Isotonic Regression by Dynamic Programming

Günter Rote

November 22, 2012

1 Weighted Isotonic L_1 Regression

Weighted isotonic L_1 regression (or weighted isotonic median regression) is the following problem:

> Approximate a given sequence of numbers $a_1, \ldots, a_n$ with weights $w_i > 0$ by an increasing sequence $x_1 \le x_2 \le \cdots \le x_n$, minimizing the L_1-error
>
> $$\sum_{i=1}^{n} w_i \cdot |x_i - a_i|.$$

This problem can be solved in $O(n \log n)$ time [1, 5]. These algorithms will be reviewed in Section 9.

A conceptually easy dynamic programming approach leads to another algorithm with running time $O(n \log n)$. The resulting program is very simple and requires a priority queue as the only auxiliary data structure.

2 Dynamic Programming Setup

We consider the subproblems

$$f_k(z) := \min\Big\{ \sum_{i=1}^{k} w_i \cdot |x_i - a_i| : x_1 \le x_2 \le \cdots \le x_k = z \Big\} \tag{1}$$

for $k = 1, \ldots, n$ and a real parameter z. This leads in a straightforward way to the following dynamic programming recursion:

$$\begin{aligned} &f_k(z) := \min\{\, f_{k-1}(x) : x \le z \,\} + w_k \cdot |z - a_k| \quad (k = 1, \ldots, n;\ z \in \mathbb{R}) \\ &f_0(z) := 0 \qquad (z \in \mathbb{R}) \end{aligned} \tag{2}$$

The following properties of these functions will be easily established by induction.

Lemma 1. (a) *f_k is a piecewise linear convex function.*

(b) *The breakpoints are located at a subset of the points a_i.*

(c) *The leftmost piece has slope $-\sum_{i=1}^{k} w_i$. The rightmost piece has slope w_k.*

3 Piecewise Linear Functions

We can represent a continuous piecewise linear function $f(x)$ as a list of *breakpoints*, see Figure 1. Each breakpoint has a *position*: the x-value where it is located, and a *value*: the slope difference between the right and the left adjacent pieces. The breakpoints are naturally ordered by position, but for the time being, we leave it unspecified whether we want to store is as a sorted list or in some other data structure. The function is convex if all breakpoints have nonnegative values.

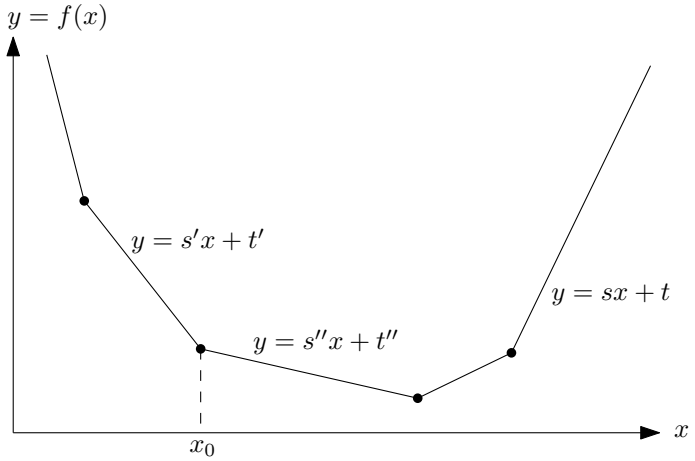

Figure 1: A piecewise linear function with four breakpoints. There is a breakpoint at position x_0 with value $s'' - s'$.

These breakpoint data determine the function f only up to addition of an arbitrary linear function. To determine f uniquely, we must add two further parameters. For example, we can take the slope s and the intercept t of the *rightmost* linear piece $y = sx + t$. We can then proceed from right to left, and across each breakpoint, the value of the breakpoint gives us the slope $\hat{s}$ of next linear piece $y = \hat{s}x + \hat{t}$, and continuity of f allows us to fix the intercept $\hat{t}$.

