Correction of rounding, typing, and sign errors with the `deducorrect` package

Mark van der Loo, Edwin de Jonge and Sander Scholtus
Package version 1.0.0

July 1, 2011

Abstract

Since raw (survey) data usually has to be edited before statistical analysis can take place, the availability of data cleaning algorithms is important to many statisticians. In this paper the implementation of three data correction methods in R are described. The methods of this package can be used to correct numerical data under linear restrictions for typing errors, rounding errors, sign errors and value interchanges. The algorithms, based on earlier work of Scholtus, are described and implementation details with coded examples are given. Although the algorithms have originally been developed with financial balance accounts in mind the algorithms are formulated generically and can be applied in a wider range of applications.

This vignette is a near-literal transcript of Van der Loo et al. (2011), which corresponds to package version `1.0-0`. Please refer to that paper in publications. The paper is included in the package. This vignette will be updated with the package when necessary.

Contents

List of Algorithms

1 Introduction

Raw statistical data is often plagued with internal inconsistencies and errors which inhibit reliable statistical analysis. Establishment survey data is particularly prone to in-record inconsistencies, because the numerical variables contained in these data are usually interrelated by many mathematical relationships. Before statistical analysis can take place, these relationships have to be checked and violations should be resolved as much as possible. While establishing that a record violates certain relationships is straightforward, deciding which fields in a record contain the actual errors can be a daunting task. In the past, much attention has been paid to this decision problem, often using Fellegi and Holt's principle (Fellegi and Holt, 1976) as the point of departure. This principle states that for non-systematic errors, and with no information on the cause of errors, one should try to make a record consistent by changing as few variables as possible.

This principle precludes using the data available in the (possibly erroneous) fields to detect and correct the error. In certain cases, naively applying Fellegi and Holt's principle will yield consistent records with nevertheless faulty data. As an example, consider a survey record with three variables x, y and z, which have to obey the relationship $x = y - z$. Such relationships frequently occur in financial profit-loss accounts. If a record happens to have values such that $x = z - y$, then Fellegi and Holt's principle suggests that either the numerical value of x, y or z should be adapted in such a way that the relationship holds, while the values in the record suggest that the values in fields y and z might have been interchanged. Swapping the values of z and y therefore seems a reasonable solution although it formally means changing two values.

This package provides three functions which do use the data in a record to detect and correct errors:

1. `correctRounding` corrects rounding errors in numerical records that cause violations of linear equality rules. The method works by making small changes to a large enough set of randomly chosen variables.

2. `correctTypos` corrects typing erros in numerical records that cause violations of linear equality rules. The method works by computing correction suggestions and checking which suggestions correspond to correcting a typing error.

3. `correctSigns` corrects sign flips and value swaps in numerical records which violates linear equality rules. The method minimizes the number of value swaps and sign flips via a binary programming formulation.

Both `correctTypos` and `correctSigns` are capable of taking account of possible rounding errors in the records.

1.1 Deductive correction

We use the term deductive correction to indicate methods which use information available in inconsistent records to deduce and solve the probable cause of error. Recently, a number of algorithms for deductive correction have been proposed by Scholtus (2008, 2009). These algorithms can solve problems not uncommon in numerical survey data, namely

- Rounding errors.
- Simple typing errors.
- Sign swaps and/or value interchanges.

The algorithms focus on solving problems in records that have to obey a set of linear relationships, each of which can be written as

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

Here, every $\mathbf{a}$ is a nonzero real vector, $\mathbf{x}$ a numerical data record and b a constant. In data-editing literature the restrictions imposed on records are often called edit rules, or edits in short. If an edit describes a relationship between a number of variables $\{x_j\}$, we say that the edit *contains* the variables $\{x_j\}$. Conversely, when x_j is part of a relationship defined by an edit we say that x_j *occurs* in the edit. We will denote a generic set of edits with E. The matrix representation of (in)equality parts of E will be denoted $\mathbf{A}$.

In this paper, we describe the `deducorrect` package for R (R Development Core Team, 2011), which implements (slight) generalizations of the algorithms proposed by Scholtus (2008, 2009). The purpose of this paper is to provide details on the algorithms and to familiarize users with the syntax of the package. For a detailed description of the available routines and their arguments we refer the reader to the reference manual that comes with the package.

The correction algorithms in the package report the results in a uniform matter. Section 1.2 provides details on the `deducorrect` output object which stores information on corrected records, applied corrections, and more. Sections 2, 3 and 4 provide details on the classes of problems that may be treated with the package, an exposition of the algorithms used and coded examples with analysis of the results. It is also shown how the examples from Scholtus (2008) and Scholtus (2009) can be treated with this software.

The package requires that linear relationships are defined with the `editrules` package (de Jonge and van der Loo, 2011). The `editrules` package offers functionality to define and manipulate sets of equality and inequality restrictions. With the `editrules` package, linear restrictions can be defined as R-statements (in `character` format) or as a matrix. As a convenience, one can define edits in any of the forms

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

Table 1: Contents of the deducorrect object. All slots can be accessed through the $ operator.

corrected	The input data with records corrected where possible.
corrections	A data.frame describing the corrections. Every record contains a row number, labeling the row in the input data, a variable name of the input data, the old value and the new value.
status	A data.frame with at least one column giving treatment information of every record in the input data. Depending on the correct function, some extra columns may be added.
timestamp	The date and time when the deducorrect object was created.
generatedby	The name of the function that called newdeducorrect to create the object.
user	The name of the user running R, deduced from the environment variables of the system using R.

and have it automatically translated to the form in (1). A short introduction to the editrules package is given in the appendix of this paper, but we refer the reader to the package documentation for more detailed information. Unless noted otherwise, all R-code examples in this paper can be executed from the R commandline after loading the deducorrect and editrules package.

Throughout, we denote the Euclidean vector norm with double bars $||\cdot||$ while single bars $|\cdot|$ denote the elementwise absolute values of the argument.

