The rspa package for minimal record adjustment

Package version 0.1.8

Mark van der Loo

July 13, 2015

Abstract

The R extension package rspa offers functionality to minimally adjust a vector $\boldsymbol{x}$ such that it obeys the system of (in)equations $\boldsymbol{Ax} \leq \boldsymbol{b}$. The package implements the successive projection algorithm that was recently described by Pannekoek and Zhang [2012]. There are several ways to define what "minimal" means here, and the package works for fairly large systems of equations. Thus far it has been tested on systems with on the order of $10^5 - 10^6$ variables and $10^4 - 10^5$ restrictions where convergence is reached in seconds.

Contents

1 Introduction

In statistics one is often confronted with records or sets of estimated values that have to obey a set of linear equations and/or inequations. Examples include records from business surveys that have to obey accountancy rules or production-consumption balances for national account systems. In practice, such data seldom obeys all restrictions leading to further inconsistencies when the data is used as input for further analyses. One solution is to minimally adjust the data so that all restrictions are obeyed.

Here, with minimal adjustment we mean that a record $\boldsymbol{x}^0 \in \mathbb{R}^n$ is replaced with a value $\boldsymbol{x}$ such that the objective function

$$d(\boldsymbol{x}, \boldsymbol{x}^0) = \left[(\boldsymbol{x} - \boldsymbol{x}^0)'\boldsymbol{W}(\boldsymbol{x} - \boldsymbol{x}^0)\right]^{1/2}, \tag{1}$$

is minimized as a function of $\boldsymbol{x}$, subject to

$$\boldsymbol{Ax} \leq \boldsymbol{b}. \tag{2}$$

Here, $\boldsymbol{W}$ is a diagonal positive weight matrix.

Note that this problem has a simple geometric interpretation. The system $\boldsymbol{Ax} \leq \boldsymbol{b}$ describes a convex region of $\mathbb{R}^n$ (possibly of lower dimension than n). The problem is to replace the vector $\boldsymbol{x}^0$ outside this region with a record $\boldsymbol{x}$ lying in it while keeping the distance $d(\boldsymbol{x}, \boldsymbol{x}^0)$ as small as possible.

The algorithm implemented in the `rspa` package is a special case of the `successive projection algorithm` for more general convex optimization problems. The successive projection algorithm and the spacial case considered here was recently applied by Pannekoek and Zhang [2012] to adjustment problems related to survey data. Their paper also contains a detailed discussion of the algorithm.

The algorithm minimizes $d(\boldsymbol{x}, \boldsymbol{x}^0)$ as a function of $\boldsymbol{x}$ so that the restrictions are obeyed up to a certain *accuracy*. This accuracy is defined as the maximum absolute deviance from Eq. (2) and will be defined more precisely in Section 5.

This paper is intended as an overview of the package. For a detailed description of functions and their parameters, refer to the manual (help pages) that included with the package. The rest of this paper is structured as follows. In Sections 2 to 4 we demonstrate some of the package's core functionality. Sections 5 and 6 provide some details on the inner workings of the package which may help users to interpret the results.

2 A simple example

The following example is borrowed from Pannekoek and Zhang [2012] and involves profit-loss account balances from a business survey. The problem described in the reference involves a record of eight variables $x_1 \ldots x_8$ that have to obey the following rules.

$$\begin{aligned} x_5 &= x_1 + x_8 \\ x_5 &= x_3 + x_4 \\ x_8 &= x_6 + x_7 \\ x_4 &\geq 0. \end{aligned}$$

The first task is to define these constraints in `R`. The `rspa` package offers several ways to do this. One option, which we will use in this problem is to make use of the `editrules` package [De Jonge and Van der Loo, 2011-2012].

```
E <- editmatrix(expression(
    x5 == x1 + x8,
    x5 == x3 + x4,
    x8 == x6 + x7,
    x4 > 0))
```

The record in the example has the following values

```
x <- c(
   x1=330,
   x2=20,
   x3=1000,
   x4=30,
   x5=950,
   x6=500,
   x7=200,
   x8=700)
```

To confirm that this vector does not meet the constraints, we call `violatedEdits` (also part of `editrules`).

```
violatedEdits(E,x,tol=1e-2)


##       edit
## record num1 num2  num3  num4
##   [1,] TRUE TRUE FALSE FALSE
```

This shows that x violates the first two rules (indicated with `TRUE`), at least to within a tolerance of 0.01.