Two functions are *added* by combining the list of breakpoints and adding the (s, t) parameters. If several breakpoints have the same position, they might be merged into one breakpoint, adding their values. However, this is not necessary; the algorithm will work just as well with equal breakpoints.

4 Carrying out the Recursion (2)

We denote by

$$g_{k-1}(z) := \min\{\, f_{k-1}(x) : x \le z \,\}$$

the intermediate function in going from f_{k-1} to f_k.

Let us assume by induction that Lemma 1 holds for f_{k-1}. The function f_{k-1} is first monotonically decreasing to a minimum at some position p_{k-1} and then monotonically increasing; therefore the optimum of $f_{k-1}(x)$ under the constraint $x \le z$ depends on the position of z relative to p_{k-1}: If $z \le p_{k-1}$ then $x = z$ is the optimum choice, and $g_{k-1}(z) = f_{k-1}(x)$. If $z \ge p_{k-1}$ then the optimum choice is $x = p_{k-1}$, and $g_{k-1}(z) = f_{k-1}(p_{k-1})$, see Figure 2.

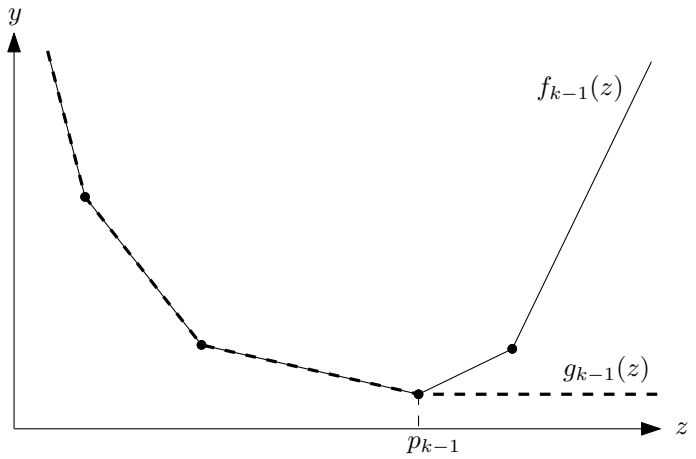

Figure 2: Constructing g_{k-1} from f_{k-1}

As a consequence of this, we get the following relation between the optimal values x^*_{k-1} and x^*_k in the optimal solution:

$$x^*_{k-1} := \min\{x^*_k, p_{k-1}\} \tag{3}$$

Summarizing, the function g_{k-1} has the same decreasing pieces as f_{k-1} but the increasing pieces are replaced by a horizontal piece of constant value $f_{k-1}(p_{k-1})$. This is formally described in Algorithm 1. In representing the functions, we only have to deal with the slope s of the rightmost piece; the intercept t is not needed. Since the leftmost slope is negative, by Lemma 1c, the **while**-loop will terminate, and the list of breakpoints will never become empty. If f_{k-1} has a horizontal piece, the algorithm will arbitrarily choose the leftmost minimum p_{k-1}.

Input: list of breakpoints of f_{k-1} and rightmost slope s
Result: updated list of breakpoints of g_{k-1} and rightmost slope s; position p_{k-1} of the (leftmost) minimum of f_{k-1}
Let B be the rightmost breakpoint;
while $s - B.value \geq 0$ **do** // next-to-last piece is not decreasing
  $s := s - B.value$;
  Delete B from the list of breakpoints;
  Let B be the rightmost remaining breakpoint;
$p_{k-1} := B.position$; // p_{k-1} is the position of the minimum of f_{k-1}.
$B.value := B.value - s$;
$s := 0$;

Algorithm 1: Converting f_{k-1} to g_{k-1}

Finally, to obtain f_k, we simply have to add the function $w_k \cdot |z - a_k|$ to g_{k-1}: add a breakpoint of value $2w_k$ at position a_k, and add w_k to s.

It is easy to see that Lemma 1 holds now for f_k. The base case ($k = 1$) is obvious, and the lemma is hence proved by induction.

5 The Weighted Regression Algorithm

We see that the algorithm only needs to access the rightmost breakpoint, and potentially delete it. A new breakpoint is inserted for each new data point a_k. This calls for a

(max-)priority queue for storing the breakpoints, using *position* as the key.