1.2 The deducorrect object and status values

Apart from the corrected records, every correct- function of the deducorrect package returns some logging information on the applied corrections. Information on applied corrections, a status indicator per record, a timestamp and user information are included and stored uniformly in a deducorrect object. See Table 1 for an overview of the contents of this object. Because of the large amount of information in a deducorrect object, the contents are summarized for printing to screen. In the example below, we define one record of data, a linear restriction in the form of an editmatrix, and apply the correctSigns correction method[1].

```
> (d <- data.frame(x = 1, y = 0, z = 1))

  x y z
1 1 0 1
```

[1]sometimes extra brackets are included to force R to print the result

```
> E <- editmatrix("x==y-z")
> sol <- correctSigns(E, d)
> sol

 deducorrect object generated by 'correctSigns' on Fri Jul  1 15:51:26 2011
 slots:  $corrected, $corrections, $status, $timestamp, $generatedby, $user

 Record status:
  invalid   partial corrected     valid       Sum
        0         0         1         0         1

 Variables corrected:
  x Sum
  1   1
```

The individual components of `sol` can be retrieved with the dollar-operator. The slot `corrected` is the same as the input data, but with corrected records, where possible:

```
> sol$corrected

   x y z
1 -1 0 1
```

The applied corrections are stored in the `corrections` slot.

```
> sol$corrections

  row variable old new
1   1        x   1  -1
```

Every row in `corrections` tells which variable in which row of the input data was changed, and what the old and new values are. The `status` slot gives details on the status of the record.

```
> sol$status

     status weight degeneracy nflip nswap
1 corrected      1          2     1     0
```

The first column is an indicator which can take five different values, indicating whether validity could be established, and/or if the record could be (partially) corrected by the method which created the deducorrect object. These values are (see Table 2 for an overview per `correct`-function):

- valid: The record violates none of the edit rules defined by the user.
- corrected: The record violated one or more edit rules but the `correct`-function could adapt the record so no rules are violated afterwards.

Table 2: The number of equalities n and inequalities m violated by an edit, before and after treatment with one of the correct-functions of `deducorrect`. The label N/A indicates that this status value does not occur for tat function. (Note that is is not the same as NA, which occurs when validity could not be established because the record has missing values.) As an example, consider the fourth row. In this case, a record enters a `correct`-function with n linear equality violations. After being treated by the function less than n, but more than 0 edit violations remain. For `correctSigns`, this situation cannot occur: the method tries to find a comlete solution. Both `correctRounding` and `correctTypos` allow for partially repairing a record, so in their case, the status is labeled "partial".

Before		After		status		
Eqs	Ineqs	Eqs	Ineqs	`correctSigns`	`correctRounding`	`correctTypos`
0	0	0	0	valid	valid	valid
0	m	0	m	invalid	invalid	invalid
n	0	n	0	invalid	invalid	invalid
n	0	$< n$	0	N/A	partial	partial
n	0	0	0	corrected	corrected	corrected
n	m	n	m	invalid	invalid	invalid
n	m	$< n$	0	N/A	partial	partial
n	m	$< n$	$< m$	N/A	partial	partial
n	m	0	0	corrected	corrected	corrected

- partial: The record violated one ore more edit rules. Some, but not all violations could be repaired.
- invalid: The records violates one or more edit rules. None of them could be repaired.
- NA: The record contains missing values, therefore edit violation cannot be establised.

The other columns of the `status` slot depend on the function which created the object and can provide more details on the chosen solutions. These are described in the coming sections.

1.3 Balance accounts and totally unimodular matrices

Most algorithms described here have been designed with financial balance accounts in mind. The balance accounts encountered in establishment surveys mostly involve integer records since financial amounts are usually reported in currency (kilo-)units. Therefore, linear edit rules of the form

$$\mathbf{Ax} = \mathbf{b} \text{ with } \mathbf{A} \in \{-1, 0, 1\}^{m\times n},\ \mathbf{x} \in \mathbb{Z}^n,\ \text{ and } \mathbf{b} \in \mathbb{Z}^m, \tag{3}$$

are frequently encountered. In all the examples of financial balance accounts encountered by the authors, the matrix $\mathbf{A}$ happened to be totally unimodular. A (not necessarily square) matrix is called *totally unimodular* when every square submatrix has determinant -1, 0, or 1. The scapegoat algorithm (Scholtus, 2008), which is used in the `correctRounding` function, requires $\mathbf{A}$ to be totally unimodular. See appendix B of Scholtus (2008) for a further discussion of total unimodularity. The `deducorrect` package offers the function `isTotallyUnimodular` which checks if a matrix is totally unimodular. The algorithm follows a recursive procedure given below.

```
procedure isTotallyUnimodular(A)
    A ← reduceMatrix(A)
    if A = ∅ then
        return TRUE
    else if Each column of A has exactly 2 nonzero elements then
        return hellerTompkins(A)
    else
        𝒜 ← raghavachari(A)
        if  Every A ∈ 𝒜 isTotallyUnimodular(A)  then
            return TRUE
        else
            return FALSE
        end if
    end if
end procedure
```

Here, reduceMatrix iteratively removes all rows and columns of $\mathbf{A}$ which have at most one nonzero element (an operation of $\mathcal{O}(n)$ in the number of columns and rows). When possible, the criterium of Heller and Tompkins (1956), which is $\mathcal{O}(2^n)$ in the number of columns is used to determine unimodularity. If this is not possible, a set of smaller matrices $\mathcal{A}$ is derived with the method of Raghavachari (1976). Every matrix in $\mathcal{A}$ is subsequently checked for total unimodularity by calling isTotallyUnimodular. In the worst case, Raghavachari's method must be called recursively and checking for unimodularity is $\mathcal{O}(n!)$ in the number of columns. For this reason, our implementation is set up so that Raghavachari's method is used only after the reduction method and the Heller-Tompkins method have been tried. Also, matrices are transposed to make sure that n is minimized in every step. In practical applications $\mathbf{A}$ is often fairly sparse and only a small portion of $\mathbf{A}$ has to be treated with the Raghavachari method.

2 `correctRounding`

2.1 Area of application

This function can be used to correct violations of linear equality restrictions because of rounding errors in one or more variables. The rounding errors are assumed to be measurement errors rather than rounding errors caused by machine computation. Rounding errors caused at measurement are on the order of a unit of measurement, much larger than errors caused by machine computation. The linear equality restrictions must be of the form

$$\mathbf{Ax} = \mathbf{b} \text{ with } \mathbf{A} \in \{-1, 0, 1\}^{m\times n},\ \mathbf{x} \in \mathbb{Z}^n,\ \text{ and } \mathbf{b} \in \mathbb{Z}^m,$$

where $\mathbf{A}$ is a totally unimodular matrix (see Section 1.3), which can be tested with the function `isTotallyUnimodular`. Linear inequalities with real coefficients can be imposed as well. The `correctRounding` function will only return solutions which do not violate any extra inequality violations.