In the example of Pannekoek and Zhang [2012], the value of x_5 is considered correct. We can therefore substitute it in the set of constraints (`substValue`) and remove the corresponding variable from the set of constraints (`reduce`).

```
E <- reduce(substValue(E,'x5',x['x5']))
```

Adjusting $\boldsymbol{x}$ with the `adjust` function from `rspa` so it meets the constraints can be done as follows.

```
(y <- adjust(E, x))

## Object of class 'adjusted'
##   Status     : success (using 'dense' method)
##   Accuracy   : 0.00488281
##   Objective  : 85.0853
##   Iterations: 12
##   Timing (s): 0
## Solution:
##       x1       x2       x3       x4       x5       x6       x7       x8
## 282.0000  20.0000 950.0049   0.0000 950.0000 484.0000 184.0000 668.0000
```

The result is an object of class `adjusted`, which holds the found solution and some convergence information on the algorithm. The solution can be accessed as `y$x`. Using `violatedEdits` again, we see that now all restrictions are obeyed.

```
violatedEdits(E, y$x, tol = 1e-2)

##       edit
## record  num1  num2  num3  num4
##   [1,] FALSE FALSE FALSE FALSE
```

The output shows that the solution obeys the restrictions to within 10^{-2}. We will focus on the convergence criterion in the next section but before that, note the following properties of the `adjust` function.

- It only adjusts variables in $\boldsymbol{x}$ that occur in at least one of the restrictions.

- If the first argument of `adjust` is an `editmatrix` and the variables in $\boldsymbol{x}$ are named, they will automatically be matched so the order of variables is unimportant.

The `adjust` function is a generic function and it accepts constraints in `matrix`, `sparseConstraints` or `editmatrix` format. For small adjustment problems, up to say a few hundred variables and constraints, the `matrix` or `editmatrix` format will be fine. Large problems, with thousands (or millions) of variables and restrictions can be defined in sparse format and in that case the adjustment problem is solved with a routine for sparse adjustment. This is explained further in section 4.

3 Treating many records

To facilitate production-wise processing of many records, the function `adjustRecords` can adjust all records in a `data.frame` to meet the same set of rules. The output contains the adjusted records as well as logging information.

As an example, we create a new set of rules and generate some random data.

```
F <- editmatrix(expression(
   x + y == z,
   x >= 0,
   y >= 0,
   z >=0
))
N <- 100
dat <- data.frame(
   x = runif(100),
   y = rnorm(100),
   z = rlnorm(100)
)
```

By construction, it is very unlikely that all generated data obey the rules in `F`. To adjust the data, a single call to `adjustRecords` is sufficient.

```
A <- adjustRecords(F,dat)
summary(A)

## Object of class 'adjustedRecords'
##  Records : 100
##  Adjusted: 100 (100 converged)
##  duration: 0s (total)
## Summary of adjusted records:
##    objective           accuracy
##  Min.   :0.007658   Min.   :0.0000000
##  1st Qu.:0.412199   1st Qu.:0.0000000
##  Median :0.736963   Median :0.0000000
##  Mean   :1.127454   Mean   :0.0009061
##  3rd Qu.:1.267093   3rd Qu.:0.0018705
##  Max.   :8.293410   Max.   :0.0033071
```

By default, all variables in dat are adjusted to meet the rules in F. However, one can optionally pass an array indicating which variables to adjust. It is also possible to pass (an array or vector of) weights to control the relative amount of change per variable. The return value of adjustRecords is an object of class adjustedValues. It contains the adjusted records in A$adjusted and a data.frame collecting status information in A$status.

