s principle should be considered as the last resort of data editing. It is useful in situations where a record violates one or more edits and there is no information about the cause of the error. In certain cases, the cause of error can be estimated with near certainty, for example in the case of typing errors in numerical data. We refer the reader to Scholtus (2008, 2009) and Van der Loo et al. (2011) for such cases.

The purpose of this paper is to give a technical overview of the representation and manipulation of edits in the `editrules` package, as well as some coded examples to get new users started.

2 Defining and checking categorical constraints

In the next section we describe the representation of edits and records as implemented in the `editrules` package. Readers not interested in the underlying principles may skip Section 2.1.

2.1 Boolean representation of records and edits

Categorical records may be represented as a vector of boolean values. A boolean vector of dimension d is an element of the boolean algebra

$$\mathbb{B}^d = \left(\{0, 1\}^d, \wedge, \vee, \neg \right), \quad (3)$$

where 0 and 1 have the usual interpretations FALSE and TRUE and the logical operators work elementwise on their operands. To facilitate the discussion we will also allow the standard arithmetic operations addition and subtraction on boolean vectors (this is also consistent with the way R handles vectors of class `logical`).

To represent a record $r = (r_1, r_2, \dots, r_n)$, assign to every category c in D_k a unique standard basisvector $\vec{\delta}_k(c)$ of \mathbb{B}^{d_k} . The boolean representation $\rho(r)$ of the full record is the direct sum

$$r \xrightarrow{\rho} \vec{\delta}_1(r_1) \oplus \vec{\delta}_2(r_2) \oplus \dots \oplus \vec{\delta}_n(r_n), \quad (4)$$

which we will write as

$$\rho(r) = \mathbf{v}_1 \oplus \mathbf{v}_2 \oplus \dots \oplus \mathbf{v}_n \equiv \mathbf{v}. \quad (5)$$

The dimension d of $\rho(r)$ is given by the total number of categories of all variables

$$d = \sum_{k=1}^n d_k. \quad (6)$$

When each record in a dataset is represented this way, summing the vectors yields the d -dimensional vectorized representation of the $d_1 \times d_2 \times \dots \times d_n$ contingency table of the dataset. This is sometimes called the complete disjunctive table.

Example 2.1. *Consider the variables marital status, age, and position in household from the domain $D = D_1 \times D_2 \times D_3$. We define*

$$D_1 = \{\text{married, unmarried, widowed, divorced}\} \quad (7)$$

$$D_2 = \{\text{underaged, adult}\} \quad (8)$$

$$D_3 = \{\text{partner, child, other}\}. \quad (9)$$

The record $r = (\text{married, adult, partner})$ has boolean representation

$$\rho(r) = (1, 0, 0, 0) \oplus (0, 1) \oplus (1, 0, 0) = (1, 0, 0, 0, 0, 1, 1, 0, 0). \quad (10)$$

An edit e is a subset of D which can be written as the cartesian product

$$e = A_1 \times A_2 \times \dots \times A_n, \text{ where } A_k \subseteq D_k, k = 1, 2, \dots, n. \quad (11)$$

The interpretation of an edit is that if a record $\mathbf{v} \in e$, then \mathbf{v} is considered invalid. The following properties follow immediately.

Remark 2.2. *If $e \subset D$ and $e' \subset D$ are edits, then $e \cup e' = \{e, e'\}$ and $e \cap e' = A_1 \cap A'_1 \times A_2 \cap A'_2 \times \dots \times A_n \cap A'_n$ are also edits.*