2.2 How it works

The `correctRounding` function uses the scapegoat algorithm described in Scholtus (2008) to suggest corrections for linear equality violations. Linear inequalities are ignored, except that corrections which cause new inequality violations are not accepted. The algorithm first selects linear edit rules violated by rounding errors. Rounding errors cause small deviations from equality and therefore deviations smaller than some ε (say, $\varepsilon = 2$) are assumed to stem from rounding errors. Next, a number of variables –called scapegoat variables– are selected randomly in such a way that rounding errors can be solved exactly and uniquely by altering the drawn scapegoat variables. Note that the number of scapegoat variables is not fixed and may vary over drawings. If the chosen solution happens to cause new inequality violations, the solution is rejected and a new set of scapegoat variables is drawn. This is repeated at most k times. See Algorithm 1 for a concise description of the basic procedure (without checking for inequalities).

2.3 Examples

Here, we will reproduce the example of Scholtus (2008), Section 5.3.2. Consider an integer-valued record with 11 variables, subject to the rules:

```
> E <- editmatrix( c("X1 + X2 == X3"
+                   ,"X2 == X4"
+                   ,"X5 + X6 + X7 == X8"
+                   ,"X3 + X8 == X9"
+                   ,"X9 - X10 == X11"))
```

Consider also the following inconsistent record:

Algorithm 1 Scapegoat algorithm

Input: Equality restriction matrix $\mathbf{A}$ and constant vector $\mathbf{b}$, record $\mathbf{x}$, rounding tolerance ε.

1: Remove rows from the system $\mathbf{Ax} = \mathbf{b}$ not satisfying $|\mathbf{a} \cdot \mathbf{x} - b| < \varepsilon$.
2: **if** $\mathbf{A} \neq \varnothing$ **and** $||\mathbf{Ax} - \mathbf{b}|| > 0$ **then**
3: Randomly permute columns of $\mathbf{A}$. Permute $\mathbf{x}$ accordingly.
4: Use QR decomposition to partition $\mathbf{A}$ columnwise in a square invertible matrix $\mathbf{A}_1$ and remaining columns $\mathbf{A}_2$. Partition $\mathbf{x}$ in $\mathbf{x}_1$ and $\mathbf{x}_2$ accordingly.
5: $\mathbf{x}_1 \leftarrow \mathbf{A}_1^{-1}(\mathbf{b} - \mathbf{A}_2\mathbf{x}_2)$
6: Unpermute $[\mathbf{x}_1, \mathbf{x}_2]$
7: **end if**
8: Restore $\mathbf{x}$ by adding the previously removed elements.

Output: $\mathbf{x}$

```
> (dat <- data.frame(t(c(12, 4, 15, 4, 3, 1, 8, 11, 27, 41, -13))))

  X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11
1 12  4 15  4  3  1  8 11 27  41 -13

> violatedEdits(E, dat)

       e1    e2   e3   e4   e5
[1,] TRUE FALSE TRUE TRUE TRUE
```

As reported by the `violatedEdits` function, this record violates edit rules 1, 3, 4, and 5.

Repairing the record can be done with

```
> set.seed(1)
> sol <- correctRounding(E, dat)
> cbind(sol$corrected, sol$status)

  X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11    status attempts
1 12  4 16  4  3  1  8 12 28  41 -13 corrected        1

> sol$corrections

   row variable old new
X3   1       X3  15  16
X8   1       X8  11  12
X9   1       X9  27  28
```

Here, we used `set.seed` to make results reproducible. The result is not exactly the same as the solution found in the reference. Here, variables x_3, x_8 and x_9 have been adapted, while in the reference x_3, x_4, x_8, x_9, and x_{10} were adapted. Since corrections are very small, smearing out the effect of adaptations over a number of variables is a reasonable option.

3 `correctTypos`

3.1 Area of application

This function can be used to correct typographical errors in an integer record. Examples of typographical errors include extra or to few digits, digit permutations and/or digit substitutions. To be precise, the method can be applied to integer records $\mathbf{x}$ which violate linear equality constraints as in Eq. (3):

$$\mathbf{Ax} = \mathbf{b} \text{ with } \mathbf{A} \in \{-1, 0, 1\}^{m\times n},\ \mathbf{x} \in \mathbb{Z}^n, \text{ and } \mathbf{b} \in \mathbb{Z}^m.$$

In fact, the function will also run when $\mathbf{A} \in \mathbb{R}^{m\times n}$. However, the nature of the algorithm is such that it is unlikely that typing errors will be found for such systems. The algorithm was developed with sets of financial balance equations in mind, where these type of problems are very common. As far as inequalities are concerned, they are currently ignored by the algorithm, in the sense that no attempt is made to repair inequality violations. However, the algorithm does not generate solutions causing extra inequality violations.

The function has a parameter ε which allows for a tolerance so that rounding errors can be ignored. The default value of ε is almost zero: it is set to the square root of `.Machine$double.eps` which amounts to approximately 10^{-8}. The value should be increased, to 2 units of measurement for example, to allow for rounding errors that are caused by measurement rather then machine computation. This way, records containing just rounding errors can be ignored by `correctTypos` but do note that in that case they will receive the status `valid`, since no typos were found.

3.2 How it works

In short, the algorithm first computes a list of suggestions which correct one or more violated edits (Algorithm 2). The corrections not corresponding to a typographical error are removed, after which the set of suggestions that maximize the number of satisfied edit rules is determined (Algorithm 3).

Suggestions are generated for the set of variables which *only* occur in violated edits since altering these variables will have no effect on already satisfied edits. For every variable x_j, define the matrix $\mathbf{A}^{(j)}$ who's rows represent edits containing x_j. Suggestions $\tilde{x}_j^{(i)}$ for every row i of $\mathbf{A}^{(j)}$ can be generated by solving for x_j:

$$\tilde{x}_j^{(i)} = \frac{1}{A_{ij}^{(j)}}\left(b_i - \sum_{j'\neq j} A_{ij'}^{(j)} x_j\right). \tag{4}$$

We keep only the unique suggestions, and reject solutions which are more than a certain Damerau-Levenshtein distance removed from the original

Algorithm 2 Generate solution candidates

Input: Record $\mathbf{x}$, a set of linear equality restrictions and a list of variables to `fixate`. A maximum Damerau-Levenshtein distance `maxdist`.

