Manipulation of linear edits and error localization with the editrules package

Package version 2.1.2

Edwin de Jonge and Mark van der Loo

February 8, 2012

Abstract

This paper is the first of two papers describing the editrules package. The current paper is concerned with the treatment of numerical data under linear constraints, while the accompanying paper (Van der Loo and De Jonge, 2011) is concerned with constrained categorical and mixed data. The editrules package is designed to offer a user-friendly interface for edit definition, manipulation and checking. The package offers functionality for error localization based on the paradigm of Fellegi and Holt and a flexible interface to binary programming based on the choice point paradigm. Lower-level functions include echelon transformation of linear systems, variable substitution and a fast Fourier-Motzkin elimination routine. We describe theory, implementation and give examples of package usage.

This vignette is a near-literal transcript of De Jonge and Van der Loo (2011). The vignette is a living document which will be updated with the package while the reference corresponds to version 1.0.

Contents

List of Algorithms

1 Introduction

The value domain of real numerical data records with n variables is often restricted to a subdomain of $\mathbb{R}^n$ due to linear equality and inequality relations which the variables in the records have to obey. Examples include equality restrictions imposed by financial balance accounts, positivity demands on certain variables or limits on the ratio of variables.

Any such restriction can be written in the form

$$\mathbf{a} \cdot \mathbf{x} \odot b \text{ with } \odot \in \{<, \leq, =\}, \tag{1}$$

where $\mathbf{x}$ is a numerical data record, $\mathbf{a}$, $\mathbf{x} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. In data editing literature, data restriction rules are referred to as *edits*, or *edit rules*. In this paper we will call edits, written in the form of Eq. (1) (specifically, without using $\geq$ or $>$), edits in *normal form*.

Large, complex surveys are often endowed with dozens or even hundreds of edit rules. For example, the Dutch Structural Business Survey, which aims to report on the financial structure of companies in the Netherlands, contains about 100 variables, and has a similar number of linear equality and inequality restrictions involving multiple variables, as well as many univariate positivity constraints.

Defining and manipulating large edit sets in matrix representation is a daunting task, because it may involve hundreds of rows and columns. This is also true for finding which variables in a record are responsible for edit violations: the so-called error localization problem.

The editrules package for the R statistical computing environment (R Development Core Team, 2011) aims to provide an environment to conveniently define, read and check linear (in)equality restrictions, perform common edit manipulations and offer error localization functionality based on the (generalized) paradigm of Fellegi and Holt (1976). This paradigm assumes that errors are distributed randomly over the variables and there is no detectable cause of error. It also decouples the detection of corrupt variables from their correction.

For some types of error, such as sign flips, typing errors or rounding errors, this assumption does not hold. The cause of these errors can be detected and are closely related to their resolution. The reader is referred to the deducorrect package (Van der Loo et al., 2011; Scholtus, 2008, 2009) for treating such errors.

The following chapters demonstrate the functionality of the editrules package with coded examples as well a description of the underlying theory and algorithms. For a detailed per-function description the reader is referred to the reference manual accompanying the package. Unless mentioned otherwise, all code shown in this paper can be executed from the R command line after loading the editrules package.

2 Defining and checking numerical restrictions

2.1 The editmatrix object

For computational processing, a general set of edits of the form

$$\mathbf{a} \cdot \mathbf{x} \odot b \text{ with } \odot \in \{<, \leq, =, \geq, >\}, \tag{2}$$

is most conveniently represented as a matrix. In the editrules package, a set of linear edits is stored as an editmatrix object. This object stores the linear relations as an augmented matrix $[\mathbf{A}, \mathbf{b}]$, where $\mathbf{A}$ is the matrix obtained by combining the $\mathbf{a}$ vectors of Eq. (2) in rows of $\mathbf{A}$ and constants b in $\mathbf{b}$. A second attribute holds the comparison operators as a character vector. Formally, we denote that every editmatrix E is defined by

$$E = \langle [\mathbf{A}|\mathbf{b}], \odot \rangle \text{ with } [\mathbf{A}|\mathbf{b}] \in \mathbb{R}^{m \times (n+1)}, \odot \in \{<, \leq, =, \geq, >\}^m, \tag{3}$$

where n is the number of variables, m the number of edit rules and the notation $\langle\ ,\ \rangle$ denotes a combination of objects. Retrieval functions for various parts of an editmatrix are available, see Table 1 (page 7) for an overview. Defining augmented matrices by hand is tedious and prone to error, which is why the editmatrix function derives edit matrices from a textual representation of edit rules. Since most functions of the editrules package expect an editmatrix in normal form (that is, $\odot \in \{<, \leq, =\}^m$), the editmatrix function by default transforms all linear edits to normal form.

As an example, consider the set of variables

turnover	t
personnel cost	c_p
housing cost	c_h
total cost	c_t
profit	p,

subject to the rules

$$\begin{aligned}
t &= c_t + p && (4)\\
c_t &= c_h + c_p && (5)\\
p &\leq 0.6t && (6)\\
c_t &\leq 0.3t && (7)\\
c_p &\leq 0.3t && (8)\\
t &> 0 && (9)\\
c_h &> 0 && (10)\\
c_p &> 0 && (11)\\
c_t &> 0. && (12)
\end{aligned}$$

Clearly, these can be written in the form of Eq. (1). Here, the equality restrictions correspond to balance accounts, the 3rd, 4th and 5th restrictions are sanity checks and the last four edits demand positivity. Figure 1 shows how these edit rules can be transformed from a textual representation to a matrix representation with the editmatrix function. To define an editmatrix, edit restrictions can be defined in usual R syntax, using == as comparison operator for equalities and <, <=, >= or > for inequalities. Coefficients may be negative or positive, and both the binary + and − operator are recognized.

As Figure 1 shows, the editmatrix object is shown on the console as a matrix, as well as a set of textual edit rules. The editrules package is capable of coercing a set of R expressions to an editmatrix and *vice versa*. To coerce text to a matrix, the editmatrix function processes the R language parse tree of the textual R expressions as provided by the R internal parse function. To coerce the matrix representation to textual representation, an R character string is derived from the matrix which can be parsed to a language object. In the example, the edits were automatically named e1, e2, ..., e9.

It is also possible to name and comment edits by reading them from a data.frame. The ability to read edit sets from a data.frame facilitates defining and maintaining the rules outside of the R environment by storing them in a user-filled database or text file. Manipulating and combining edits, for example through variable elimination methods will cause editrules to drop or change the names and drop the comments, as they become meaningless after certain manipulations.

2.2 Basic manipulations and edit checking

Table 1 shows simple manipulation functions available for an editmatrix. Basic manipulations include retrieval functions for the augmented matrix, coefficient matrix, constant vector and operators of an editmatrix. There are also functions to test for and transform to normality.

When groups of editrules are unrelated, that is, if they do not share any variables, the edit matrix can be decomposed as

$$E = E_1 \oplus E_2 \oplus \ldots \oplus E_k, \tag{13}$$

where the E_j are mutually independent edit matrices and $\oplus$ is the direct sum operator. The function blocks expects an editmatrix and returns a list of independent edit matrices composing the original one. Splitting an edit matrix into independent blocks can yield a significant speedup in error localization problems.

The function violatedEdits expects an editmatrix and a data.frame or a named numeric vector. It returns a logical array where every row indicates which edits are violated (TRUE) by records in the data.frame. It has an optional argument tol, (default: square root of machine precision) which can

```
> E <- editmatrix(c(
+ "t  == ct + p" ,
+ "ct == ch + cp",
+ "p  <= 0.6*t",
+ "ct <= 0.3*t",
+ "cp <= 0.3*t",
+ "t  >  0",
+ "ch >  0",
+ "cp >  0",
+ "ct >  0"), normalize=TRUE)
> E

Edit matrix:
     ct  p    t ch cp Ops CONSTANT
num1 -1 -1  1.0  0  0  ==        0
num2  1  0  0.0 -1 -1  ==        0
num3  0  1 -0.6  0  0  <=        0
num4  1  0 -0.3  0  0  <=        0
num5  0  0 -0.3  0  1  <=        0
num6  0  0 -1.0  0  0   <        0
num7  0  0  0.0 -1  0   <        0
num8  0  0  0.0  0 -1   <        0
num9 -1  0  0.0  0  0   <        0

Edit rules:
num1 : t == ct + p
num2 : ct == ch + cp
num3 : p <= 0.6*t
num4 : ct <= 0.3*t
num5 : cp <= 0.3*t
num6 : 0 < t
num7 : 0 < ch
num8 : 0 < cp
num9 : 0 < ct
```

Figure 1: Defining an editmatrix from a character vector containing verbose edit statements. The option normalize=TRUE ensures that all comparison operators are either $<$, $\leq$ or $==$.

be increased to ignore rounding errors. Figure 3 demonstrates the result of checking two records against the edit rules defined in Eqs. (4)–(12). Indexing of edits with the [operator is restricted to selection only.