An edit, expressed as in Eq. (11) is said to be in normal form. A variable v_k is *involved* in an edit if $A_k \subset D_k$. Conversely, we say that e *involves* v_k if v_k is involved in e . A variable v_k for which $A_k = D_k$ is not involved in e . Since every category i of D_k is mapped to a unique basis vector $\vec{\delta}_k(i)$, edits have a boolean representation $\rho(e)$, given by

$$e \xrightarrow{\rho} \bigvee_{i \in A_1} \vec{\delta}_1(i) \oplus \bigvee_{i \in A_2} \vec{\delta}_2(i) \oplus \dots \oplus \bigvee_{i \in A_n} \vec{\delta}_n(i), \quad (12)$$

which may simply be written as

$$\rho(e) = \mathbf{a}_1 \oplus \mathbf{a}_2 \oplus \dots \oplus \mathbf{a}_n \equiv \mathbf{a}. \quad (13)$$

Example 2.3. *Using the data model from Example 2.1, the edit that says that under aged people cannot be married has set representation*

$$e = \{\text{married}\} \times \{\text{under aged}\} \times \{\text{partner, child, other}\}, \quad (14)$$

which translates to the boolean representation

$$\rho(e) = (1, 0, 0, 0) \oplus (0, 1) \oplus (1, 1, 1) = (1, 0, 0, 0, 0, 1, 1, 1, 1). \quad (15)$$

It is easy to check that an edit contains variable k if and only if the inner product $\mathbf{1}_{d_k} \cdot \mathbf{a}_k < d_k$, where $\mathbf{1}_{d_k}$ is a d_k vector of ones.

A record \mathbf{v} violates an edit if $v_k \in A_k$, $k = 1, 2, \dots, n$. In the boolean representation this can be written as a condition on the standard inner product between the boolean representation of a record and an edit:

$$\sum_{k=1}^n \vec{\delta}_k(v_k) \cdot \mathbf{a}_k = \rho(\mathbf{v}) \cdot \mathbf{a} = n. \quad (16)$$

Suppose that E is a set of edits of the form (11). It is not difficult to verify that an edit $e \in E$ is redundant if

$$A_k = \emptyset, \text{ for any } k \in 1, 2, \dots, n \quad (17)$$

or

$$e \subseteq e' \text{ with } e' \in E. \quad (18)$$

In (17), e is redundant since it cannot contain any records. It can be tested by checking if any $\mathbf{1}_{d_k} \cdot \mathbf{a}_k = 0$. In the case of (18), e is redundant because any edit violating e also violates e' . Using $\rho(e) = \mathbf{a}$ and $\rho(e') = \mathbf{a}'$, this can be tested by checking if $\mathbf{a} \wedge \mathbf{a}' = \mathbf{a}$ or equivalently if $\mathbf{a} \vee \mathbf{a}' = \mathbf{a}'$.

In the `editrules` the boolean representation is mainly used to store edits and to manipulate them with methods like variable substitution and elimination. Data records can be stored in `data.frame` objects, as usual.

2.2 The editarray object

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

$$E = \langle \mathbf{A}, \mathbf{ind} \rangle, \text{ with } \mathbf{A} \in \{0, 1\}^{m \times d} \text{ and } d = \sum_{k=1}^n d_k, \quad (19)$$

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 `ind` is a nested list which relates columns of \mathbf{A} to variable names and categories. Labeling variables with $k \in 1, 2, \dots, n$ and category values with $c \in 1, 2, \dots, d_k$, we use the following notations:

$$\mathbf{ind}(k, c) = \sum_{l < k} d_l + c \quad (20)$$

$$\mathbf{ind}(k) = \{\mathbf{ind}(k, c) \mid c \in D_k\}. \quad (21)$$

So $\mathbf{ind}(k, c)$ is the column index in \mathbf{A} for variable k and category c and $\mathbf{ind}(k)$ is the set of column indices corresponding to the 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.

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 male subject cannot be pregnant”, or “an under-aged subject cannot be married”. To facilitate the definition of edit arrays, `editrules` is equipped with a parser, which takes R-statements in `character` format, and translates them to an `editarray`.