$L \leftarrow \varnothing$

Determine $J_0 = \{j : x_j$ occurs *only* in violated edits and not in `fixate`$\}$

for $j \in J_0$ **do**

  Determine the matrix $\mathbf{A}^{(j)}$ of violated edits containing x_j and associated constant vector $\mathbf{b}^{(j)}$

  for every row i of $A^{(j)}$ **do**

    $\tilde{x}_j^{(i)} \leftarrow \left(b_i^{(j)} - \sum_{j' \neq j} A_{ij'}^{(j)} x_{j'}\right) / A_{ij}^{(j)}$

    $L \leftarrow L \cup \tilde{x}_j^{(i)}$ ▷ Only new values are added

  end for

end for

Remove $\tilde{x}_j^{(i)}$ from L for which $d_{\text{DL}}(\tilde{x}_j^{(i)}, x_j) > \texttt{maxdist}$

Output: List L of m unique solution suggestions for record $\mathbf{x}$.

value. The *Damerau-Levenshtein* distance d_{DL} between two strings s and t is the minimum number of character insertions, deletions, substitutions and transpositions necessary to change s into t or *vice versa* (Damerau, 1964; Levenshtein, 1966). The remaining set of suggestions $\{x_j^{(i)}\}$ will in general contain multiple suggestions for each violated edit i and multiple suggestions for each variable x_j. Using a tree search algorithm, a subset of $\{x_j^{(i)}\}$ is selected which maximizes the number of resolved edits. The tree search is sped up considerably by pruning branches which resolve the same edit multiple times or use multiple suggestions for the same variable. When multiple solutions are found, only the variables which obtain the same correction suggestion in each solution are adapted.

This algorithm generalizes the algorithms of Scholtus (2009) in the following two ways: first, the imposed linear restrictions are generalized from $\mathbf{Ax} = \mathbf{0}$ to $\mathbf{Ax} = \mathbf{b}$. Secondly, the original algorithm allowed for a single *digit* insertion, deletion, transposition or substitution. The more general Damerau-Levenshtein distance used here treats the digits as characters, allowing for sign changing, which is forbidden if only digit changes are allowed. Also, by applying a standard Damerau-Levenshtein algorithm it is easy to allow for corrections spanning larger values d_{DL}. That is, one could allow for multiple typos in a single field. Moreover, the Damerau-Levenshtein distance as implemented in the `deducorrect` package allows one to define different weights to the four types of operations involved, adding some extra flexibility to the method.

Algorithm 3 Maximize number of resolved edits

Input: Record $\mathbf{x}$, a list of linear equality restrictions and a list of solution suggestions $L = \{L_\ell = \tilde{x}^{(i_\ell)}_{j_\ell} : \ell = 1, 2, \ldots, m\}$

$k \leftarrow 0$
$s \leftarrow \mathsf{NULL}$
procedure $\textsc{tree}(\mathbf{x}, L)$
  if $L \neq \varnothing$ **then**
    $\textsc{tree}(\mathbf{x}, L \backslash L_1)$ ▷ Left branch: don't use suggestion
    $x_{j_1} \leftarrow L_1$ ▷ Right branch: use suggestion
    $L \leftarrow L \backslash \{x^{(i_\ell)}_{j_\ell} \in L : j_\ell = j_1$ **or** $x^{(i_\ell)}_{j_\ell}$ occurs in same edit as $L_1\}$
    $\textsc{tree}(\mathbf{x}, L)$
  else
    if Number of edits n resolved by x larger then k **then**
      $k \leftarrow n$
      $s \leftarrow x$
    end if
  end if
end procedure

Output: (partial) solution s, resolving maximum number of edits.

3.3 Examples

In this section we show the most important options of the `correctTypos` function. After a simple, worked-out example we reproduce the results in Chapter 4 of Scholtus (2009).

First, define a simple one-record dataset with an associated edit rule.

```
> dat <- data.frame(x = 123, y = 192, z = 252)
> (E <- editmatrix("z == x + y"))

Edit matrix:
    x  y z Ops CONSTANT
e1 -1 -1 1  ==        0

Edit rules:
e1 : z == x + y
```

Obviously, the edit in `E` is not satisfied since $123+192 = 315$. As can be seen from the output of `editmatrix`, we have $b = 0$, so the correction candidates here are:

$$\tilde{x}^{(1)} = 0 - \frac{-1 \cdot 192 + 1 \cdot 252}{-1} = 60 \quad (5)$$

$$\tilde{y}^{(1)} = 0 - \frac{-1 \cdot 123 + 1 \cdot 252}{-1} = 129 \quad (6)$$

$$\tilde{z}^{(1)} = 0 - \frac{-1 \cdot 123 - 1 \cdot 192}{1} = 315 \quad (7)$$

The Damerau-Levenshtein distances between the candidates and their originals are given by:

$$d_{\mathrm{DL}}(\tilde{x}^{(1)}, x) = d_{\mathrm{DL}}(60, 123) = 3 \text{ (two substitutions and an insertion)} \tag{8}$$
$$d_{\mathrm{DL}}(\tilde{y}^{(1)}, y) = d_{DL}(129, 192) = 1 \text{ (one transposition)} \tag{9}$$
$$d_{\mathrm{DL}}(\tilde{z}^{(1)}, z) = d_{DL}(315, 252) = 3 \text{ (three substitutions)} \tag{10}$$

In this case, there is just one candidate with $d_{\mathrm{DL}} = 1$, solving the inconsistency with just one digit transposition. Running the record through `correctTypos` indeed finds the digit transposition:

```
> correctTypos(E, dat)$corrected

    x   y   z
1 123 129 252
```

Scholtus (2009) (Chapter 4) treats a series of examples which we will reproduce here. We consider a dataset with 11 variables, subject to the following edit rules.

```
> E <- editmatrix( c("x1 + x2 == x3"
+                   ,"x2 == x4"
+                   ,"x5 + x6 + x7 == x8"
+                   ,"x3 + x8 == x9"
+                   ,"x9 - x10 == x11"))
```