The R generic summary and plot functions have been overloaded to get a quick glance of result quality and amount of change from the original data. Figure 1 shows the result of plotting A of the above example. Two plots are created. The top panel shows a kernel density estimate of the accuracy values for each record. The lower panel shows a kernel density estimate of the objective function value. The actual values are shown as a "rug plot" under the density plots. In this very simple example, the accuracy is exactly zero for half of the treated records, indicating that the constraints are met exactly after adjusting.

It is well-known that kernel density estimates can extrapolate into regions where the actual probability density equals zero. Here, we need to make sure that the estimated probability density for accuracy or objective function equals zero for values < 0. This is achieved by estimating the accuracy density under a square root transform, and transforming back for graphical representation. For the objective function a log-transform is applied.

4 Treating large problems

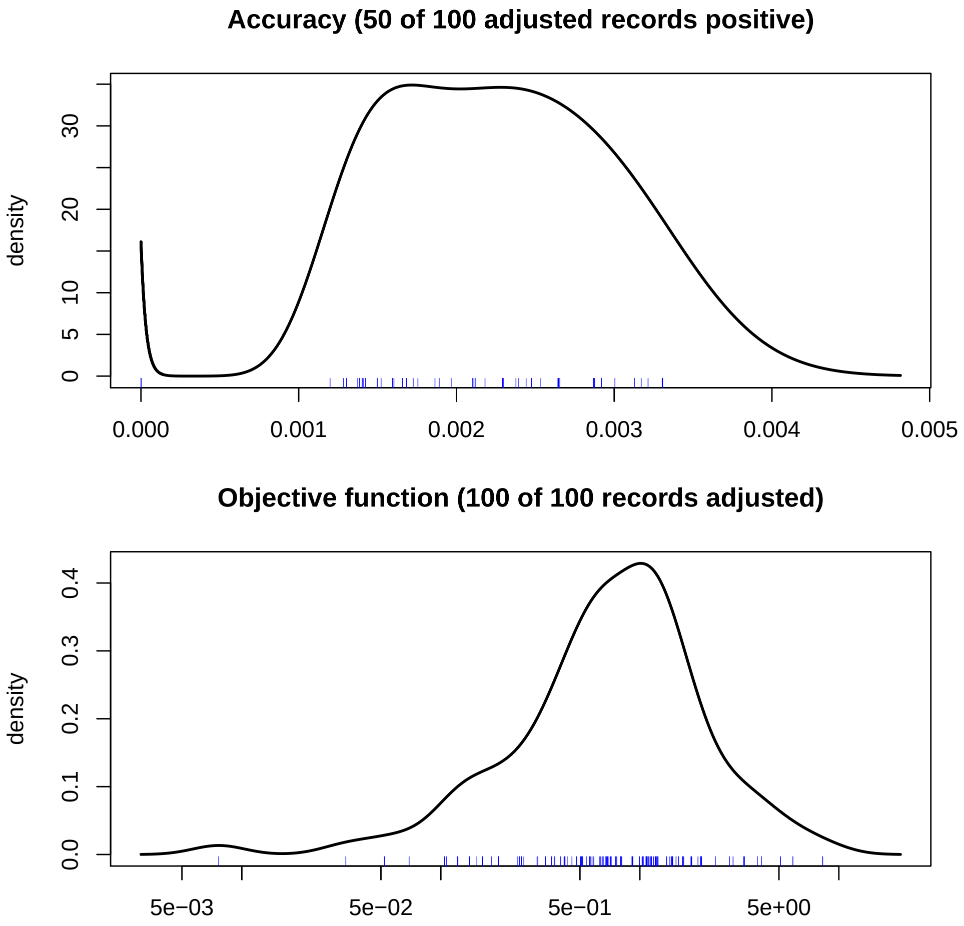
For problems where $\boldsymbol{x}$ has many coefficients and when there are many restrictions, the package includes an adjustment algorithm based on a sparse representation of the restrictions. In a sparse representation, elements of the restriction matrix $\boldsymbol{A}$ that are zero are not stored in computer memory.

