Exceedingly Simple Monotone Regression

Jan de Leeuw

Version 02, march 30, 2017

Abstract

A C implementation of Kruskal's up-and-down-blocks monotone regression algorithm for use wth .C(), and a comparison with other implementations.

Contents

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory deleeuwpdx.net/pubfolders/jbkPava has a pdf version, the bib file, the complete Rmd file with the code chunks, and the R source code.

1 Introduction

O no ! Not another monotone regression implementation !! There are already so many !!!

There is `isoreg()` in the `stats` package (R Development Core Team (2017)), `gpava()` in `isotone()` (De Leeuw, Hornik, and Mair (2009)), and `pava` in `Iso` (Turner (2015)). There is also `wmonreg()` in `smacof` (De Leeuw and Mair (2009), Mair, De Leeuw, and Groenen (2015)) and `amalgm()` from De Leeuw (2016). Now `gpava()` is written in R, `amalgm()` calls the Fortran from Cran (1980), `pava()` from `Iso` calls ratfor Fortran, and `isoreg()` only does unweighted least squares. I wanted something in C (because C is not Fortran) which did weighted monotone regression, and I wanted to use the .C() interface (because I am exceedingly simple).

Now `wmonreg` from `smacof` fits the bill. It was written in 2014 by Patrick Groenen and Gertjan van den Burg. Same as our proposed algorithm here, it implements the up-and-down-blocks algorithm of Kruskal (1964). If we compare computation time then `wmonreg()` is the main competitor. But I also wanted code that was easy to modify for different unimodal loss functions and that performed relatively uniformly over the range of "almost in the correct order" to "order completely wrong".

2 Algorithm

The best possible description of the algorithm was already given by Kruskal (1964) (page 127). We copy his recipe, with a slight change of notation and terminology.

"Our algorithm starts with the finest possible partitions into blocks, and joins the blocks together step by step until the correct partition is found. The finest possible partition consists naturally of n blocks, each containing only a single x_i.

Suppose we have any partition into consecutive blocks. We shall use $\overline{x}_b$ to denote the average of the x_i in block b. If b_-, b, b_+ are three adjacent blocks in ascending order, then we call b up-satisfied if $\overline{x}_b < \overline{x}_{b_+}$ and down- satisfied if $\overline{x}_{b_-} < \overline{x}_b$. We also call b up-satisfied if it is the highest block, and down-satisfied if it is the lowest block.

At each stage of the algorithm we have a partition into blocks. Furthermore, one of these blocks is active. The active block may be up-active or down-active. At the beginning, the lowest block, consisting of x_{min}, is up-active. The algorithm proceeds as follows. If the active block is up-active, check to see whether it is up-satisfied. If it is, the partition remains unchanged but the active block becomes down-active; if not, the active block is joined with the next higher block, thus changing the partition, and the new larger block becomes down-active. On the other hand, if the active block is down-active, do the same thing but upside-down. In other words, check to see whether the down-active block is down-satisfied. If it is, the partition remains unchanged but the active block becomes up-active; if not, the active block is joined with the next lower block into a new block which becomes up-active. Eventually this alternation between up-active and down- active results in an active block which is simultaneously up-satisfied and down-satisfied. When this happens, no

further joinings can occur by this procedure, and we transfer activity up to the next higher block, which becomes up-active. The alternation is again performed until a block results which is simultaneously up-satisfied and down-satisfied. Activity is then again transferred to the next higher block, and so forth until the highest block is up-satisfied and down-satisfied. Then the algorithm is finished and the correct partition has been obtained."

Our implementation `jbkPava()`, which uses Kruskal's initials to distinguish it from other pava's, stays as close as possible to this description. Blocks are C structures, collected in an array. Each block has a value, a weight, a size, the index of the previous block, and the index of the the next block.

```
struct block {
    double value;
    double weight;
    int size;
    int previous;
    int next;
};
```

At the end the block values are expanded and returned in the original vector. Space for the blocks is allocated and freed in the routine (i.e. it is not under the control of R). Given Kruskal's description of the up-and-down blocks algorithm, the code in the appendix should be pretty readable.

3 Timings

We apply the six monotone regression methods to vectors of length 10,000 with unit weights, repeating each analysis 100 times. We report the elapsed time from `system.time()`.

3.1 Random

The vectors of length 10,000 are normal random numbers (a different vector for each of the runs).

```
## wmonreg        0.294

## gpava        224.882

## isoreg         0.154

## amalgm         2.222
```

```
## Iso::pava     24.722

## jbkPava        0.183
```

Clearly `jbkPava()` and `isoreg()` are the clear winners, although `isoreg()` has the advantage that it does not have to take weighted means and keep track of the block weights. Note that `gpava()`, written in R, is abysmally slow. The two methods `amalgm()` and `Iso::pava()` that call Fortran routines are not really competitive. `wmonreg()` is doing quite well, but `jbkPava()` is more than twice as fast.

3.2 Reversed

We apply our six methods, one hundred times each, to the vector `10000:1`, which obviously will produce a completely merged vector with all elements equal to the mean.

```
## wmonreg        9.475

## gpava        532.118

## isoreg         0.053

## amalgm         2.754

## Iso::pava     34.659

## jbkPava        0.062
```

In this situtation, in which blocks are never up-satisfied and always down-satisfied, again `jbkPava()` and `isoreg()` are best. `wmonreg()` and `Iso::pava()` lose ground.