The following dataframe contains the correct record (`example 4.0`) as well as the manipulated erroneous records.

```
> dat

              x1  x2   x3  x4  x5 x6 x7  x8    x9  x10 x11
example 4.0 1452 116 1568 116 323 76 12 411  1979 1842 137
example 4.1 1452 116 1568 161 323 76 12 411  1979 1842 137
example 4.2 1452 116 1568 161 323 76 12 411 19979 1842 137
example 4.3 1452 116 1568 161   0  0  0 411 19979 1842 137
example 4.4 1452 116 1568 161 323 76 12   0 19979 1842 137
```

This `data.frame` can be read into R by copying the code from the `correctTypos` help page. As can be seen, example 4.1 has a single digit transposition in x_4, example 4.2 has the same error, and an extra 9 inserted in x_9, example 4.3 contains multiple extra errors (in x_5, x_6 and x_7 which cannot be explained by simple typing errors. Finally, example 4.4 also has multiple errors which cannot all be explained by simple typing errors. This example has multiple solutions which solve an equal amount of errors.

The violated edit rules may be listed with the function

```
> violatedEdits(E, dat)
```

```
        e1    e2    e3    e4    e5
[1,] FALSE FALSE FALSE FALSE FALSE
[2,] FALSE  TRUE FALSE FALSE FALSE
[3,] FALSE  TRUE FALSE  TRUE  TRUE
[4,] FALSE  TRUE  TRUE  TRUE  TRUE
[5,] FALSE  TRUE  TRUE  TRUE  TRUE
```

Now, to apply as many typo-corrections as possible:

```
> sol <- correctTypos(E, dat)
> cbind(sol$corrected, sol$status)
```

```
              x1  x2   x3  x4  x5 x6 x7  x8    x9  x10 x11    status
example 4.0 1452 116 1568 116 323 76 12 411  1979 1842 137     valid
example 4.1 1452 116 1568 116 323 76 12 411  1979 1842 137 corrected
example 4.2 1452 116 1568 116 323 76 12 411  1979 1842 137 corrected
example 4.3 1452 116 1568 116   0  0  0 411  1979 1842 137   partial
example 4.4 1452 116 1568 116 323 76 12   0 19979 1842 137   partial
```

Our implementation finds the exact same solutions as in the original paper of Scholtus (2009). Also see this reference for a thorough analysis of the results.

4 correctSigns

4.1 Area of application

This function can be used to solve sign errors and value swaps which cause linear equalities to fail. Possible presence of linear inequalities are taken into account when resolving errors, but they are not part of the error detection process. The function has an argument ε which allows one to ignore rounding errors that can mask sign errors and value swaps. The standard value is the square root of machine accuracy (`.Machine$double.eps`). It should be increased to 2 units of measurement or more to account for rounding errors caused by measurement.

4.2 How it works

The function `correctSigns` tries to change the sign of (combinations of) variables and/or swap the order of variables to repair inconsistent records. Sign flips and value swaps are closely related since

$$-(x-y) = y - x, \tag{11}$$

These simple linear relations frequently occur in profit-loss accounts for example. Basically, `correctSigns` first tries to correct a record by changing one sign. If that doesn't yield any solution, it tries changing two, and so on. If the user allows value swaps as well, it starts by trying to correct the record with a single sign flip or variable swap. If no solution is found, all combinations of two such actions are tried, and so on. The algorithm only treats the variables which have nonzero coefficients in one of the violated equality constraints. Since the number of combinations grows exponentially with the number of variables to treat, the user is given some control over the volume of the search space to cover in a three different ways.

1. The variables which are allowed to flip signs or variable pairs which may be interchanged simultaneously can be determined by the user. Knowledge of the origin of the data will usually give a good idea on which variables are prone to sign errors. For example, in surveys on profit-loss accounts, respondents sometimes erroneously submit the cost as a negative number.

Once variables which may change sign, and variable pairs which may be permuted are determined, the number of combinations may still become large. If there are n possible sign flips and value swaps, there are $\sum_k \binom{n}{k} = 2^n$ possible repair actions in total. The second option allows the user to

2. limit the maximum number k of simultaneous sign flips and/or value swaps that may be tested. This is controlled by the `maxActions` parameter in Algorithm 4.

Since the function tries to repair the record with $k = 1$, $k = 2, \ldots$, an extra control parameter allows the user to

3. exit the function when the number of combinations $\binom{n}{k}$ becomes too large. This is controlled by the `maxCombinations` parameter in Algorithm 4.

To account for sign errors and variable swap errors which are masked by rounding errors, the user can provide a nonnegative tolerance ε, so the set of equality constraints are checked as

$$|\mathbf{Ax} - \mathbf{b}| < \varepsilon, \tag{12}$$

elementwise.

The function tries to find and apply the minimal number of actions (sign flips and/or variable swaps) necessary to repair the record. It is not guaranteed that a solution exists, nor that the solution is unique. If multiple solutions are found, the solution which minimizes a weight is chosen. The user has the option to assign weights to every variable, or to every action. The total weight of a solution is the sum over the weights of the altered variables or the sum over the weight of the actions performed. Actions with higher weight are therefore less likely to be performed and variables with higher weight are less likely to be altered.

Algorithm 4 Record correction for `correctSigns`

Input: A numeric record $\mathbf{x}$, a tolerance ε. A set of equality and inequality constraints of the form

$$\begin{aligned}\mathbf{Ax} &= \mathbf{b}\\ \mathbf{Bx} &\leq \mathbf{c},\end{aligned}$$

A list `flip` of variables of which the signs may be flipped, a list `swap` of variable pairs of which the values may be interchanged, an integer `maxActions`, an integer `maxCombinations` and a weight vector.

Create a list `actions`, of length n containing those elements of `flip` and `swap` that affect variables that occur in violated rows of A.
Create an empty list S.
$k \leftarrow 0$
while $S = \varnothing$ **and** $k < \min(\texttt{maxActions}, n)$ **do**
  if not $\binom{n}{k} > \texttt{maxCombinations}$ **then**
    $k \leftarrow k + 1$
    Generate all $\binom{n}{k}$ combinations of k actions.
    Loop over those combinations, applying them to x. Add solutions obeying $|\mathbf{Ax} - \mathbf{b}| < \varepsilon$ and $\mathbf{Bx} \leq \mathbf{c}$ to S.
  end if
end while
if not $S = \varnothing$ **then**
  Compute solution weights and choose solution with minimum weight. Choose the first solution in the case of degeneracy.
end if
Apply the chosen solution, if any, to $\mathbf{x}$.