Such a sparse representation is held in a sparseConstraints object. The function sparseConstraints constructs such objects and accepts arguments in the form of either

- An editmatrix
- A matrix $\boldsymbol{A}$, a constant vector $\boldsymbol{b}$ and an integer $n_=$ to indicate that the first $n_=$ rows of $\boldsymbol{A}$ and $\boldsymbol{b}$ represent equalities.
- A data.frame holding row indices, column indices and non-zero coefficients of $\boldsymbol{A}$ in its three columns, the vector $\boldsymbol{b}$ and $n_=$.

For large problems, the data.frame method is probably the most convenient so this will be demonstrated below.

```
plot(A)
```



Accuracy (50 of 100 adjusted records positive)

Objective function (100 of 100 records adjusted)

Figure 1: Plotting an object of class `adjustedRecords` yields a density plot of the accuracy (maximum absolute deviance from constraints) and objective function value per record.

Consider again the constraints in `E` of page 4. After substituting $x_5 = 950$, it reads:

```
E


## Edit matrix:
##      x1 x8 x3 x4 x6 x7 Ops CONSTANT
## num1 -1 -1  0  0  0  0  ==     -950
## num2  0  0 -1 -1  0  0  ==     -950
## num3  0  1  0  0 -1 -1  ==        0
## num4  0  0  0 -1  0  0   <        0
##
## Edit rules:
## num1 : 950 == x1 + x8
## num2 : 950 == x3 + x4
## num3 : x8 == x6 + x7
## num4 : 0 < x4
```

To create an object of class `sparseConstraints` we generate a `data.frame` called `rc` and a vector `b`:

```
rc <- data.frame(
   row = c( 1, 1, 2, 2, 3, 3, 3, 4),
   col = c( 1, 2, 3, 4, 2, 5, 6, 4),
  coef = c(-1,-1,-1,-1, 1,-1,-1,-1)
)
b <- c(-950, -950, 0,0)
```

Compare these indices and coefficients with the `editmatrix` representation above. With the `sparseConstraints` function a sparse representation is generated.

```
e <- sparseConstraints(rc, b, neq=3, sorted=TRUE)
e


## Sparse numerical constraints.
##   Variables   : 6
##   Restrictions: 4 (printing 4)
##   1 : -1*X0 + -1*X1 = -950
##   2 : -1*X2 + -1*X3 = -950
##   3 : 1*X1 + -1*X4 + -1*X5 = 0
##   4 : -1*X3 < 0
```

By passing the argument `sorted=TRUE`, we tell `sparseConstraints` that the input `data.frame` is sorted increasingly by column number (so it does not have to sort it again). The function detected here that row- and column indices are "base 1" (the lowest value equals 1). It is also possible to pass coefficient definitions which are base 0. Note that we did not feed `sparseConstraints` any names, so it makes up some names to represent the rules in textual form.

The `sparseConstraints` object is a *reference object* that holds a pointer to an object outside of `R`'s memory. Therefore, objects of class `sparseConstraints`

- cannot be copied. Copying generates a pointer to the same object.
- cannot be saved. Only the pointer to the external object will be stored. The external object is destroyed by `R`'s garbage collector when `R` closes, or when the `sparseConstraints` object is deleted or overwritten.

In a future version we might add a export option so that such objects can be saved as a fixed-width file, for example.