Figure 1 shows a simple example of defining an `editmatrix` with the `editrules` package. The first two edits in Figure 1 define the data model. 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.

```

> 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.

The function `datamodel` accepts an edit array as input and returns an overview of variables and their categories for easy inspection. 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`.

Internally, `editarray` uses the R internal `parse` function to transform the character expressions to a parse tree, which is subsequently traversed recursively to derive the entries of the editmatrix. The opposite is also possible. 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.

The character expressions that can be read by `editarray`, such as

"if (gender == 'male') pregnant = 'no'"

follows standard R syntax, which should be already familiar to the reader. Note that double quotes are used to enclose the whole string, while single

Table 1: Functions and operators that may be used to define edits with editarray

Operator	Description
%in%	Set membership*
==	Equality*
if(<condition>) <expression>	conditional statement
c(' <cat1> ', ' <cat2> ', ...)	categories, character or logical
&&, &	logical AND
,	logical OR
!	logical NOT

*Right-hand side is evaluated.

quotes are used for category names. Table 1 shows which operators and functions can be used to specify categorical edit rules.

Categories may be literal characters, or booleans. It is worth noting that expressions on the right hand side of the %in% and == operators are evaluated. One useful application of this is that the categories, or data model can be defined outside of the edits:

```
> xval <- letters[1:4]
> yval <- c(TRUE,FALSE)
> 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:FALSE 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 above example also illustrates the use of boolean categories.

2.3 Coercion, checking, redundancy and infeasibility

Table 2 lists basic functions of editarray objects. The `datamodel` function retrieves the variables and categories in an edit array, 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 used when edits are printed to the screen. Optionally, coercing the `datamodel` to character form can be switched off. The result of `as.data.frame` version contains 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

Table 2: Basic functions for objects of class `editarray`. Only mandatory arguments are shown, refer to the built-in documentation for optional arguments.

Function	description
<code>datamodel(E)</code>	get <code>datamodel</code>
<code>getVars(E)</code>	get a list of variables
<code>as.data.frame(E)</code>	coerce edits to <code>data.frame</code>
<code>contains(E)</code>	which edits contains which variable
<code>as.character(E)</code>	coerce edits to <code>character</code> vector
<code>violatedEdits(E,x)</code>	check which edits are violated by <code>x</code>
<code>isObviouslyRedundant(E)</code>	find redundancies [Eq. (17)], duplicates
<code>duplicated(E)</code>	find duplicate edits
<code>isSubset(E)</code>	find edits, subset of another edit [Eq. (18)]
<code>isObviouslyInfeasible(E)</code>	detect contradictions
<code>substValue(E,var,value)</code>	substitute a value

Algorithm 1 `ISSUBSET(E)`

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

- 1: $\mathbf{s} \leftarrow (\text{FALSE})^m$
- 2: **for** $(\mathbf{a}^{(i)}, \mathbf{a}^{(i')}) \in \text{rows}(\mathbf{A}) \times \text{rows}(\mathbf{A})$ **do**
- 3: **if** $\mathbf{a}^{(i)} \vee \mathbf{a}^{(i')} = \mathbf{a}^{(i')}$ **then**
- 4: $s_i \leftarrow \text{TRUE}$

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

edits (columns). It works by parsing the `editarray` to R-expressions and evaluating them 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, redundant edits of the form of Eq. (17) may arise. Such redundancies can be detected in the boolean representation with `isObviouslyRedundant`. By default, this function also checks for duplicate edits, but this may be turned off. The function `duplicated` is overloaded from the standard R function and the function `isSubset` (pseudocode in Algorithm 1) detects which edits are a subset or duplicate of another one. 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. They can be detected by checking if $\sum_{j=1}^d \rho(e)_j = d$.

Algorithm 2 SUBSTVALUE(E, k, v)

Input: an editarray $E = \langle \mathbf{A}, \mathbf{ind} \rangle$, a variable index k and a value v

1: $i \leftarrow \mathbf{ind}(k, v)$

2: $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\mathbf{a} \in \text{rows}(\mathbf{A}) \mid a_i = \text{FALSE}\}$ \triangleright Remove rows not involving v

3: $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\mathbf{a}_j^t \in \text{columns}(\mathbf{A}) \mid j \in \mathbf{ind}(k) \setminus i\}$ \triangleright Remove categories $\neq v$

4: Update \mathbf{ind}

Output: $\langle \mathbf{A}, \mathbf{ind} \rangle$ with v substituted for variable k .