Output: $\mathbf{x}$

This algorithm is a generalization of the original algorithms in Scholtus (2008) in two ways. First, the original algorithm was designed with a specific type of profit-loss account in mind, while the algorithm of `deducorrect` can handle any set of linear equalities. Second, the original algorithm was not designed to take account of inequality restrictions, which is a feature of the algorithm in this work. In Section 4.4 it is shown how the results of the original example can be reproduced.

4.3 Some simple examples

In this section we walk through most of the options of the `correctSigns` function. We will work with the following six records as example.

```
> (dat <- data.frame(
+     x = c( 3, 14, 15,  1, 17, 12.3),
+     y = c(13, -4,  5,  2,  7, -2.1),
+     z = c(10, 10,-10, NA, 10, 10 )))
```

```
     x    y   z
1  3.0 13.0  10
2 14.0 -4.0  10
3 15.0  5.0 -10
4  1.0  2.0  NA
5 17.0  7.0  10
6 12.3 -2.1  10
```

We subject this data to the rule

$$z = x - y. \tag{13}$$

With the `editrules` package, this rule can be parsed to an `editmatrix`.

```
> E <- editmatrix(c("z == x-y"))
```

Obviously, not all records in `dat` obey this rule. This can be checked with a function from the `editrules` package:

```
> cbind(dat, violatedEdits(E, dat))

     x    y   z    e1
1  3.0 13.0  10  TRUE
2 14.0 -4.0  10  TRUE
3 15.0  5.0 -10  TRUE
4  1.0  2.0  NA    NA
5 17.0  7.0  10 FALSE
6 12.3 -2.1  10  TRUE
```

Records 1, 2, 3 and 6 violate the editrule, record 5 is valid and for record 4 validity cannot be established since it has no value for z. If `correctSigns` is called without any options, all variables x, y and z can be sign-flipped:

```
> sol <- correctSigns(E, dat)
> cbind(sol$corrected, sol$status)

     x    y   z    status weight degeneracy nflip nswap
1  3.0 13.0 -10 corrected      1          1     1     0
2 14.0  4.0  10 corrected      1          1     1     0
3 15.0  5.0  10 corrected      1          1     1     0
4  1.0  2.0  NA      <NA>      0          0     0     0
5 17.0  7.0  10     valid      0          0     0     0
6 12.3 -2.1  10   invalid      0          0     0     0

> sol$corrections

  row variable old new
1   1        z  10 -10
2   2        y  -4   4
3   3        z -10  10
```

So, the first three records have been corrected by flipping the sign of z, y and z respectively. Since no weight parameter was given, the `weight` in the output is just the number of variables whose have been sign-flipped. The `degeneracy` column records the number of solutions with equal weight that were found for each record. Record 4 is not treated, since validity could not be established, record 5 was valid to begin with and record 6 could not be repaired with sign flips. However, record 6 seems to have a rounding error. We can try to accomodate for that by allowing a tolerance when checking equalities.

```
> sol <- correctSigns(E, dat, eps = 2)
> cbind(sol$corrected, sol$status)

     x    y   z    status weight degeneracy nflip nswap
1  3.0 13.0 -10 corrected      1          1     1     0
2 14.0  4.0  10 corrected      1          1     1     0
3 15.0  5.0  10 corrected      1          1     1     0
4  1.0  2.0  NA      <NA>      0          0     0     0
5 17.0  7.0  10     valid      0          0     0     0
6 12.3  2.1  10 corrected      1          1     1     0

> sol$corrections

  row variable   old   new
1   1        z  10.0 -10.0
2   2        y  -4.0   4.0
3   3        z -10.0  10.0
4   6        y  -2.1   2.1
```

Indeed, changing the sign of y in the last record brings the record within the allowed tolerance. Suppose that we have so much faith in the value of z, that we do not wish to change its sign. This can be done with the `fixate` option:

```
> sol <- correctSigns(E, dat, eps = 2, fixate = "z")
> cbind(sol$corrected, sol$status)

      x     y   z    status weight degeneracy nflip nswap
1  -3.0 -13.0  10 corrected      2          1     2     0
2  14.0   4.0  10 corrected      1          1     1     0
3 -15.0  -5.0 -10 corrected      2          1     2     0
4   1.0   2.0  NA      <NA>      0          0     0     0
5  17.0   7.0  10     valid      0          0     0     0
6  12.3   2.1  10 corrected      1          1     1     0

> sol$corrections

  row variable  old   new
1   1        x  3.0  -3.0
```

```
2   1        y 13.0 -13.0
3   2        y -4.0   4.0
4   3        x 15.0 -15.0
5   3        y  5.0  -5.0
6   6        y -2.1   2.1
```

Indeed, we now find solutions whitout changing z, but at the price of more sign flips. By the way, the same result could have been obtained by

```
> correctSigns(E, dat, flip = c("x", "y"))
```

The sign flips in record one and three have the same effect of a variable swap. Allowing for swaps can be done as follows.

```
> sol <- correctSigns(E, dat, swap=list(c("x","y")),
+     eps=2, fixate="z")
> cbind(sol$corrected, sol$status)

     x    y   z    status weight degeneracy nflip nswap
1 13.0  3.0  10 corrected      1          1     0     1
2 14.0  4.0  10 corrected      1          1     1     0
3  5.0 15.0 -10 corrected      1          1     0     1
4  1.0  2.0  NA      <NA>      0          0     0     0
5 17.0  7.0  10     valid      0          0     0     0
6 12.3  2.1  10 corrected      1          1     1     0

> sol$corrections

  row variable  old  new
1   1        x  3.0 13.0
2   1        y 13.0  3.0
3   2        y -4.0  4.0
4   3        x 15.0  5.0
5   3        y  5.0 15.0
6   6        y -2.1  2.1
```

Notice that apart from swapping, the algorithm still tries to correct records by flipping signs. What happened here is that the algorithm first tries to flip the sign of x, then of y, and then it tries to swap x and y. Each is counted as a single action. If no solution is found, it starts trying combinations. In this relatively simple example the result turned out well. In cases with more elaborate systems of equalities and inequalities, the result of the algorithm becomes harder to predict for users. It is therefore in general advisable to

- Use as much knowledge about the data as possible to decide which variables to flip sign and which variable pairs to swap. The problem treated in section 4.4 is a good example of this.
- Keep `flip` and `swap` disjunct. It is better to run the data a few times times through `correctSigns` with different settings.