2.3 Obvious redundancy and infeasibility

After manipulating a linear edit set by value substitution and/or variable elimination, it can contain redundant edits or become infeasible. The editrules package has two methods available which check for easily detectable

Table 1: Simple manipulation functions for objects of class editmatrix. Only the mandatory arguments are shown, refer to the built-in documentation for optional arguments.

function	description
getA(E)	Get matrix $\mathbf{A}$
getb(E)	Get constant vector $\mathbf{b}$
getAb(E)	Get augmented matrix $[\mathbf{A}, \mathbf{b}]$
getOps(E)	Get comparison operators
E[i,]	Select edit(s)
as.editmatrix(A,b,ops)	Create an editmatrix from its attributes
normalize(E)	Transform E to normal form
isNormalized(E)	Check whether E is in normal form
violatedEdits(E, x)	Check which edits are violated by $\mathbf{x}$
duplicated(E)	Check for duplicates in rows of E
isObviouslyRedundant(E)	Check for tautologies and duplicates in E
isObviouslyInfeasible(E)	Check for contradictions in rows of E
isFeasible(E)	Complete feasibility check for E
blocks(E)	Decompose E in independent blocks
summary(E)	Summarize the contents of E

redundancies or infeasibility. The Fourier-Motzkin elimination method has auxiliary built-in redundancy removal, which is described in Section 3.3.

A system of inequalities $\mathbf{Ax} \leq \mathbf{b}$ is called infeasible or overconstraint when there is no real vector $\mathbf{x}$ satisfying it. It is a consequence of Farkas' lemma (Farkas (1902), but see Schrijver (1998) and/or Kuhn (1956)) on feasibility of systems of linear equalities, that a system is infeasible if and only if $0 \leq -1$ can be derived by taking positive linear combinations of the rows of the augmented matrix $[\mathbf{A}, \mathbf{b}]$.

The function isFeasible eliminates variables one by one using Fourier-Motzkin elimination (Section 3.3), and checks if such infeasible rules arise. If none are found after the last variable has been eliminated, the system is feasible. This function is useful in checking the feasibility of large sets of edits, which may contain contradictory edits after maintenance.

A complete feasibility check is as computationally expensive as solving a system of inequalities. Therefore, the function isObviouslyInfeasible was written to perform a quick check on obvious inconsistent rules in an editmatrix. It returns a logical indicating whether an obvious contradiction of the form $0 < -1$ or $0 = 1$ is present in an editmatrix. The latter inconsistency can be caused by substitution of values in the edit matrix. Algorithm 1 gives the pseudo-code for reference purposes.

Both value substitution and variable elimination derive new edits, that may be of the form $0 \leq 1$ or $0 = 0$. The function isObviouslyRedundant detects such rules and returns a logical vector indicating which rows of an

editmatrix are redundant. By default, the function detects row duplicates within an adjustable tolerance, but this may be switched of by providing the option duplicates=FALSE. Pseudo-code is given in Algorithm 2. The actual implementation avoids explicit loops and makes use of R's built-in duplicated function, which is also overloaded for editmatrix (see Table 1).

3 Manipulation of linear restrictions

There are two fundamental operations possible on edit sets, both of which reduce the number of variables involved in the edit set. The first, most simple one is to substitute a variable with a value. The second possibility is variable elimination. For a set of linear inequalities, one can apply Fourier-Motzkin elimination to eliminate a variable. The package also has functionality to rewrite systems of equalities in echelon form. Table 2 (page 10) gives an overview.

3.1 Value substitution

Given a set of m linear edits as defined in Eq. (3). For any record $\mathbf{x}$ it must hold that

$$\mathbf{A}\mathbf{x} \odot \mathbf{b}, \quad \odot \in \{<, \leq, =, \geq, >\}^m. \tag{14}$$

Substituting one of the unknowns x_j by a certain value x amounts to replacing the j^{th} column of $\mathbf{A}$ with $\mathbf{0}$ and $\mathbf{b}$ with $\mathbf{b} - \mathbf{a}'_j x$. After this, the reduced record of unknowns, with x_j replaced by x has to obey the adapted system (14). For reference purposes, Algorithm 3 spells out the substitution routine. Figure 4 shows how substValue can be called from the R environment. The function is set up so multiple variables can be substituted in a single call as well.

3.2 Gaussian elimination

The well-known Gaussian elimination routine has been implemented as a utility function, enabling users to reduce the equality part of their edit matrices to reduced row echelon form. The echelon function has been overloaded to take either an R matrix or an editmatrix as argument. In the latter case, the equalities are transformed to reduced row echelon form, while inequalities are left untreated. Gaussian elimination is explained in many textbooks (see for example Lipschutz and Lipson (2000)). Algorithm 4 is written in a notation which is close to our R implementation in the sense that it involves just one explicit loop. Figure 5 demonstrates a call to the R function.

3.3 Fourier-Motzkin elimination

Fourier-Motzkin elimination [Fourier (1826); Motzkin (1936), but see Williams (1986) for an elaborate or Schrijver (1998) for a concise description] is an extension of Gaussian elimination to solving systems of linear inequalities. While Gaussian elimination is based on the reversible operations of row permutation and linear combination, Fourier-Motzkin elimination is based on the irreversible action of taking positive combinations of rows.

A full Fourier-Motzkin operation on a system of inequalities involves eliminating variables (where possible) one by one from the augmented matrix $[\mathbf{A}|\mathbf{b}]$. Eliminating a single variable is an important step in the error localization algorithm elaborated in Section 4.

Consider a system of inequalities $\mathbf{A}\mathbf{x} \leq \mathbf{b}$. The j^{th} variable is eliminated by generating a positive combination of every row of $[\mathbf{A}|\mathbf{b}]$ where $A_{ij} > 0$ with every row of $[\mathbf{A}|\mathbf{b}]$ where $A_{ij} < 0$ such that for the resulting row the j^{th} coefficient equals zero. Rows of $[\mathbf{A}|\mathbf{b}]$ for which $A_{ij} = 0$ are copied to the resulting system. If the system does not contain rows for which $A_{ij} > 0$ and rows for which $A_{ij} < 0$, the result is the removal of all rows with nonzero A_{ij}

Mixed systems with linear restrictions of the form $\mathbf{a} \cdot \mathbf{x} \odot b$ with $\odot \in \{<, \leq, =\}$ can in principle be transformed to a form where every $\odot \in \{\leq\}$. Restricions with $\odot \in \{<\}$ can be transformed to $\leq$ by subtracting a suitible small number from the right hand side of the inequation. However, it is more efficient to take the comparison operators into account when combining rows. In that case, new rules are derived by first solving the j^{th} variable from each equality and substituting them in each inequality. Next, inequalities are treated as stated before. When inequalities are combined where one comparison operator is $<$ and the other is $\leq$, it is not difficult to show that $<$ becomes the operator for the resulting inequality.

It is a basic result of the theory of linear inequalities that the system resulting from a single variable elimination is equivalent to the original system. In Fourier-Motzkin elimination, h elimination steps can generate up to $(\frac{1}{2}m)^{2h}$ new rows (m being the original number of rows), of which many are redundant. Since the number of redundant rows increases fast during elimination, removing (most of) them is highly desirable. In our implementation, we use the property that if h variables have been eliminated, any row derived from more than $h + 1$ rows of the original system is redundant. This result was originally stated by Černikov (1963) and rediscovered by Kohler (1967). A proof can also be found in Williams (1986). For the implementation in R, an editmatrix is augmented with an integer h, recording the number of eliminations and a logical array $\mathbf{H}$, which records for each edit from which original edit it was derived. Obviously, $\mathbf{H}$ is TRUE only on the diagonal when $h = 0$. It is worth mentioning that by using R's vectorized indices and recycling properties, it is possible to avoid any explicit looping