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. In subsection 3.2 we describe an elimination method, which appears to be new to the field of data editing. In subsection ?? it is shown that the method yields results equivalent to Fellegi and Holt's elimination method while running in polynomial instead of exponential time. Finally, in subsection ?? we give an example of eliminating variables with the `editrules` package. Readers not interested in the technical background can skip subsections 3.2 and ??.

3.1 Value substitution

If it is assumed that in a record, one of the variables takes a certain 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, thus 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 variable and category names. In the following example the editmatrix defined in Figure 1 is used.

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

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

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

In this case, the variable “gender” is substituted by the value “female”. With the gender is fixed, the datamodel reduces to $\{\text{male}\} \times \{\text{no}, \text{yes}\}$ and the restriction “if male then pregnant = true” becomes meaningless and is therefore removed.

The R implementation of `substValue` has an extra option, allowing to choose if the datamodel is reduced or not, which by default is set to `TRUE`.

3.2 Variable elimination by category resolution

Given two edits e and e' , with boolean representations \mathbf{a} and \mathbf{a}' respectively. We define the *resolution operator* \mathfrak{R}_k as:

$$\begin{aligned} \mathbf{a} \mathfrak{R}_k \mathbf{a}' &= \mathbf{a}_1 \wedge \mathbf{a}'_1 \oplus \dots \oplus \mathbf{a}_{k-1} \wedge \mathbf{a}'_{k-1} \\ &\oplus \mathbf{a}_k \vee \mathbf{a}'_k \oplus \mathbf{a}_{k+1} \wedge \mathbf{a}'_{k+1} \oplus \dots \oplus \mathbf{a}_n \wedge \mathbf{a}'_n \end{aligned} \quad (22)$$

For two edit sets \mathbf{A} and \mathbf{A}' , we also introduce the notation

$$\mathbf{A} \mathfrak{R}_k \mathbf{A}' = \{\mathbf{a} \mathfrak{R}_k \mathbf{a}' \mid (\mathbf{a}, \mathbf{a}') \in \text{rows}(\mathbf{A}) \times \text{rows}(\mathbf{A}')\}. \quad (23)$$

Observe that the resolution operator has the following properties, relevant for record checking.

$$\rho(\mathbf{v}) \in \mathbf{a} \mathfrak{R}_k \mathbf{a}' \Rightarrow \rho(\mathbf{v}) \in \mathbf{a} \vee \rho(\mathbf{v}) \in \mathbf{a}' \quad (24)$$

$$\rho(\mathbf{v}) \in \mathbf{a} \Rightarrow \rho(\mathbf{v}) \in \mathbf{a} \mathfrak{R}_k \mathbf{a}' \vee \mathbf{a} \mathfrak{R}_k \mathbf{a}' = \emptyset \quad (25)$$

That is, if a record violates $\mathbf{a} \mathfrak{R}_k \mathbf{a}'$, it does so because it violates \mathbf{a} and/or \mathbf{a}' . Therefore, $\mathbf{a} \mathfrak{R}_k \mathbf{a}'$ is also an edit in the sense that a record is invalid if it falls in the derived edit. When $\mathbf{a}_k = \mathbf{a}'_k$, the resulting edit is the intersection of the original edits, in which case the resulting edit is redundant.

The operator is called resolution operator since its action strongly resembles a resolution operation from formal logic. If $\mathbf{a}_k \vee \mathbf{a}'_k = (\text{TRUE})^{d_k}$, the operator “resolves” or eliminates the k^{th} variable and we are left with a relation between the other variables, regardless of the value of variable k . The edit resulting from a resolution operation on two explicitly defined edits is called an *implied edit*. If the resolution operation happens to eliminate one of the variables, it is called an *essentially new implied edit*. These terms were introduced by Fellegi and Holt (1976) who first solved the problem of error localization for categorical data.