Not allowing any sign flips can be done with the option `flip=c()`.

```
> sol <- correctSigns(E, dat, flip = c(), swap = list(c("x", "y")))
> cbind(sol$corrected, sol$status)

     x    y   z    status weight degeneracy nflip nswap
1 13.0  3.0  10 corrected      1          1     0     1
2 14.0 -4.0  10   invalid      0          0     0     0
3  5.0 15.0 -10 corrected      1          1     0     1
4  1.0  2.0  NA      <NA>      0          0     0     0
5 17.0  7.0  10     valid      0          0     0     0
6 12.3 -2.1  10   invalid      0          0     0     0

> sol$corrections

  row variable old new
1   1        x   3  13
2   1        y  13   3
3   3        x  15   5
4   3        y   5  15
```

This yields less corrected records. However running the data through

```
> correctSigns(E, sol$corrected, eps = 2)$status

     status weight degeneracy nflip nswap
1     valid      0          0     0     0
2 corrected      1          1     1     0
3     valid      0          0     0     0
4      <NA>      0          0     0     0
5     valid      0          0     0     0
6 corrected      1          1     1     0
```

will fix the remaining edit violations. The last two statements are easier to interpret than the one before that.

4.4 Sign errors in a profit-loss account

Here, we will work through the example of chapter 3 of Scholtus (2008). This example considers 4 records, labeled case a, b, c, and d, which can be defined in `R` as

```
> dat <- data.frame(
+    case = c("a","b","c","d"),
+    x0r = c(2100,5100,3250,5726),
+    x0c = c(1950,4650,3550,5449),
+    x0  = c( 150, 450, 300, 276),
+    x1r = c(   0,   0, 110,  17),
+    x1c = c(  10, 130,  10,  26),
+    x1  = c(  10, 130, 100,  10),
```

```
+     x2r = c(  20,  20,  50,   0),
+     x2c = c(   5,   0,  90,  46),
+     x2  = c(  15,  20,  40,  46),
+     x3r = c(  50,  15,  30,   0),
+     x3c = c(  10,  25,  10,   0),
+     x3  = c(  40,  10,  20,   0),
+     x4  = c( 195, 610,-140, 221))
```

A record consists of 4 balance accounts of which the results have to add up to a total. Each $x_{i,r}$ denotes some kind of revenue, x_{ic} some kind of cost and x_i the difference $x_{i,r} - x_{i,c}$. There are operating, financial, provisions and exeptional incomes and expenditures. The differences x_0, x_1, x_2 and x_3 have to add up to a given total x_4. These linear restrictions must be defined with the use of the editrules package.

```
> E <-editmatrix(c(
+      "x0 == x0r - x0c",
+      "x1 == x1r - x1c",
+      "x2 == x2r - x2c",
+      "x3 == x3r - x3c",
+      "x4 == x0 + x1 + x2 + x3"))
> E

Edit matrix:
   x0 x0c x0r x1 x1c x1r x2 x2c x2r x3 x3c x3r x4 Ops CONSTANT
e1  1   1  -1  0   0   0  0   0   0  0   0   0  0  ==        0
e2  0   0   0  1   1  -1  0   0   0  0   0   0  0  ==        0
e3  0   0   0  0   0   0  1   1  -1  0   0   0  0  ==        0
e4  0   0   0  0   0   0  0   0   0  1   1  -1  0  ==        0
e5 -1   0   0 -1   0   0 -1   0   0 -1   0   0  1  ==        0

Edit rules:
e1 : x0 + x0c == x0r
e2 : x1 + x1c == x1r
e3 : x2 + x2c == x2r
e4 : x3 + x3c == x3r
e5 : x4 == x0 + x1 + x2 + x3
```

Checking which records violate what edit rules can be done with the violatedEdits function of editrules.

```
> violatedEdits(E, dat)

        e1    e2    e3    e4    e5
[1,] FALSE  TRUE FALSE FALSE  TRUE
[2,] FALSE  TRUE FALSE  TRUE FALSE
[3,]  TRUE FALSE  TRUE FALSE  TRUE
[4,]  TRUE  TRUE  TRUE FALSE  TRUE
```

So record 1 (case a) for example, violates the restrictions e_1: $x_1 = x_{1,r} - x_{1,c}$ and e_5, $x_0 + x_1 + x_2 + x_3 = x_4$. We can try to solve the inconsistencies by allowing the following flips and swaps:

```
> swap <- list(
+     c("x1r","x1c"),
+     c("x2r","x2c"),
+     c("x3r","x3c"))
> flip <- c("x0","x1","x2","x3","x4")
```

Trying to correct the records by just flipping and swapping variables indicated above corresponds to trying to solve the system of equations

$$\left\{\begin{array}{lcl} x_0 s_0 &=& x_{0,r} - x_{0,c} \\ x_1 s_1 &=& (x_{1,r} - x_{1,c}) t_1 \\ x_2 s_2 &=& (x_{2,r} - x_{2,c}) t_2 \\ x_3 s_3 &=& (x_{3,r} - x_{3,c}) t_3 \\ x_4 s_4 &=& x_0 s_0 + x_1 s_1 + x_2 s_2 + x_3 s_3 \\ \multicolumn{3}{l}{(s_0, s_1, s_2, s_3, s_4, t_1, t_2, t_3) \in \{-1, 1\}^8} \end{array}\right. , \tag{14}$$

where every s_i corresponds to a sign flip and t_j corresponds to a value swap, see also Eqn. (3.4) in Scholtus (2008). Using the `correctSigns` function, we get the following.

```
> cor <- correctSigns(E, dat, flip = flip, swap = swap)
> cor$status

     status weight degeneracy nflip nswap
1 corrected      1          1     1     0
2 corrected      2          1     0     2
3 corrected      2          1     1     1
4   invalid      0          0     0     0
```

As expected from the example in the reference, the last record could not be corrected because the solution is masked by a rounding errors. This can be solved by allowing a tolerance of two measurements units.

```
> cor <- correctSigns(E, dat, flip = flip, swap = swap, eps = 2)
> cor$status

     status weight degeneracy nflip nswap
1 corrected      1          1     1     0
2 corrected      2          1     0     2
3 corrected      2          1     1     1
4 corrected      2          1     2     0

> cor$corrected
```

	case	x0r	x0c	x0	x1r	x1c	x1	x2r	x2c	x2	x3r	x3c	x3	x4
1	a	2100	1950	150	0	10	-10	20	5	15	50	10	40	195
2	b	5100	4650	450	130	0	130	20	0	20	25	15	10	610
3	c	3250	3550	-300	110	10	100	90	50	40	30	10	20	-140
4	d	5726	5449	276	17	26	-10	0	46	-46	0	0	0	221

The latter table corresponds exactly to Table 2 of Scholtus (2008).