Table 2: Edit manipulation functions. Only mandatory functions are shown. Refer to the built-in documentation for optional arguments

function	description
substValue(E,var,value)	(multiple) value substitution
echelon(E)	bring equalities in echelon form
eliminate(E,var)	Fourier-Motzkin elimination
getH(E)	derivation history of E
geth(E)	nr. of eliminated variables

in the elimination process. Algorithm 5 gives an overview of the algorithm where explicit loops are included for readability. Figure 6 shows how one or more variables can be eliminated from an editmatrix with the eliminate function. Note that when multiple variables are eliminated, the editmatrix must be overwritten at every iteration to ensure that the history **H** is updated accordingly.

```
> data(edits)
> edits

  name          edit               description
1   b1   t == ct + p             total balance
2   b2 ct == ch + cp              cost balance
3   s1    p <= 0.6*t             profit sanity
4   s2   cp <= 0.3*t     personnel cost sanity
5   s3   ch <= 0.3*t       housing cost sanity
6   p1          t >0       turnover positivity
7   p2        ch > 0   housing cost positivity
8   p3        cp > 0 personnel cost positivity
9   p4        ct > 0     total cost positivity

> editmatrix(edits)

Edit matrix:
   ct  p    t ch cp Ops CONSTANT
b1 -1 -1  1.0  0  0  ==        0
b2  1  0  0.0 -1 -1  ==        0
s1  0  1 -0.6  0  0  <=        0
s2  0  0 -0.3  0  1  <=        0
s3  0  0 -0.3  1  0  <=        0
p1  0  0 -1.0  0  0   <        0
p2  0  0  0.0 -1  0   <        0
p3  0  0  0.0  0 -1   <        0
p4 -1  0  0.0  0  0   <        0

Edit rules:
b1 : t == ct + p [ total balance ]
b2 : ct == ch + cp [ cost balance ]
s1 : p <= 0.6*t [ profit sanity ]
s2 : cp <= 0.3*t [ personnel cost sanity ]
s3 : ch <= 0.3*t [ housing cost sanity ]
p1 : 0 < t [  turnover positivity ]
p2 : 0 < ch [ housing cost positivity ]
p3 : 0 < cp [ personnel cost positivity ]
p4 : 0 < ct [ total cost positivity ]
```

Figure 2: Declaring an editmatrix with a data.frame. The input data.frame is required to have three columns named name, edit (textual representation of the edit rule) and description (a comment stating the intent of the rule). All must be of type character.

```
> # define two records in a data.frame
> dat <- data.frame(
+   t = c(1000, 1200),
+  ct = c(400,  200),
+  ch = c(100,  350),
+  cp = c(200,  575),
+  p =  c(500,  652 ))
> # check for violated edits
> violatedEdits(E,dat)

      edit
record num1 num2  num3  num4  num5  num6  num7  num8  num9
     1 TRUE TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
     2 TRUE TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
```

Figure 3: Checking which edits are violated for every record in a data.frame. The editmatrix is the same as used in Fig. 2. The first record violates e1, e2 and e4, the second record violates e1, e2, and e5.

Algorithm 1 isObviouslyInfeasible(E)

Input: a normalized editmatrix E

for $\mathbf{a}_i \cdot \mathbf{x} \odot_i b_i \in E$ **do**
  if $\mathbf{a}_i = \mathbf{0} \wedge \neg 0 \odot_i b_i$ **then**
    return TRUE
return FALSE

Output: ▷ logical indicating if E is obviously infeasible.

Algorithm 2 isObviouslyRedundant(E, duplicates, ε)

Input: a normalized editmatrix E, with m edits, a boolean "duplicates", and a tolerance ε.

$\mathbf{v} \leftarrow (\text{FALSE})^m$
for $\mathbf{a}_i \cdot \mathbf{x} \odot_i b_i \in E$ **do**
  if $\mathbf{a}_i = 0 \wedge 0 \odot_i b_i$ **then**
    $v_i \leftarrow$TRUE
if duplicates **then**
  for $\{(\mathbf{a}_i \cdot \mathbf{x} \odot_i b_i, \mathbf{a}_j \cdot \mathbf{x} \odot_j b_j) \in E \times E : j > i\}$ **do**
    if $|(\mathbf{a}_i, b_i) - (\mathbf{a}_j, b_j)| \leq \varepsilon$ element wise $\wedge \odot_i = \odot_j$ **then**
      $v_j \leftarrow$TRUE

Output: $\mathbf{v}$ ▷ logical vector indicating which rows of E are obviously redundant.

Algorithm 3 substValue(E, j, x)

Input: $E = \langle [\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_j, \ldots, \mathbf{a}_n] | \mathbf{b}], \odot \rangle$, $x \in \mathbb{R}$, $j \in \{1, 2, \ldots n\}$
▷ Note that here, the subscripts of $\mathbf{a}$ denote the column index of $\mathbf{A}$

Output: $\langle [\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_{j-1}, \mathbf{0}, \mathbf{a}_{j+1}, \ldots \mathbf{a}_n] \, | \mathbf{b} - \mathbf{a}_j x], \odot \rangle$

```
> substValue(E,"t",10)

Edit matrix:
     ct  p t ch cp Ops CONSTANT
num1 -1 -1 0  0  0  ==      -10
num2  1  0 0 -1 -1  ==        0
num3  0  1 0  0  0  <=        6
num4  1  0 0  0  0  <=        3
num5  0  0 0  0  1  <=        3
num7  0  0 0 -1  0   <        0
num8  0  0 0  0 -1   <        0
num9 -1  0 0  0  0   <        0

Edit rules:
num1 : 10 == ct + p
num2 : ct == ch + cp
num3 : p <= 6
num4 : ct <= 3
num5 : cp <= 3
num7 : 0 < ch
num8 : 0 < cp
num9 : 0 < ct
```

Figure 4: Substituting the value 10 for the turnover variable using the substValue function. substValue can substitute multiple values as well.

Algorithm 4 ECHELON(E)

Input: An editmatrix $\langle [\mathbf{A}|\mathbf{b}], = \rangle$, $[\mathbf{A}|\mathbf{b}] \in \mathbb{R}^{m \times (n+1)}$, $m \leq n+1$.

$I \leftarrow \{1, 2, \ldots, m\}$

$J \leftarrow \{1, 2, \ldots, n+1\}$

for $j \in I$ **do** ▷ eliminate variables

  $i \leftarrow \arg\max_{i' \,:\, j \leq i' \leq m} |A_{i'j}|$

  if $|A_{ij}| > 0$ **then**

    if $i > j$ **then**

      Swap rows i and j of $[\mathbf{A}|\mathbf{b}]$.

    $[\mathbf{A}|\mathbf{b}]_{I \setminus j, J} \leftarrow [\mathbf{A}|\mathbf{b}]_{I \setminus j, J} - [\mathbf{A}|\mathbf{b}]_{I \setminus j, j} \otimes [\mathbf{A}|\mathbf{b}]_{j,J} A_{jj}^{-1}$

Divide each row $[\mathbf{A}|\mathbf{b}]_{i,J}$ by A_{ii} when $A_{ii} \neq 0$

Move rows of $[\mathbf{A}|\mathbf{b}]$ with all zeros to bottom.

Output: E, transformed to reduced row echelon form.

```
> (E2 <- editmatrix(c("2*x1 + x2 -x3 == 8",
+                     "2*x3 + 11 == 3*x1 + x2",
+                     "x2 + 2*x3 + 3 == 2*x1")
+                   ))

Edit matrix:
     x1 x2 x3 Ops CONSTANT
num1  2  1 -1  ==        8
num2 -3 -1  2  ==      -11
num3 -2  1  2  ==       -3

Edit rules:
num1 : 2*x1 + x2 == x3 + 8
num2 : 2*x3 + 11 == 3*x1 + x2
num3 : x2 + 2*x3 + 3 == 2*x1

> echelon(E2)

Edit matrix:
     x1 x2 x3 Ops CONSTANT
num1  1  0  0  ==        2
num2  0  1  0  ==        3
num3  0  0  1  ==       -1

Edit rules:
num1 : x1 == 2
num2 : x2 == 3
num3 : x3 == -1
```

Figure 5: Transforming linear equalities of an editmatrix to reduced row echelon form. If the editmatrix argument contains inequalities, these are copied to the resulting system.

Algorithm 5 ELIMINATE(E, j). In the actual implementation all explicit loops are avoided by making use of R's recycling properties and vectorized indices.