Algorithm 2 shows the complete algorithm that we can now put together. The main loop turns g_{k-1} into f_k and then into g_k. We start with the function $g_0(z) = 0$, and add $w_1 \cdot |z - a_1|$ to it to obtain f_1. The algorithm records the minimum position p_k for each function. In the last loop, the minimum of f_n is found as part of the construction of g_n.

Finally, the optimum solution (x_i) is computed in a simple loop according to (3), starting with the minimum $x_n^* = p_n$ of the function f_n.

```
Q := ∅; // priority queue of breakpoints ordered by the key position
s := 0;
for k = 1, ..., n do
    Q.add(new breakpoint B with B.position := a_k, B.value := 2w_k);
    s := s + w_k; // Now we have computed f_k.
    B := Q.findmax;
    while s − B.value ≥ 0 do
        s := s − B.value;
        Q.deletemax;
        B := Q.findmax;
    p_k := B.position;
    B.value := B.value − s;
    s := 0; // Now we have computed g_k.
// Now compute the optimal solution x_1, ..., x_n:
x_n := p_n;
for k = n − 1, n − 2, ..., 1 do
    x_k := min{x_{k+1}, p_k};
```

Algorithm 2: Weighted isotonic l_1 regression by dynamic programming

6 Runtime Analysis

In total, n elements are inserted in the queue Q. Each iteration of the **while**-loop removes an element from Q, and therefore the overall number of executions of the **while**-loop is bounded by n. With a heap data structure for Q, each operation *deletemax*, or *insert* can be carried out in $O(\log n)$ time. The *findmax* operation takes only constant time. Hence, the overall running time is $O(n \log n)$.

Theorem 1. *Algorithm 2 solves the weighted isotonic L_1 regression problem in $O(n \log n)$ time.* □

7 Unweighted Regression

In the unweighted case ($w_i \equiv 1$), some steps can be simplified: s is always 0 at the beginning of the **for**-loop and 1 before the **while**-loop. Thus, the variable s can be eliminated, and the **while**-loop can be turned into an **if**-statement. Breakpoints have weight 1 or 2.

8 Weighted Isotonic L_2 Regression

It is straightforward to extend the approach to the L_2-error

$$\sum_{i=1}^{n} w_i \cdot (x_i - a_i)^2.$$

Exercises. 1) Show that the functions f_k defined in analogy to (1) for the L_1 case are piecewise quadratic convex functions. Explore their further properties, in analogy to Lemma 1.
2) Design an appropriate efficient data structure for representing this class of functions.
3) Show that the dynamic programming recursions can be solved in $O(n)$ overall time, using only a stack as a data structure.
4) Compare the algorithm to the algorithm of Stout [5, Fig. 5 and Fig. 7] and find out whether the two algorithms carry out essentially the same calculations.

9 Other Algorithms

The most extensively studied approach for the isotonic regression problem is the algorithm *Pool Adjacent Violators* (PAV). This classical method, which has often been rediscovered, starts by combining adjacent values that are not monotone ($a_{i+1} < a_i$) into pairs, and it further combines groups into larger groups as long as the medians of adjacent groups are out of order.

Ahuja and Orlin [1] gave an $O(n \log n)$ algorithm that is based on the PAV principle but uses scaling for speedup. It refines estimates for the optimum x_i values in a binary-search-like manner. To get a running time that is independent of the range of values, the algorithm replaces the given values a_i by $1, \ldots, n$ while keeping their relative order fixed.

Stout [5] has given an efficient direct implementation of the PAV approach, using mergeable trees (for example, AVL trees or 2-3-trees, see [3]), achieving a running time of $O(n \log n)$.