5 Final remarks

This paper demonstrates our implementation of three data correction methods, initially devised by one of us (Scholtus (2008, 2009)). With the `deducorrect` R package, users can correct numerical data records which violate linear equality restrictions for rounding errors, typographical errors and sign errors and/or value transpositions. Since both the algorithms correcting for typographical and sign errors can take rounding errors into account, a typical data-cleaning sequence would be to start with correcting for sign- and typographical errors, ignoring rounding errors and subsequently treating the rounding errors. We note that data cleaning can be sped up significantly if independent blocks of editrules are treated separately. If an matrix representation of a set of edits can be written as a direct sum $\mathbf{A} = \mathbf{A}_1 \oplus \mathbf{A}_2$, data can be treated for editrules in $\mathbf{A}_1$ and $\mathbf{A}_2$ independently. The `editrules` package offers functionality to split editmatrices into blocks via the `findblocks` function.

References

Damerau, F. (1964). A technique for computer detection and correction of spelling errors. *Communications of the ACM 7*, 171–176.

de Jonge, E. and M. van der Loo (2011). *editrules: R package for parsing and manipulating edit rules.* R package version 0.4-1.

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

Heller, I. and C. Tompkins (1956). An extension of a theorem of Dantzig's. In H. Kuhn and A. Tucker (Eds.), *Linear inequalities and related systems*, pp. 247–254. Princeton University Press.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, 707–710.

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.

Raghavachari, M. (1976). A constructive method to recognize the total unimodularity of a matrix. *Zeitschrift für Operations Research 20*, 59–61.

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). Correction of rounding, typing and sign errors with the `deducorrect` package. Technical Report 11XXX (in press), Statistics Netherlands, Den Haag. This paper will become available via the website of Statistics Netherlands.

A Some notes on the editrules package

The `editrules` package (de Jonge and van der Loo, 2011) is a package for reading, parsing and manipulating numerical and categorical editrules. It offers functionality to conveniently construct edit matrices from verbose edit rules, stated as R statements. As an example consider the following set of edits on records with profit p, cost c, and turnover t.

$$\left\{\begin{array}{lcl} t & \geq & 1 \\ c & \geq & 0 \\ t & = & p+l \\ p & < & 0.6t. \end{array}\right. \tag{15}$$

The first two rules indicate that cost must be nonnegative, and turnover must larger than or equal to 1. The third rule indicates that the profit-loss account must balance, and the last rule indicates that profit cannot be more thatn 60% of the turnover. Denoting a record as a vector (p, l, t), these rules can be denoted as matrix equations:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} p \\ l \\ t \end{bmatrix} \geq \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{16}$$

$$\begin{bmatrix} 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} p \\ l \\ t \end{bmatrix} = 0 \tag{17}$$

$$\begin{bmatrix} 1 & 0 & -0.6 \end{bmatrix} \begin{bmatrix} p \\ l \\ t \end{bmatrix} < 0 \tag{18}$$

In the `editrules` package, these linear rules are all stored in a single object, called an `editmatrix`. It can be constructed as follows:

```
> (E <- editmatrix(c(
+     "t >= 1",
+     "l >= 0",
+     "t == p + l",
+     "p < 0.6*t")))

Edit matrix:
      t  l  p Ops CONSTANT
e1 -1.0  0  0  <=       -1
e2  0.0 -1  0  <=        0
e3  1.0 -1 -1  ==        0
e4 -0.6  0  1   <        0

Edit rules:
e1 : 1 <= t
```

```
e2 : 0 <= l
e3 : t == l + p
e4 : p < 0.6*t
```

An `editmatrix` object stores a stacked matrix representation of linear edit resrictions. Alternatively, one can define edits as a matrix and cast it into an `editmatrix` object:

```
> E <- matrix(c(
+      1,   0, 0,
+      0,   1, 0,
+      1,  -1,-1,
+     -0.6, 1, 1),
+     nrow=4,
+     byrow=TRUE,
+     dimnames=list(
+         1:4,
+         c("t","l","p")
+     )
+ )
> b <- c(1,0,0,0)
> ops <- c(">=", ">=", "==", ">")
> (E <- as.editmatrix(E,b,ops))

Edit matrix:
     t  l  p Ops CONSTANT
1  1.0  0  0  >=        1
2  0.0  1  0  >=        0
3  1.0 -1 -1  ==        0
4 -0.6  1  1   >        0

Edit rules:
1 : t >= 1
2 : l >= 0
3 : t == l + p
4 : l + p > 0.6*t
```

There are more storage modes in `editrules` which we will not detail here. Users can extract (in)equalities through the `getOps` function which returns a vector of comparison operators for every row. For example:

```
> E[getOps(E)==">=", ]

Edit matrix:
  t l p Ops CONSTANT
1 1 0 0  >=        1
2 0 1 0  >=        0

Edit rules:
1 : t >= 1
2 : l >= 0
```

Alternatively, the comparison operators of an edit matrix may be normalized:

```
> editmatrix(editrules(E), normalize = TRUE)

Edit matrix:
     t  l  p Ops CONSTANT
1 -1.0  0  0  <=       -1
2  0.0 -1  0  <=        0
3  1.0 -1 -1  ==        0
4  0.6 -1 -1   <        0

Edit rules:
1 : 1 <= t
2 : 0 <= l
3 : t == l + p
4 : 0.6*t < l + p
```

The `editrules` package offers functionality to check data against any set of editrules. The function `violatedEdits`, for example returns a boolean matrix indicating which record violates what editrules. `editrules` also offers editrule manipulation functionality, for example to split editmatrices into independent blocks. For further functionality of the `editrules` package, refer to the package documentation.