Input: A normalized editmatrix $E = \langle [\mathbf{A}|\mathbf{b}], \odot, \mathbf{H}, h \rangle$, and a variable index j.

if $\mathbf{H} = \varnothing$ **then**
  $\mathbf{H} \leftarrow \text{diag}(\text{TRUE})^m$
  $h \leftarrow 0$
$J \leftarrow \{1, 2, \ldots, n+1\}$
$I_0 \leftarrow \{i : A_{ij} = 0\}$
$I_= \leftarrow \{i : \odot_i \in \{=\}\} \backslash I_0$
$I_+ \leftarrow \{i : A_{ij} > 0\} \backslash I_=$
$I_- \leftarrow \{i : A_{ij} < 0\} \backslash I_=$
for $i \in \{1, 2, \ldots, m\} \backslash I_0$ **do** ▷ All rows get j^{th} coefficient in $\{-1, 0, 1\}$
  if $\odot_i \in \{<, \leq\}$ **then**
    $[\mathbf{A}|\mathbf{b}]_{i,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} |A_{ii}|^{-1}$
  else
    $[\mathbf{A}|\mathbf{b}]_{i,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} A_{ii}^{-1}$

▷ Substitute equalities and inequalities with positive j^{th} coefficient in inequalities with negative j^{th} coefficient:

for $(i, j) \in (I_= \cup I_+) \times I_-$ **do**
  $k \leftarrow k + 1$
  $[\tilde{\mathbf{A}}|\tilde{\mathbf{b}}]_{k,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} + [\mathbf{A}|\mathbf{b}]_{j,J}$
  $\tilde{\mathbf{H}}_{k,J} \leftarrow \mathbf{H}_{i,J} \vee \mathbf{H}_{j,J}$
  if $\odot_i \in \{<\}$ **then** $\tilde{\odot}_k \leftarrow \odot_i$ **else** $\tilde{\odot}_k \leftarrow \odot_j$

▷ Substitute equalities in inequalities with positive j^{th} coefficient

for $(i, j) \in I_+ \times I_=$ **do**
  $k \leftarrow k + 1$
  $[\tilde{\mathbf{A}}|\tilde{\mathbf{b}}]_{k,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} - [\mathbf{A}|\mathbf{b}]_{j,J}$
  $\tilde{\mathbf{H}}_{k,J} \leftarrow \mathbf{H}_{i,J} \vee \mathbf{H}_{j,J}$
  $\tilde{\odot}_k \leftarrow \odot_i$

for $\{(i, j) \in I_=^{\times 2} : j > i\}$ **do** ▷ Substitute equalities in equalities
  $k \leftarrow k + 1$
  $[\tilde{\mathbf{A}}|\tilde{\mathbf{b}}]_{k,J} \leftarrow [\mathbf{A}|\mathbf{b}]_{i,J} - [\mathbf{A}|\mathbf{b}]_{j,J}$
  $\tilde{\mathbf{H}}_{k,J} \leftarrow \mathbf{H}_{i,J} \vee \mathbf{H}_{j,J}$
  $\tilde{\odot}_k \leftarrow \odot_i$

$$\tilde{E} \leftarrow \left\langle \left[\tilde{\mathbf{A}}|\tilde{\mathbf{b}}]', [\mathbf{A}|\mathbf{b}]'_{I_0,J}\right]', (\tilde{\odot}, \odot_{I_0}), \tilde{\mathbf{H}}, h+1 \right\rangle$$

Remove edit rules of $\tilde{E}$ which have more than $h+1$ elements of $\mathbf{H}_{i,J}$ TRUE
Remove edit rules of $\tilde{E}$ for which ISOBVIOUSLYREDUNDANT$(\tilde{E})$ is TRUE

Output: editmatrix $\tilde{E}$ with variable j eliminated and updated history

```
> eliminate(E,"t")

Edit matrix:
          ct          p t ch        cp Ops CONSTANT
e1 -1.000000  0.6666667 0  0  0.000000  <=        0
e2  2.333333 -1.0000000 0  0  0.000000  <=        0
e3 -1.000000 -1.0000000 0  0  3.333333  <=        0
e4 -1.000000 -1.0000000 0  0  0.000000   <        0
e5  1.000000  0.0000000 0 -1 -1.000000  ==        0
e6  0.000000  0.0000000 0 -1  0.000000   <        0
e7  0.000000  0.0000000 0  0 -1.000000   <        0
e8 -1.000000  0.0000000 0  0  0.000000   <        0

Edit rules:
e1 : 0.666666666666667*p <= ct
e2 : 2.33333333333333*ct <= p
e3 : 3.33333333333333*cp <= ct + p
e4 : 0 < ct + p
e5 : ct == ch + cp
e6 : 0 < ch
e7 : 0 < cp
e8 : 0 < ct

> F <- E
> for ( var in c("t","cp","p") ) F <- eliminate(F,var)
> F

Edit matrix:
           ct p t        ch cp Ops CONSTANT
e1 -2.5000000 0 0  0.000000  0   <        0
e2  0.8333333 0 0 -3.333333  0  <=        0
e3  0.8333333 0 0  0.000000  0  <=        0
e4 -1.0000000 0 0  1.000000  0   <        0
e5  0.0000000 0 0 -1.000000  0   <        0
e6 -1.0000000 0 0  0.000000  0   <        0

Edit rules:
e1 : 0 < 2.5*ct
e2 : 0.833333333333334*ct <= 3.33333333333333*ch
e3 : 0.833333333333334*ct <= 0
e4 : ch < ct
e5 : 0 < ch
e6 : 0 < ct
```

Figure 6: Above: eliminating t from the editmatrix with the eliminate function. Below: to eliminate multiple variables, the original editmatrix must be overwritten at each iteration to ensure that the derivation history is updated at every step.

4 Error localization for numerical data

While checking whether a numerical record violates any imposed restrictions (within a certain limit) is easy, finding out which variable(s) of the record cause the violation(s) can be far from trivial. If possible, the cause of the violation should be sought out, since it leads immediately to repair suggestions. The deducorrect package (Van der Loo et al., 2011) mentioned above offers functionality to detect and repair typing errors, rounding errors and sign errors. Although not directly available in R, methods for detecting and repairing unit measure errors or other systematic errors have been described in literature and may readily be implemented in R (see De Waal et al. (2011) Chapter 2 for an overview).

After systematic errors with detectable causes in a data set have been resolved, one may assume that remaining errors are distributed randomly (but not necessarily uniformly) over one or more of the variables. In that case, error localization based on the (generalized) principle of Fellegi and Holt can be applied.

4.1 The generalized Fellegi-Holt paradigm

In line with the good practice of altering source data as little as possible, the paradigm of Fellegi and Holt (1976) advises to edit an as small number of variables as possible, under the condition that after editing, every edit rule can be obeyed. A generalization of this principle says that a weighted number of variables should be minimized. More formally the principle yields the following problem. Given a record $\mathbf{x}$, violating a number of edits in an edit matrix E (see Eq. (3)) with m rules and n variables, find G such that

$$
\begin{aligned}
G = & \underset{g \subset \{1,2,\ldots,n\}}{\operatorname{argmin}} \sum_{j \in g} w_j \\
& \text{such that a solution } \tilde{\mathbf{x}} \in \mathbb{R}^{|G|} \text{ exists for} \\
& \sum_{j \in G} A_{ij} \tilde{x}_j \odot_i b_i - \sum_{j \notin G} A_{ij} x_j, \quad i \in \{1, 2, \ldots, m\}.
\end{aligned} \tag{15}
$$

In other words, for every variable in $\mathbf{x}$, we have to decide whether to use or adapt its value. Unadapted variables can be replaced with their observed value x_j while the values of the remaining variables have to be changed into $\tilde{x}_j$, such that these values form no contradiction. The solution to (15) need not be unique, but there is always at least one solution unless the edit rules in E are contradictory.

The minimization (15) amounts to a binary search problem, of which the search space increases as 2^n (n the number of variables). De Waal (2003) and De Waal et al. (2011) describe a branch-and-bound binary search algorithm which generates all minimal weight solutions. It works by generating the following binary tree: the root node contains E and $\mathbf{x}$ and weight $w = 0$.

Both left and right child nodes of the root node receive a copy of the objects in their parent. In the left child node, x_1 is assumed correct and its value is substituted in E. In the right child node, x_1 is assumed to contain an error and it is eliminated from E by Fourier-Motzkin elimination. The weight w in the right node is increased by w_1. Each child node gets a left and right child node where x_2 is substituted or eliminated, and so on until every variable has been treated. Every path from root to leaf represents one element of the search space. A branch is pruned when E contains obvious inconsistencies, so no combinations not satisfying the condition in (15) are generated. If a solution with certain weight w is found, branches developed later, receiving a higher weight are pruned as well.

To clarify the above, in the next subsection we give two worked examples. Subsection (4.5) describes a flexible binary search algorithm, which we implemented to support general binary search problems. Subsection 4.4 describes its application to the branch-and-bound algorithm mentioned above.