3.3 Ordered

We apply our six methods, one hundred times each, to the vector `1:10000`, which obviously just returns the vector itself. There is no merging at all, block are always up-satisfied and down-satisfied, and we merely loop through the vector.

```
## wmonreg        0.047

## gpava          7.765

## isoreg        39.892
```

```
## amalgm          0.113

## Iso::pava       0.039

## jbkPava         0.061
```

`wmonreg()` is best-in-show, which is interesting because in many applications we expect the vector to be closer to the correct ordering than to random or reverse ordering. Both Fortran calling routines perform well. `isoreg()` falls flat on its face and is even beaten by `gpava()`. There is probably some reasonable explanation for this, but since unweighted `isoreg()` is not really in the competition I did not explore this.

3.4 Conclusion

Both `pava()` and `wmonreg()` perform well over the range of examples. There is an indication that `pava()` is considerably better for vectors that are out of order, while `wmonreg()` may be better for vectors that are already close to correct. In future versions of this paper we will try to speed up `pava()` performance.

4 Code

4.1 R code

```
dyn.load("jbkPava.so")

jbkPava <- function (x, w = rep(1, length(x))) {
  h <-
    .C("jbkPava",
       x = as.double(x),
       w = as.double (w),
       n = as.integer(length(x)))
  return (h$x)
}
```

4.2 C code

```c
#include <stdlib.h>
#include <stdbool.h>

struct block {
    double value;
    double weight;
    int size;
    int previous;
    int next;
};

void jbkPava (double *x, double *w, const int *n) {
    struct block *blocks = calloc ((size_t) * n, sizeof(struct block));
    for (int i = 0; i < *n; i++) {
        blocks[i].value = x[i];
        blocks[i].weight = w[i];
        blocks[i].size = 1;
        blocks[i].previous = i - 1;
        blocks[i].next = i + 1;
    }
    int active = 0;
    do {
        bool upsatisfied = false;
        int next = blocks[active].next;
        if (next == *n) upsatisfied = true;
        else if (blocks[next].value > blocks[active].value)
            upsatisfied = true;
        if (!upsatisfied) {
            double ww = blocks[active].weight + blocks[next].weight;
            int nextnext = blocks[next].next;
            double wxactive = blocks[active].weight * blocks[active].value;
            double wxnext = blocks[next].weight * blocks[next].value;
            blocks[active].value = (wxactive + wxnext) / ww;
            blocks[active].weight = ww;
            blocks[active].size += blocks[next].size;
            blocks[active].next = nextnext;
            if (nextnext < *n)
                blocks[nextnext].previous = active;
            blocks[next].size = 0;
        }
        bool downsatisfied = false;
        int previous = blocks[active].previous;
        if (previous == -1) downsatisfied = true;
        else if (blocks[previous].value < blocks[active].value)
```

```
            downsatisfied = true;
        if (!downsatisfied) {
            double ww = blocks[active].weight + blocks[previous].weight;
            int previousprevious = blocks[previous].previous;
            double wxactive = blocks[active].weight * blocks[active].value;
            double wxprevious = blocks[previous].weight * blocks[previous].value;
            blocks[active].value = (wxactive + wxprevious) / ww;
            blocks[active].weight = ww;
            blocks[active].size += blocks[previous].size;
            blocks[active].previous = previousprevious;
            if (previousprevious > -1)
                blocks[previousprevious].next = active;
            blocks[previous].size = 0;
        }
        if ((blocks[active].next == *n) && downsatisfied) break;
        if (upsatisfied && downsatisfied) active = next;
    } while (true);
    int k = 0;
    for (int i = 0; i < *n; i++) {
        int blksize = blocks[i].size;
        if (blksize > 0.0) {
            for (int j = 0; j < blksize; j++) {
                x[k] = blocks[i].value;
                k++;
            }
        }
    }
    free (blocks);
}
```

References

Cran, G. W. 1980. "Algorithm AS 149: Amalgamation of Means in the Case of Simple Ordering." *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, 209–11.

De Leeuw, J. 2016. "Exceedingly Simple Isotone Regression with Ties." 2016.

De Leeuw, J., K. Hornik, and P. Mair. 2009. "Isotone Optimization in R: Pool-Adjacent-Violators Algorithm (PAVA) and Active Set Methods." *Journal of Statistical Software* 32 (5): 1–24.

De Leeuw, J., and P. Mair. 2009. "Multidimensional Scaling Using Majorization: SMACOF in R." *Journal of Statistical Software* 31 (3): 1–30.

Kruskal, J. B. 1964. "Nonmetric Multidimensional Scaling: a Numerical Method." *Psychometrika* 29: 115–29.

Mair, P., J. De Leeuw, and P. J. F. Groenen. 2015. "Multidimensional Scaling: SMACOF

in R."

R Development Core Team. 2017. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. %7Bhttp://www.R-project.org%7D.

Turner, R. 2015. *Iso: Functions to Perform Isotonic Regression*. %7Bhttps://CRAN.R-project.org/package=Iso%7D.