The resolution operator can be used to eliminate a variable k from a set of edits represented by \mathbf{A} category by category as follows (Algorithm 3). Suppose that j is the column index of the first category of k . Collect all pairs of $(\mathbf{a}^+, \mathbf{a}^-)$ obeying $a_j^+ = \text{TRUE}$ and $a_j^- = \text{FALSE}$. If there are no edits of type \mathbf{a}^+ , the variable cannot be eliminated and the empty set is returned. Otherwise, copy all \mathbf{a}^+ to a new set of edits and add every $\mathbf{a}^+ \mathfrak{R}_k \mathbf{a}^-$. By construction, these new edits all have $a_j = \text{TRUE}$. Possibly, redundant edits have been produced, and these may be removed. The procedure is iterated

Algorithm 3 ELIMINATE(E, k)

Input: an editarray $E = \langle \mathbf{A}, \mathbf{ind} \rangle$, a variable index k

```
1: for  $j \in \mathbf{ind}(k)$  do  
2:    $\mathbf{A}^+ = \{\mathbf{a} \in \text{rows}(\mathbf{A}) : a_j = \text{TRUE}\}$   
3:    $\mathbf{A}^- = \{\mathbf{a} \in \text{rows}(\mathbf{A}) : a_j = \text{FALSE}\}$   
4:   if  $\mathbf{A}^+ = \emptyset$  then  
5:      $\mathbf{A} \leftarrow \emptyset$   
6:     break  
7:    $\mathbf{A} \leftarrow \mathbf{A}^+ \cup \mathbf{A}^+ \mathfrak{R}_k \mathbf{A}^-$   
8:   Delete rows which have  $\text{ISUBSET}(\langle \mathbf{A}, \mathbf{ind} \rangle) = \text{TRUE}$ .
```

Output: editarray $\langle \mathbf{A}, \mathbf{ind} \rangle$ with variable k eliminated

for every category of k , adding a category for which each $a_j = \text{TRUE}$ at each iteration.

3.3 An example with eliminate

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 than solutions found earlier.

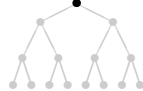
Recall the datamodel of Example 2.1, with variables *marital status*, *age* and *position in household*. We define the following two edits:

- $e^{(1)}$ under aged people cannot be married
- $e^{(2)}$ If you are not married you cannot be a partner

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

$$r = (\text{married}, \text{under aged}, \text{partner}). \quad (26)$$

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




\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 $\mathbf{v} = \rho(r)$, 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 generated in depth-first fashion. In the first step, we substitute r_1 , yielding



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

Substitute v_1 , $w = 0$.


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



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

Substitute v_2 , $w = 0$.

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, the value $\mathbf{v} \cdot \mathbf{a} = 3$. Which 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
-----	---	--	--	---	---	---

Eliminate v_2 , $w = 1$.

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



\mathbf{v}	1		1

Substitute v_3 , $w = 1$.

Which 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 *age*. However, since eliminating age 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



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

Eliminate v_1 , $w = 1$.

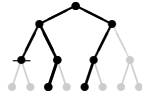
Here $\mathbf{a}^{(3)} = \mathbf{a}^{(1)} \Re_1 \mathbf{a}^{(2)}$. It is interpreted as the rule that under-aged people cannot be a partner in the household (no matter what the marital status is). Creating the next child node by substituting *age*, we get



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

Substitute v_2 , $w = 1$.

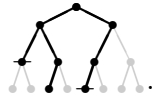
Going down the tree and substituting *position in household* yields



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

Substitute v_3 , $w = 1$.