4.2 Two examples

To illustrate the binary search algorithm outlined above we will consider a simple two-dimensional example. The reader is encouraged to follow the reasoning below by checking the calculations using the R-functions mentioned in the previous sections.

Consider a 2-variable record (x, y) subject to the set of constraints E:

$$E = \begin{cases} e_1: & y > x - 1 \\ e_2: & y > -x + 3 \\ e_3: & y < x + 1 \\ e_4: & y < -x + 5. \end{cases} \tag{16}$$

Each separate inequality yields a half-plane of which the border is determined by the line obtained by replacing $<$ or $>$ by $=$. The intersection of the four half-planes is the region of allowed records. In this example, the region is a diamond, depicted as the gray area in Figure 1. The borders are labeled with the edit rules in Eq. (16). Consider the record $(x = 2, y = -1)$, depicted as the bottom black dot in Figure 1. It is easy to confirm either graphically or by substitution that $(2, -1)$ violates edits e_1 and e_2, and that the record can be made consistent by altering only y and leaving x constant (indicated by the black arrow). It is also clear from the graph that the allowed values for y are between 1 and 3 (indicated by the thin black vertical line in the diamond). The case $(x = 0, y = 0)$ also violates e_1 and e_2 and can only be repaired by altering both x and y, while the record $(x = -1, y = 2)$ can be repaired by changing x only.

In the following we show that the binary search algorithm described in the previous subsection indeed solves the error localization problem for

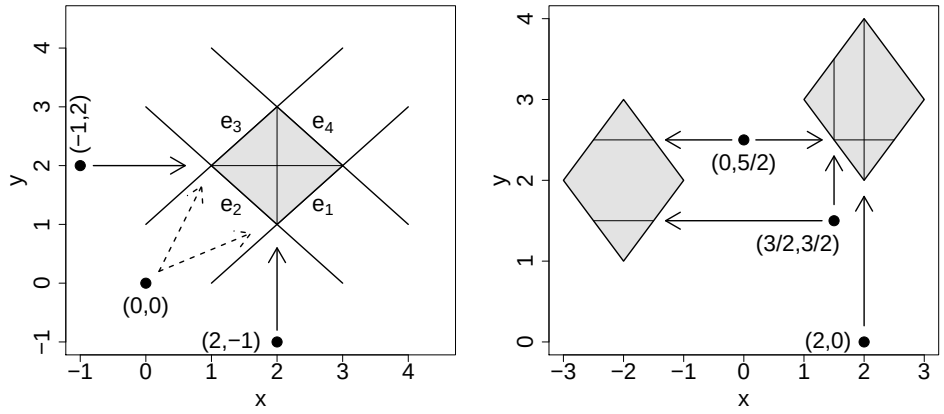

Figure 1: Graphic representation of edit rules and the allowed area. Left panel: a convex case, as defined by Eq. (16). Right panel: the non-convex unconnected case, as defined by Eq. (24). Gray areas indicate the valid record domain, black dots indicate erroneous records and black arrows indicate the solution of the error localization problem, while the thin black lines show the range of solutions. The dotted arrows in the left panel indicate the range of directions in which the record (0,0) can move to reach the valid area.

$(x = 2, y = -1)$. To find the unweighted, least number of variables to adapt, so that E can be fulfilled, consider the triple

$$T_0 = \langle E, (2, -1), w = 0) \rangle \,, \tag{17}$$

This is the root node of the binary search tree described in the previous subsection, with w the initial solution weight. The left child is generated by assuming that the first value in the record is correct. We therefore replace the variable x in E by its value in the record, which yields after removing redundancies,

$$T_{0l} = \left\langle \begin{array}{l} y > 1 \\ y < 3 \end{array} , (2, -1), 0 \right\rangle . \tag{18}$$

In this notation, each time a left (right) node is added, the subscript of T is augmented with an l (r). Substituting one of the values further restricts the possible values for variables that have not been treated yet. In fact, after the error localization problem has been solved, substituting all unaltered values into E yields a set of equations which determine the range of the variables which have to be altered or imputed.

Since no variables were eliminated, the weight in T_{0l} is 0, and the record has not changed. In the right child of the root, x is assumed to be wrong,

and therefore eliminated using Fourier-Motzkin elimination:

$$T_{0r} = \left\langle \begin{array}{l} y > 1 \\ y < 3 \end{array} , (x, -1), 1 \right\rangle . \tag{19}$$

The system of equations left after elimination of x illustrates the geometrical interpretation of Fourier-Motzkin elimination. The range of y corresponds to the projection of the diamond in the left pane of Figure 1 onto the y-axis. (The fact that T_{0l} yields the same system is mere coincidence and depends on the fact that the x-coordinate in the record at hand equals 2). Calculating the left child of T_{0l} means substituting y by -1 in the edits of T_{0l}. This yields

$$T_{0ll} = \left\langle \begin{array}{l} -1 > 1 \\ -1 < 3 \end{array} , (2, -1), 0 \right\rangle , \tag{20}$$

where the contradiction $-1 > 1$ indicates that T_{0ll} is not a solution (which is obvious since none of the values in the records are assumed incorrect). The right child of T_{0l} is obtained by eliminating y:

$$T_{0lr} = \langle \varnothing, (2, y), 1 \rangle , \tag{21}$$

where the tautology $0 < 2$ was removed. This end node does represent a solution, since no conflicting rules have been generated. To see if any other solutions exist, continue to calculate the left child node of T_{0r}

$$T_{0rl} = \left\langle \begin{array}{l} -1 > 1 \\ -1 < 3 \end{array} , (x, -1), 1 \right\rangle , \tag{22}$$

which is no solution since its edits hold a contradiction. The final, right child node of T_{0r} reads

$$T_{0rr} = \langle \varnothing, (x, y), 2 \rangle , \tag{23}$$

which also is a solution, but since both x and y have to be adapted, it has a higher weight than the solution T_{0lr} found earlier.

The edit sets described so far involved a single set of (in)equalities, yielding a convex record domain in $\mathbb{R}^n$. However, in practical cases the sets of allowed values for a record need not be convex, or even connected. As an example consider the space of allowed records, indicated by the gray areas in the right panel of Figure 1. Such a range can be defined by a conditional edit of the form

$$\textbf{if } e_0 : x < 0 \textbf{ then } \left\{ \begin{array}{ll} e_1 : & y > x + 3 \\ e_2 : & y > -x + 1 \\ e_3 : & y < x + 5 \\ e_4 : & y < -x + 1 \end{array} \right. \textbf{ else } \left\{ \begin{array}{ll} e'_1 : & y > x \\ e'_2 : & y > -x + 4 \\ e'_3 : & y < x + 2 \\ e'_4 : & y < -x + 6. \end{array} \right. \tag{24}$$

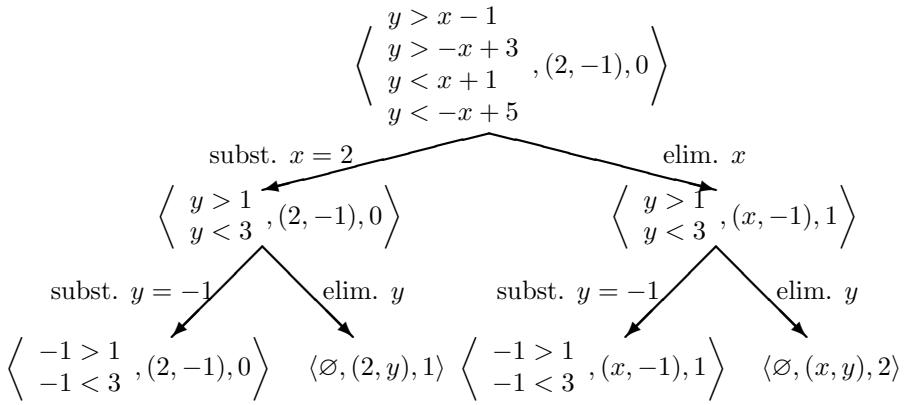

Figure 2: Graphical representation of the binary tree used to solve the error localization problem for the record $(x = -2, y = -1)$, subject to the edits of Eqn.(16). Each node contains an edit set, a (partially completed) record and the solution weight.

This error localization problem can be handled by solving the partial localization problems for $\{e_0, e_1, \ldots, e_4\}$ and $\{\overline{e}_0, e_1', \ldots, e_4'\}$ separately, where $\overline{e}_0$ stands for the complement $\overline{e}_0 : x \geq 0$. The partial solution with the lowest weight solves the complete optimization problem. As an illustration consider the record $(x = 2, y = 0)$ in the right panel of Figure 1. The error localization problem corresponding to $x < 0$ yields a solution where both x and y have to be altered, while the localization problem corresponding to $x \geq 0$ implies that only y needs to be altered.