Next, we define a new vector that matches these constraints and adjust it.

```
x_sparse <- c(330, 700, 1000, 30, 500, 200)
(adjust(e, x_sparse))


## Object of class 'adjusted'
##   Status     : success (using 'sparse' method)
##   Accuracy  : 0.00488281
##   Objective : 85.0853
##   Iterations: 12
##   Timing (s): 0.001
## Solution:
## [1] 282.0000 668.0000 950.0049   0.0000 484.0000 184.0000
```

Which gives the expected results.

Finally, we show in Figure 2 a transcript of a case where we treated an adjustment problem of 474 948 variables under 60 675 constraints. We use the `LaF` package of Van der Laan [2012] to read the matrix $\boldsymbol{A}$ in row-column-coefficient format from a fixed-width file. Next, the constant vector `b` and `x` are read using `R`'s default `I/O` funcitons. The `sparseConstraints` object is generated and stored in `e`. A printout shows the number of variables and constraints. Because of the large number of variables no rules are printed explicitly. A call to `adjust` adjusts the vector minimally and returns an `adjusted` object. The solution was found in 5.66 seconds on an `Intel`$^{\circledR}$ `Core`$^{\mathrm{TM}}$ `i7-2677M` CPU running at 1.80GHz.

```
library(LaF)
## Loading required package: Rcpp
library(rspa)
## Loading required package: editrules

# read A-matrix
laf <- laf_open_fwf(
    file = "prob2A.txt",
    column_types = c("integer","integer","double"),
    column_widths = c(10,10,4)
    )
rowcol <- laf[]
laf <- close(laf)

# read b-vector
b <- read.csv("prob2b.txt",header=FALSE)[,1]

# read x-vector
x <- read.csv("prob2x.txt",header=FALSE)[,1]

e <- sparseConstraints(rowcol,b,neq=length(b))
e
## Sparse numerical constraints.
##  Variables   : 474948
##  Restrictions: 60675 (printing 0)

y <- adjust(e, x)
y
## Object of class 'adjusted'
##  Status    : success (using 'sparse' method)
##  Accuracy  : 0.00770226
##  Objective : 47430.5
##  Iterations: 552
##  Timing (s): 5.661
## Solution (truncated at 10):
## [1]  5.4028322  3.5246546  3.7979088  1.1202164  2.5304367  0.2037056
## [7] 97.9161357  1.5714899  4.5743395 -1.1756605
```

Figure 2: Transcript of treating an adjustment problem with 474 948 variables under 60 675 equality restrictions.

5 About the adjustment algorithm and convergence

The `rspa` package implements the *successive projection algorithm* as described by Pannekoek and Zhang [2012]. Given a vector $\boldsymbol{x}^0$ for which $\boldsymbol{A}\boldsymbol{x}^0 \not\leq \boldsymbol{b}$. The algorithm solves the following minimization problem

$$\begin{aligned} & \arg\min_{\boldsymbol{x}} (\boldsymbol{x}-\boldsymbol{x}^0)'\boldsymbol{W}(\boldsymbol{x}-\boldsymbol{x}^0) \\ s.t. & \\ \boldsymbol{A}\boldsymbol{x} &\leq \boldsymbol{b}, \end{aligned} \tag{3}$$

where $\boldsymbol{W}$ is a diagonal weight matrix with all weights positive. By default, all weights are chosen equal to 1 in the package. In words, the algorithm finds the vector with the smallest (weighted) Euclidean distance from the starting vector $\boldsymbol{x}^0$ that obeys the restrictions.