However, whatever value we would choose for the marital status variable, it would always result in an erroneous record (since $\mathbf{a}^{(3)}$ has TRUE on all categories). Therefore, we go up one step in the tree. Eliminating *age* 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 *position in household* 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* \rightarrow eliminate *age* \rightarrow 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* will make the full record valid. Note that the branch-and-bound algorithm reduced the number of nodes to be evaluated from 15 to 8 in this case.

Table 3: Slots in the `errorLocation` object

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

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 `errorLocalization`. It contains a boolean matrix with error locations for each record as well as a status report containing degeneracies, solution weights runtimes 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 ?? 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

	maritalStatus	age	positionInHousehold
1	married	under aged	child
2	unmarried	adult	other
3	widowed	adult	marriage partner

The edits state that under aged cannot be married and that you cannot be a marriage partner if you're not married. Clearly, the first and third record disobey some rules of these rules while the second record is valid. The first

```

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

  maritalStatus      age positionInHousehold
1    married under aged             child
2   unmarried    adult             other
3    widowed    adult  marriage partner

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

Object of class 'errorLocation' generated at Thu Oct  6 14:44:57 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             FALSE
  3      FALSE FALSE             TRUE

Status:
  weight degeneracy user system elapsed maxDurationExceeded
1      1          1    0      0    0.002                FALSE
2      0          1    0      0    0.001                FALSE
3      1          2    0      0    0.005                FALSE

```

Figure 2: Localizing errors in a data.frame of records. The data model is as defined in Example 2.1. 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.

record can be repaired by adapting age and the second record can be made consistent by changing either *positin 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

5 Conclusions

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*. Wiley handbooks in survey methodology. Hoboken, New Jersey: 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 American Statistical Association* 71, 17–35.
- Scholtus, S. (2008). Algorithms for correcting some obvious inconsistencies and rounding errors in business survey data. Technical Report 08015, Statistics Netherlands, Den Haag. The papers are available in the inst/doc directory of the R package or via the website of Statistics Netherlands.
- Scholtus, S. (2009). Automatic correction of simple typing error in numerical data with balance edits. Technical Report 09046, Statistics Netherlands, Den Haag. The papers are available in the inst/doc directory of the R package or via the website of Statistics Netherlands.
- Van der Loo, M., E. de Jonge, and S. Scholtus (2011). *deducorrect: Deductive correction of simple rounding, typing and sign errors*. R package version 1.0-1.
- Willenborg, L. (1988). *Computational aspects of data processing*, Volume 54 of *CWI Tract*. Amsterdam: Centre for Mathematics and Computer Science.
- 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.

A Notation

Symbol	Explanation and reference
\mathbf{a}	An edit, in boolean representation: $\mathbf{a} = \rho(e)$, Eq. (13).
\mathbf{a}_k	Boolean representation of one variable in \mathbf{a} . $\mathbf{a} = \oplus_{k=1}^n \mathbf{a}_k$.
\mathbf{A}	Set of edits, in $m \times d$ boolean representation.
c	Label for a single category of D_k .
D	Set (domain) of all possible categorical records, Eq. (1).
D_k	Set of possible categories for variable k . Eq. (2).
d	Number of categories (in total), Eq. (6).
d_k	Number of categories in D_k .
e	An edit, in set representation: $e \subseteq D$, [Eq. (11)].
E	An editarray, Eq. (19), or a set of edits in set representation.
\mathbf{ind}	Function relating categories c of variable k to columns in \mathbf{A} , Eqs.(20) and (21).
i	row index in \mathbf{A} (labeling edits).
j	column index in \mathbf{A} (labeling categories).
m	Number of edits.
n	Number of variables.
r	Categorical record, in set representation: $r \in D$.
\mathfrak{R}_k	Resolution operator Eq. (22).
ρ	Map, sending set representation to boolean representation.
\mathbf{v}	Categorical record, in boolean representation: $\mathbf{v} = \rho(r)$.
\mathbf{v}_k	Boolean representation of a single variable $\mathbf{v} = \oplus_{k=1}^n \mathbf{v}_k$.