To generalize this example, note that a conditional edit set of the form

$$\textbf{if } E_0 \textbf{ then } E_1 \textbf{ else } E_2, \tag{25}$$

can be written as

$$(E_0 \wedge E_1) \vee (\overline{E}_0 \wedge E_2), \tag{26}$$

which may be treated by finding the minimum weight solution between the solutions generated by $E_0 \wedge E_1$ and $\overline{E}_0 \wedge E_2$. Taking the complement can cause the number of partial localization problems to grow quickly. As an illustration, consider the following case where taking the complement yields three cases to be treated by the error localization routine.

$$\begin{aligned} &\textbf{if } (x = 0) \textbf{ then } E_1 \textbf{ else } E_2 \\ &\quad \Leftrightarrow \quad ((x = 0) \wedge E_1) \vee ((x \neq 0) \wedge E_2) \\ &\quad \Leftrightarrow \quad ((x = 0) \wedge E_1) \vee ((x < 0) \wedge E_2) \vee ((x > 0) \wedge E_2). \end{aligned} \tag{27}$$

Table 3: 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

The number of partial error localization problems to be treated grows as $2n_{\text{eq}} + n_{\text{ineq}}$, where n_{eq} is the number of equalities and n_{ineq} the number of inequalities in E_0. This is easily derived from Eq. (26) since by De Morgan's rule, if $E_0 = e_1 \wedge e_2 \wedge \ldots \wedge e_k$, then

$$\overline{E}_0 = \overline{e_1 \wedge e_2 \wedge \ldots \wedge e_k} = \overline{e}_1 \vee \overline{e}_2 \vee \ldots \vee \overline{e}_k. \tag{28}$$

Here, each negated inequality translates to a single inequality, while each negated equality yields two inequalities (as in Eq. (27)).

We will have more to say on conditional edits in the accompanying paper (Van der Loo and De Jonge, 2011) where the error localization problem for categorical and mixed data are treated.

4.3 Error localization with localizeErrors

The function localizeErrors accepts an editmatrix and a data.frame, and returns an object of class errorLocation. An errorLocation object contains the locations of errors for each record in the data.frame as well as logging information, solution weights and degeneracy. Table 3 gives an overview of the slots.

Apart from the mandatory arguments (an editarray and a data.frame), there are optional arguments which will be passed to the underlying errorLocalizer function which is described in detail in the next section. These arguments are given in table 4.

Figure 7 shows an example of localizing errors with errorLocalizer. The function applies a branch-and-bound algorithm to find the least weighted number of variables which may be adapted in such a way that all edits are satisfied. If there are multiple degenerate (equally weighted) solutions, one of those solutions is drawn at random.

Table 4: Arguments of localizeErrors. Optional arguments are given in square brackets. The optional arguments are also arguments of the underlying errorLocalizer function.

Argument	description
E	An editmatrix or an editarray.
dat	The data, in the form of a data.frame.
[weight]	Nonnegative weights for each variable in dat.
[maxadapt]	Maximum number of variables to adapt.
[maxweight]	Maximum weight of solution, if weights are not given, this is equal to the maximum number of variables to adapt.
[maxduration]	Maximum time (in seconds), spent searching for a solution for a single record.

4.4 Error localization with errorLocalizer

The error localization problem detailed in the previous subsections can be automated with errorLocalizer. This function expects an editmatrix, a named numerical record and optionally a vector of reliability weights with the same length as the record. There are extra options to control the maximum number of variables to adapt (maxadapt), the maximum weight (maxweight) and the maximum search time (maxduration) in seconds. The return value of errorLocalizer is not the solution to the error localization problem but an object of class backtracker. With a backtracker object the branch-and-bound tree can be searched to find solutions one by one. The internal machinery of backtracker objects is detailed in the next subsection, in this section it is shown how to use such objects to solve error localization problems.

Consider again the edits of Eqn. (16), and the record $(x = 2, y = -1)$. Figure 8 shows how the error localization problem can be solved with the backtracker object returned by errorLocalizer. By calling the built-in searchNext function, the backtracker object traverses the binary search tree depth-first, until the first solution is found or maxduration is exceeded. If a solution is found, the contents of the current node is returned to the user as a list. It contains the current solution weight w and a named logical vector called adapt, indicating which variables have to be adapted. If maxduration is exceeded or no solution is found, NULL is returned. The backtracker object property maxdurationExceeded indicates if the time limit has been exceeded or not.

As expected, in the example y is pointed out as the variable to change. At this point, the backtracker object contains all the information needed to continue the search for new solutions, starting from the node where it just ended. It also stores some information on the elapsed time needed for the previous search in the form of a standard proc_time object.

```
> E <- editmatrix(c(
+     "x + y == z",
+     "x > 0",
+     "y > 0",
+     "z > 0"))
> dat <- data.frame(
+     x = c(1,-1,1),
+     y = c(-1,1,1),
+     z = c(2,0,2))
> # localize all errors in the data
> localizeErrors(E,dat)

Object of class 'errorLocation' generated at Wed Feb  8 16:51:03 2012
call : localizeErrors(E, dat)
method : localizer
slots: $adapt $status $call $method $user $timestamp

Values to adapt:
      adapt
record     x     y     z
     1 FALSE  TRUE FALSE
     2  TRUE FALSE  TRUE
     3 FALSE FALSE FALSE

 Status:
  weight degeneracy  user system elapsed maxDurationExceeded
1      1          1 0.012      0   0.008               FALSE
2      2          1 0.004      0   0.006               FALSE
3      0          1 0.004      0   0.005               FALSE
```

Figure 7: Localizing errors for every record in a data.frame with localizeErrors

Another call to searchNext will search for the next solution in the tree, with lower weight. However, since in this example there is only one solution, searchNext returns NULL.

The method searchNext is not the only method of the backtracker object returned by errorLocalizer. The available methods are

- `$searchNext` Searches for the next solution with a lower weight than the previously found solution.
- `$searchAll` Returns all solutions encountered in the branch-and-bound search before maxduration is exceeded.
- `$searchBest` Returns the lowest-weight solution of all solutions found before maxduration is exceeded. If multiple solutions have the same, minimum weight, it returns one of those solutions at random.

```
> E1 = editmatrix(c(
+     "y >  x - 1",
+     "y > -x + 3",
+     "y <  x + 1",
+     "y < -x + 5"))
> bt <- errorLocalizer(E1, c(x=2,y=-1))
> bt$searchNext()

$w
[1] 1

$adapt
    x     y
FALSE  TRUE

> bt$duration

   user  system elapsed
  0.000   0.000   0.002

> bt$maxdurationExceeded

[1] FALSE

> bt$searchNext()

NULL
```

Figure 8: Localizing errors with the backtracker object generated by errorLocalizer. After a search is performed, the backtracker object holds information on the duration of the search, and if the time-limit for a search was exceeded.

All these methods accept the following optional arguments:

- `maxduration` The number of seconds after which to stop the search. The default value is the value passed to errorLocalizer, which created the object.
- `VERBOSE` Print the path in the search tree and contents of each node during search.

Any backtracker object is equipped with the searchNext and searchAll methods. The searchBest method is specific for the backtracker object returned by errorLocalizer.

The backtracker method offers a flexible interface for error localization. To understand what happens when there are multiple solutions, consider the case of a simple balance account for profit (p), loss (l) and turnover (t):

```
> E <- editmatrix(c("p + c == t"))
> r <- c(p=755, c=125, t=200)
> bt <- errorLocalizer(E, r)
```

The record obviously violates the edit in E. Since there is only a single edit rule, there are three solutions, all of which can be found by calling bt$searchNext

```
> bt$searchNext()$adapt

    p     c     t
FALSE FALSE  TRUE

> bt$searchNext()$adapt

    p     c     t
FALSE  TRUE FALSE

> bt$searchNext()$adapt

    p     c     t
 TRUE FALSE FALSE
```

Each solution has weight 1. Suppose that the turnover value is trusted more, for example because it comes from a more reliable source. We may increase its reliability weight by providing a weight vector:

```
> bt <- errorLocalizer(E, r, weight=c(1,1,2))
> bt$searchNext()$adapt

    p     c     t
FALSE  TRUE FALSE

> bt$searchNext()$adapt

    p     c     t
 TRUE FALSE FALSE

> bt$searchNext()$adapt

NULL
```

The solution where turnover must be adapted is not found anymore. The reason is that errorLocalizer makes sure that during the search for solutions, variables with the highest reliability weight are the last ones to be assumed incorrect. Since it has found solutions for the less reliable variables (p and c), it won't search for solutions with higher weight.