Another approach, which I have not found in the literature, is to model the problem as a minimum-cost network flow problem: The unknowns x_i are flow values on a chain; the inequalities $x_i \leq x_{i+1}$ become conservation-of-flow relations, with an additional entering arc taking the slack. The resulting network is series-parallel, and hence can be solved in $O(n \log n)$ time, by an algorithm of Booth and Tarjan [2]. However, their algorithm also relies on mergeable trees, and moreover, it needs $O(n \log^* n)$ space to recover the optimum solution. So this approach is not preferable to Stout's algorithm. The algorithm follows the dynamic programming paradigm, and thus, in spirit, it is closer to the algorithm of this paper.

Ahuja and Orlin [1] wrongly attribute an earlier $O(n \log n)$ time algorithm to [4], but this paper has only an algorithm with $O(n \log^2 n)$ runtime.

Comparison. The algorithm of Stout [5] involves tree data structures (for example, AVL trees or 2-3-trees) augmented with weight information, and needs the nonstandard merging operation: two trees of size m and n with $m \leq n$ can be merged in $O(m \log \frac{n}{m}) = O(\log \binom{m+n}{n})$ time.

The scaling PAV algorithm of Ahuja and Orlin [1], on the other hand, requires no data structures beyond arrays and linked lists. The algorithm itself, however, is not so simple. Moreover, it requires an initial sort of the elements.

The dynamic programming algorithm is very simple and requires just a priority queue, in which each element is inserted once and retrieved at most once. Thus, the priority queue has to perform the same *insert* operations and fewer *deletemax* operations than would be required for heapsort; one might expect that the dynamic programming algorithm is finished while the scaling PAV algorithm is still busy in its sorting phase.

Incremental Computation (Prefix Regression). The dynamic programming algorithm processes data as they arrive, producing the solution for the first i items after reading them. (To have the of the objective function *value* always ready, the algorithm must be extended to deal with the intercept t in addition to the slope s.)

Stout [5] has called this problem the *prefix isotonic regression* problem: solving the regression problem for all prefixes of the input. His algorithm solves also this problem in $O(n \log n)$ time. He uses it as a subroutine for the *unimodal* regression problem.

The scaling PAV algorithm of Ahuja and Orlin is not suitable for incremental computation.

Data-Sensitivity. Another question is how the algorithm responds to input that is almost sorted. This is a natural assumption in the statistical applications, where the data "should" be monotone but is distorted by measurement errors.

The PAV algorithm will have an advantage, since values a_i that are in the correct order with respect to their neighbors and are approximated by themselves ($x_i = a_i$) will be looked at only once. Moreover, data will usually cluster in small groups, and in the $O(\log n)$ bound on the tree operations, the parameter n can be replaced by the size of groups.

The scaling PAV algorithm of Ahuja and Orlin [1], on the other hand, is completely insensitive to the data: in addition to sorting, it will always perform $O(\log n)$ linear sweeps over the data.

The dynamic programming algorithm might potentially benefit from almost sorted data. At least in the case when the input comes in truly sorted order, the algorithm will never call *deletemax*. To make use of this, one would need a priority queue where it is cheaper to retrieve (by *findmax*) and delete elements that have recently been inserted.

References

[1] R. K. Ahuja and J. B. Orlin, *A fast scaling algorithm for minimizing separable convex functions subject to chain constraints*, Operations Research **49** (2001), 784–789, doi:10.1287/opre.49.5.784.10601.

[2] Heather Booth and Robert Endre Tarjan, *Finding the minimum-cost maximum flow in a series-parallel network*, J. Algorithms **15** (1993), 416–446, doi:10.1006/jagm.1993.1048.

[3] M. R. Brown and R. E. Tarjan, *A fast merging algorithm*, J. Assoc. Comp. Mach. **26** (1979), 211–226, doi:10.1145/322123.322127.

[4] P. M. Pardalos, G. L. Xue, and Y. Li, *Efficient computation of an isotonic median regression*, Applied Mathematics Letters **8** (1995), no. 2, 67–70, doi:10.1016/0893-9659(95)00013-G.

[5] Quentin F. Stout, *Unimodal regression via prefix isotonic regression*, Comp. Stat. and Data Anal. **53** (2008), 289–297, doi:10.1016/j.csda.2008.08.005.