To define the convergence criterion, we separate the equality from inequality restrictions and write

$$\boldsymbol{A}_{=}\boldsymbol{x} = \boldsymbol{b}_{=} \tag{4}$$

$$\boldsymbol{A}_{\leq}\boldsymbol{x} \leq \boldsymbol{b}_{\leq}. \tag{5}$$

We now define $\varepsilon_{=}$ as the maximum difference between the left- and right hand side of (4) (the infinity norm, also know as L_∞ or the Chebyshev distance):

$$\varepsilon_{=} = \|\boldsymbol{A}_{=}\boldsymbol{x} - \boldsymbol{b}_{=}\|_\infty . \tag{6}$$

We also introduce the notation $\boldsymbol{d}_{\leq} = \boldsymbol{A}_{\leq}\boldsymbol{x} - \boldsymbol{b}_{\leq}$ and define

$$\varepsilon_{\leq} = \left\|\tfrac{1}{2}\left(|\boldsymbol{d}_{\leq}| + \boldsymbol{d}_{\leq}\right)\right\|_\infty . \tag{7}$$

This formulation ensures that $\varepsilon_{\leq} > 0$ only when at least one inequality is not obeyed by $\boldsymbol{x}$. The algorithm works by iteratively improving $\boldsymbol{x}^0$ until the convergence parameter $\max(\varepsilon_{=}, \varepsilon_{\leq}) < \varepsilon$. In other words: the algorithm terminates when the largest deviation from any of the (in)equality restrictions is met within a small parameter ε.

In the case that the set of user-defined constraints are infeasible (contradictory), the algorithm either diverges, resulting in `NaN`-coefficients, or the algorithm oscillates without convergence until the maximum number of iterations have been performed. Both cases are detected and reported. For sets of constraints that are stored as an `editmatrix` object, the function `isFeasible` of the `editrules` packages is able to check whether the set of constraints are contradictory.

Below is an example where the algorithm starts oscillating. It adjusts `x` for the first constraint, violating the second, adjusts it for the second constraint, violating the first, and so on.

```
e <- editmatrix(expression( x < 0, x > 1))
isFeasible(e)

## [1] FALSE

adjust(e,c(x=0.5))

## Object of class 'adjusted'
##   Status     : maximum number of iterations reached (using 'dense' method)
##   Accuracy  : 1
##   Objective : 0.5
##   Iterations: 1000
##   Timing (s): 0
## Solution:
## x
## 1
```

6 Some notes on implementation

The core algorithms are implemented as `C` routines (following the `C99` standard) that can be called from the `R` environment. The successive projection algorithm is implemented in a dense and a sparse version. The dense version is called by `adjust.matrix` and is also the default method that is called by `adjust.editmatrix`. If the optional argument `method='sparse'` is passed to `adjust.editmatrix`, the `editmatrix` object will be coerced to a `sparseConstraints` object prior to adjusting.

The `sparseConstraints` object is represented under the hood as a `C` `struct` that resides outside of `R`'s memory. It is an `R_ExternalPtr` object, packed in an `R` `environment` which is put in a `S3` class. Since the object is not in `R`'s memory, there is no point in trying to save a `sparseConstraints` object: only the pointer value will be stored while the external structure will be destroyed when `R` closes.

7 Conclusion

With the `R` extension package `rspa`, we have made the successive projection algorithm available for `R` users. In this paper we demonstrated how (lots of) small adjustments problems can conveniently be solved using a `editmatrix` definition of rules while large problems can be solved using a sparse representation of the problem.

Future work may include extending the package to allow for the weight matrix $\boldsymbol{W}$ to be non-diagonal and the possibility to read large problems from multiple formats.

8 Acknowledgements

I am greatly indebted by Guido van den Heuvel who critically reviewed the `C` code and pointed out many of the finer details of the `C99` standard to me. Any remaining bugs are of course of my doing.

References

J. Pannekoek and L.-C. Zhang. Optimal adjustments for inconsistency in imputed data. Technical report, Statistics Netherlands, 2012. In press.

E de Jonge and M van der Loo. *editrules: R package for parsing, applying and manipulating edit rules and error localization.*, 2011-2012. URL `http://cran.r-project.org/package=editrules`. R package version 2.5-0.

D.J. van der Laan. *LaF: methods for fast access to large ASCII files*, 2012. URL `http://cran.r-project.org/package=LaF`. R package version 0.4.