If we add more restrictions, the number of solutions to the error localization problem decreases. Here, we demand that the cost to turnover ratio does not exceed 0.6.

```
> E <- editmatrix(c(
+         "p + c == t",
+         "c - 0.6*t >= 0"))
> bt <- errorLocalizer(E, r)
> bt$searchNext()$adapt

    p     c     t
FALSE  TRUE  TRUE

> bt$searchNext()$adapt

    p     c     t
 TRUE FALSE FALSE

> bt$searchNext()$adapt

NULL
```

Here, first a solution of weight 2 is found, which may later be rejected in favor of the solution which demands only that the profit variable should be changed.

With errorLocalizer records with missing data can be handled as well. Variables with missing values are treated as variables that need to be adapted: they are eliminated from the edit matrix prior to further error localization. In the next example we add some extra variables and demand positivity of all variables.

```
> # An example with missing data.
> E <- editmatrix(c(
+     "p + c1 + c2 == t",
+     "c1 - 0.3*t >= 0",
+     "p > 0",
+     "c1 > 0",
+     "c2 > 0",
+     "t > 0"))
> x <- c(p=755, c1=50, c2=NA,t=200)
> bt <- errorLocalizer(E,x)
> bt$searchNext()$adapt

    p    c1    c2     t
FALSE  TRUE  TRUE  TRUE

> bt$searchNext()$adapt

    p    c1    c2     t
 TRUE FALSE  TRUE  TRUE

> (s <- bt$searchNext()$adapt)
```

```
    p    c1    c2     t
 TRUE  TRUE  TRUE FALSE
```

There are three equivalent solutions, all of which include the field with the missing value (c2). To obtain the restrictions for the variables which have altered, simply substitute all values which are retained in the solution, for example:

```
> substValue(E,names(x)[!s],x[!s])

Edit matrix:
     c1 c2  p t Ops CONSTANT
num1  1  1  1 0  ==      200
num2 -1  0  0 0  <=      -60
num3  0  0 -1 0   <        0
num4 -1  0  0 0   <        0
num5  0 -1  0 0   <        0

Edit rules:
num1 : c1 + c2 + p == 200
num2 : 60 <= c1
num3 : 0 < p
num4 : 0 < c1
num5 : 0 < c2
```

This system of equations must be obeyed if p, c1 and c2 are going to be adapted or imputed.

4.5 General binary search with the backtracker object

As stated in subsection 4.1, the error localization problem can be interpreted as a (pruned) binary programming problem. To facilitate implementation of error localization for numerical, categorical and mixed data, as well as to help further research in error localization algorithms, we implemented general-purpose binary search functionality in the form of backtracking programming. A backtracking algorithm (Knuth, 1968) finds solutions to a computational problem by building incrementally candidate solutions. It starts with a partial solution and extends the partial solution in subsequent steps until it is a valid solution. When a partial solution is extended the full state of the current (sub) problem is stored in a "choice point". If a partial solution is not valid, the algorithm will "back track" to the last previously stored choice point and continue its search. In other words, it prunes invalid search subtrees and does not waste computation time on invalid solutions. Furthermore the algorithm allows users to specify how to extend a partial solution and when a partial solution is invalid.

Backtracking is a specific form of the more general "choice point" programming which stems from the field of nondeterministic programming. In

nondeterministic programming, the control flow of a program is not determined explicitly by the programmer with standard branching statements. In stead, choice points may be created which store the full state of a program so that control flow can at any time return to a stored state and choose a new path from there. Choice point programming is supported by various niche programming environments, such as Alma-0 (Partington, 1997) and ELAN (Vittek, 1996). See Moreau (1998) for a clear introduction or Mart-Oliet and Mesguer (2002) for a bibliographic overview. The choice point paradigm offers an excellent environment for programming backtracking algorithms, of which the branch-and-bound algorithm of subsection 4.1 is just a specific example.

The R language is ideally suited to develop choice point-like systems because of its first-class environments. An R environment can be thought of as a list of R objects, forming the scope for expression evaluation. Expressions are a series of R statements which may create, manipulate and remove R objects within an environment. Having first-class environments means that expressions can also be used to create, manipulate and delete environments like any other R object. Moreover, expressions can be evaluated in any environment created by the programmer.

In our implementation, we model the search tree as a binary search tree, in which each node is a binary choice (left or right) for extending the current partial solution. In the BACKTRACKER object the sequence of connected nodes is represented by a sequence of nested environments. Each environment stores the state of a binary "choice point" . Such a series of nested environments is equivalent to a stack, where a PUSH-operation corresponds to nesting a new environment and a POP-operation ensures that the next expression will be evaluated in the last-pushed environment. Since environments are nested, expressions evaluated in a child node have read access to information stored in the parent node. Pseudo-code for the BACKTRACKER object is given in Algorithm 6. Expressions are denoted with Greek letters ψ or ϕ, environments are denoted as $\mathcal{E}$ and :: is the scope resolution operator. The symbol $\mathcal{S}$ denotes a formal stack. We denote the result of evaluating an expression ϕ in an environment $\mathcal{E}$ as $\phi(\mathcal{E})$. One can think of ϕ as a subroutine which alters the internal state of $\mathcal{E}$. It is also possible for ϕ to generate a return value (by issuing a return statement) which is pushed to the enveloping environment, similar to the action of a standard function.

To construct a BACKTRACKER object, the user provides an expression ϕ_0 to initialize the root node, expressions ϕ_l and ϕ_r to be evaluated at left and right child nodes and an expression ψ to evaluate the contents of a node. The initialization expression usually consists of a number of variable declarations. Expressions ϕ_l and ϕ_r alter the state of left or right child node, any returned values are ignored. The expression ψ serves two purposes. First of all, it

judges a node $\mathcal{E}$ and must return one of the following values:

$$\psi(\mathcal{E}) = \begin{cases} \text{TRUE} & \text{if environment } \mathcal{E} \text{ contains a solution} \\ \text{FALSE} & \text{if environment } \mathcal{E} \text{ cannot lead to a solution} \\ \text{NULL} & \text{if environment } \mathcal{E} \text{ contains a partial solution.} \end{cases} \quad (29)$$

Secondly, ψ may be used to update weights or other administration and to prepare the variables in a node for output. The method searchNext generates nodes in the binary tree, depth-first and returns the (contents of) the first environment corresponding to a solution. If BT is the instance of a BACKTRACKER object, then each call to BT::searchNext will return a new, and better solution, until all solutions are found, in which case NULL is returned. A call to BT::searchAll (not shown in pseudo-code) will return all solutions. Since search spaces grow exponentially with tree depth, the backtracker object can be equipped with a time limit for tree search or a maximum tree depth. The latter is mainly useful for debugging purposes.

The backtracker function constructs a backtracker object and accepts the following arguments:

- isSolution : An R expression corresponding to ψ of Eqn. (29).
- choiceLeft : An R expression for execution in left child nodes (ϕ_l) .
- choiceRight : An R expression for execution in right child nodes (ϕ_r).
- maxduration : Optional: the default maximum time in seconds for a tree search with $searchNext() or $searchAll(). This time may be overwritten by passing a new maxduration when calling a search function.
- maxdepth : Optional: The maximum tree search depth.
- ... : Named arguments, to initialize the root node (ϕ_0).

As an example, Figure 9 shows a simple implementation of the branch-and-bound algorithm for error localization (the implementation in errorLocalizer is somewhat more advanced and faster than this example). The top environment (root node) receives an edit matrix E, a record r, a vector of variable names that have yet to be treated (totreat), a logical vector indicating whether a variable should be altered or not (adapt), a weight vector weight with reliability weights for each variable. Also, the weight wsol of the current solution is initialized to the maximum possible weight.

The expression isSolution first computes the weight of the current solution by adding all elements of weight for which adapt==TRUE. Next, it checks if the editmatrix is infeasible, or if the current weight exceeds the weight of the last found solution. Since wsol is initialized on the maximum weight, the latter can only happen when at least one solution has been found. If either condition is met, the branch must be pruned, so FALSE is returned.

Algorithm 6 backtracker object. ϕ_j and ψ are expressions, $\mathcal{E}$ and $\mathcal{E}'$ environments :: is the scope resolution operator and $\mathcal{S}$ a stack.

Struct BACKTRACKER $(\phi_0, \phi_l, \phi_r, \psi)$
  $\mathcal{S} \leftarrow$ NEWSTACK
  $\mathcal{E} \leftarrow$ NEWENVIRONMENT
  $\mathcal{E}$ :: treatedleft $\leftarrow$ FALSE
  $\mathcal{E}$ :: treatedright $\leftarrow$ FALSE
  $\phi_0(\mathcal{E})$ ▷ ϕ_0 Initialize root node
  PUSH$(\mathcal{E}, \mathcal{S})$
  Method SEARCHNEXT
    $\mathcal{E} \leftarrow$ POP$(\mathcal{S})$ ▷ POP returns NULL if stack is empty
    while $\psi(\mathcal{E}) \in \{$FALSE, NULL$\} \wedge \mathcal{E} \neq$ NULL **do**
      if $\neg\mathcal{E}$ :: treatedleft **then**
        $\mathcal{E}' \leftarrow \mathcal{E}$ ▷ Create child node
        $\phi_l(\mathcal{E}')$ ▷ Treat child node
        $\mathcal{E}$ :: treatedleft $\leftarrow$ TRUE ▷ Mark parent node
        PUSH$(\mathcal{E}, \mathcal{S})$
        PUSH$(\mathcal{E}', \mathcal{S})$
      else if $\neg\mathcal{E}$ :: treatedright **then**
        $\mathcal{E}' \leftarrow \mathcal{E}$
        $\phi_r(\mathcal{E}')$
        $\mathcal{E}$ :: treatedright $\leftarrow$ TRUE
        PUSH$(\mathcal{E}, \mathcal{S})$
        PUSH$(\mathcal{E}', \mathcal{S})$
      $\mathcal{E} \leftarrow$ POP$(\mathcal{S})$
    return $\mathcal{E}$
  EndMethod
EndStruct

Otherwise, it is checked whether any variables are left to treat. If so, the search continues. If not, the solution weight in the top environment is set (using the <<- operator) and TRUE is returned. Before returning, output is prepared by copying the variable adapt from the enveloping environment, and removing the empty vector totreat.

In choiceLeft, the first variable to be treated is chosen and its value replaced in the editmatrix. The value of E in the call to substValue is copied automatically from the enveloping environment which by construction holds the parent node of the node under treatment. For the same reason assigning the indexed value of adapt works. The value corresponding to the variable under treatment in adapt is set to FALSE since a variable for which the value is substituted in the editmatrix is assumed correct in the treated node. Finally, the vector of variables to be treated is updated.

In choiceRight, the same administrative chores are performed as in the

choiceLeft. The only difference is that in the right node a variable is eliminated from the editmatrix, and therefore assumed incorrect.

The editmatrix used here corresponds to edit e_1 and e_2 of Eqn. (16), which are the edits violated by the record $(x = 2, y = -1)$. As expected, a single call to bt$searchNext() yields the correct solution.

5 Related R-packages

The editrules packages provides methods to specify, modify and solve sets of linear constraints. Solving systems of linear constraints is the domain of linear programming (Schrijver, 1998). The comprehensive R Archive Network (CRAN, 2011) provides several R packages that use external libraries to solve linear programming problems. For example R packages linprog (Henningsen, 2010) and lpSolve (Berkelaar and others, 2011). editrules takes a different approach for a number of reasons.

First of all, the specification of constraints in editrules is in R syntax, while other packages typically use the specification format of the external library. This facilitates the maintenance of edits and reuse of these statements within R. It is very useful to check data within R before, during and after data analysis.

Secondly, De Waal et al. (2011), Chapter 3.4.9 compare various implementations of error localizers, based on specifically written branch-and-bound software and based on general linear solvers. They observed that branch and bound algorithms for error localization problems in realistic data are as fast as linear programming techniques, but have the added advantage of returning multiple equivalent solutions to the specified problem. errorLocalizer is an improved implementation of their original branch and bound algorithm.

Thirdly, editrules provides a powerful toolbox to write advanced editing and backtracking operations on sets of edits using R statements. Some linear programming libraries also offer branch and bound or branch and cut methods, but these typically have to be specified in the original programming language of the library. In editrules all coding is in R.

6 Conclusions

The editrules package offers an interface to define and manipulate sets of linear (in)equality restrictions. Linear restrictions can be entered textually for automated translation to matrix form or *vice versa*. Edit sets can be manipulated by value substitution or variable elimination, through a newly developed fast routine for Fourier-Motzkin elimination. The latter routine also allows the user to check sets of linear (in)equalities for internal consistency.

The package offers the ability to identify the edit rules violated by a set of records. Based on the generalized Fellegi-Holt assumption, one can localize the erroneous fields in edit-violating records. The error localization routines are based on a backtracker-programming paradigm which is exported to user space, providing users with a flexible and easy to use interface for solving binary programming problems.

```
> bt <- backtracker(
+     isSolution = {  # check for solution or pruning
+         w <- sum(weight[adapt])
+         if ( isObviouslyInfeasible(.E) || w > wsol ) return(FALSE)
+         if (length(.totreat) == 0){
+             wsol <<- w
+             adapt <- adapt
+             return(TRUE)
+         }
+     },
+     choiceLeft = {  # things to do in the left node
+         .var <- .totreat[1]
+         .E <- substValue(.E, .var , r[.var])
+         .totreat <- .totreat[-1]
+
+         adapt[.var] <- FALSE
+     },
+     choiceRight = { # things to do in the right node
+         .var <- .totreat[1]
+         .E <- eliminate(.E, .var)
+         .totreat <- .totreat[-1]
+
+         adapt[.var] <- TRUE
+     },
+     # Initialize variables in root node
+     .E = editmatrix(c("y > x-1 ","y > -x+3")),
+     .totreat = c("x","y"),
+     r = c(x=2,y=-1),
+     adapt = c(x=FALSE, y=FALSE),
+     weight = c(1,1),
+     wsol = 2
+ )
> bt$searchNext()

$w
[1] 1

$adapt
    x     y
FALSE  TRUE
```

Figure 9: Solving a simple error localization problem using the backtracker object directly.

References

Berkelaar, M. and others (2011). *lpSolve: Interface to Lp_solve v. 5.5 to solve linear/integer programs*. R package version 5.6.6.

Černikov, S. N. (1963). The solution of linear programming problem by elimination of unknowns. In *Soviet mathematics DOKLADY 2*.

CRAN (2011). Comprehensive R Archive Network. http://www.cran.r-project.org.

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.

Farkas, G. (1902). Über die Theorie der Einfachen Ungleichungen. *Journal für die reine und angewandte mathematik 124*, 1–27.

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

Fourier, J. (1826). Solution d'une question particulière du calcul des inégalités. *Oeuvres II*, 317–328.

Henningsen, A. (2010). *linprog: Linear Programming / Optimization*. R package version 0.9-0.

Knuth, D. (1968). *The Art of Computer Programming*, Volume 4A, Enumeration and Backtracking. Reading, Massachusetts: Addison-Wesley.

Kohler, D. (1967). Projections of convex polyhedral sets. Technical Report ORC 67-29, University of California, Berkely.

Kuhn, H. W. (1956). Solvability and consistency for linear equations and inequalities. *The American Mathematical Monthly 63*, 217–232.

Lipschutz, S. and M. Lipson (2000). *Linear algebra* (third ed.). Schaum's outline series. McGraw-Hill.

Mart-Oliet, N. and J. Mesguer (2002). Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science 285*, 121–154.

Moreau, P.-E. (1998, June). A choice-point library for backtrack programming. In *JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic*.

Motzkin, T. S. (1936). Beitrage zur Theorie der Linearen Ungleichungen. Inaugural Dissertation, Basel-Jerusalem.

Partington, V. (1997). Implementation of an imperative programming language with backtracking. Technical Report P9714, University of Amsterdam, Programming research group.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. ISBN 3-900051-07-0.

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.

Schrijver, A. (1998). *Theory of linear and integer programming*. Wiley-Interscience series in descrete mathematics and optimization. New York: John Wiley and Sons.

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

Van der Loo, M., E. De Jonge, and S. Scholtus (2011). Correction of rounding, typing and sign errors with the deducorrect package. Technical Report 201119, Statistics Netherlands. R package version 1.0-0.

Vittek, M. (1996). A compiler for nondeterministic term rewriting systems. In H. Ganziger (Ed.), *Proceedings of RTA'96*, Volume 1103 of *Lecture Notes in Computer Science*, New Brunswick (New Jersey), pp. 154–168. Springer-Verlag.

Williams, H. (1986). Fourier's method of linear programming and its dual. *The American mathematical monthly 93*, 681–695.

Index