Super Smoothing with Xlisp-Stat

Jason Bond
Afshan Tabazadeh

December 19, 1994

1 Description

The super smoothing algorithm, originally implemented in FORTRAN by Jerome Friedman of Stanford University, is a method by which a smooth curve may be fitted to a two-dimensional array of points. Its implementation is presented here in the Xlisp-Stat language.

The smoothing function takes two one-dimensional sequences, $\underline{x}$ and $\underline{y}$ with weights $\underline{w}$, of the same length and returns a smoothed sequence $\underline{\hat{y}}_s$. Each smoothed estimate $\hat{y}_{s_i}$ is obtained by a linear prediction of the form:

$$\begin{aligned}
y_{s_i} &= \alpha_{\ell_i} + \gamma_{\ell_i} x_i + \epsilon_{\ell_i} \quad (i = 1, \ldots, n) && (1)\\
E(\epsilon_{\ell_i}) &= 0, \quad Var(\epsilon_{\ell_i}) = \sigma^2 && (2)\\
\underline{\beta}_{\ell_i} &= (\mathbf{X'}_{\ell_i}\mathbf{W}_{\ell_i}\mathbf{X}_{\ell_i})^{-1}\mathbf{X'}_{\ell_i}\mathbf{W}_{\ell_i}\underline{y}_{\ell_i} \;=\; (\alpha_{\ell_i}\ \gamma_{\ell_i})' && (3)\\
\underline{y}_{\ell_i} &= (y_{i-\frac{J}{2}}, \ldots, y_i, \ldots, y_{i+\frac{J}{2}})' && (4)\\
\mathbf{X}_{\ell_i} &= \begin{pmatrix} 1 & x_{i-\frac{J}{2}} \\ \vdots & \vdots \\ 1 & x_i \\ \vdots & \vdots \\ 1 & x_{i+\frac{J}{2}} \end{pmatrix} \quad \mathbf{W}_{\ell_i} = \begin{pmatrix} w_{i-\frac{J}{2}} & 0 & 0 & \ldots & 0 \\ 0 & w_{i-\frac{J}{2}+1} & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & w_{i+\frac{J}{2}} \end{pmatrix} && (5)
\end{aligned}$$

The estimate of each smoothed y_{s_i} is obtained from a simple linear model where only a symmetric neighborhood of J pairs $((x_{i-\frac{J}{2}}, y_{i-\frac{J}{2}}), \ldots, (x_{i+\frac{J}{2}}, y_{i+\frac{J}{2}}))$ are used in the estimation of $\underline{\beta}_{\ell_i}$. Here we assume J to be an even integer. Whenever a symmetric neighborhood of size J is not available, such as the case of $i < \frac{J}{2}$ or $i > n - \frac{J}{2}$, the nearest neighborhood of size J is used.

Changing the value of J also changes the degree of smoothing. Small changes in J, however, tend not to significantly alter the resulting curve estimate. It should be expected that small neighborhoods produce erratic curves whereas large values of J tends to linearize the curve estimate. The case of $J = n$ represents the linear regression of $\underline{y}$ on $\underline{x}$ using all of $\underline{x}$ to estimate $\underline{\beta}_{\ell_1} = \ldots = \underline{\beta}_{\ell_n}$. Although the smoothing may be done with the same fixed value J for each i, a varying neighborhood of size $J(x_i)$ around x_i produces smoother looking curves with less bias.

Choice of neighborhood size can either be given as a fixed value or a variable neighborhood may be chosen automatically for each x_i. For the variable span smoother, the choice of neighborhood size for x_i is determined by the minimum over $J = (.05n, .2n, .5n)$ of the smoothed (using $J = .2n$) absolute cross-validated residual:

$$\left|r_{(i)}(J)\right| = \left|\frac{[y_i - \hat{y}_{\ell_i}]}{\left(1 - \frac{1}{\sum_{\ell_i} w_i} - \frac{(x_i - \bar{x}_{\ell_i})^2}{Var(x_{\ell_i})}\right)}\right| \tag{6}$$

against x_i and is denoted $J(x_i)$. Therefore $J(x_i) \in (.05n, .2n, .5n)$. "Cross Validation" is performed by removing each observation in turn while smoothing the remaining $n-1$ observations. The span sizes ($.05n$, $.2n$, $.5n$) are referred to henceforth as the ($J_t = tweeter, J_m = midrange$, and $J_w = woofer$) span values.

If the underlying curve is known to be quite smooth, span estimates can be forced toward the *woofer* span through Bass Enhancement. The variable span $J_v(x_i)$ with Bass Enhancement $\alpha \in (0, \ldots, 10)$ is defined as follows.

$$J_v(x_i) = J(x_i) + (J_w - J(x_i))R_i^{10-\alpha} \tag{7}$$

$$R_i = \left[\frac{\hat{e}(\hat{y}, J(x_i) \mid x_i)}{\hat{e}(\hat{y}, J_w \mid x_i)}\right] \tag{8}$$

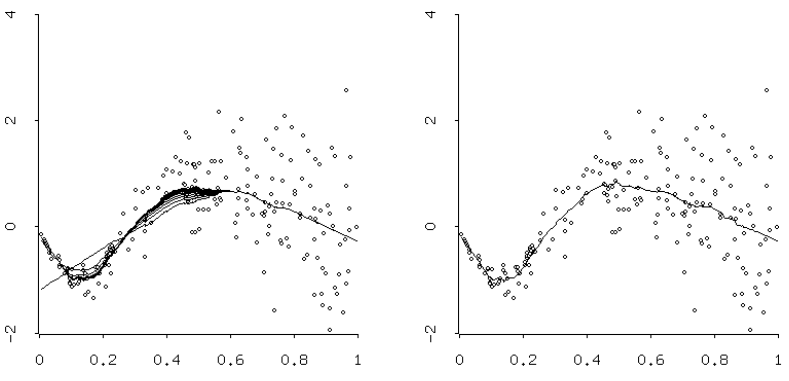
where $\hat{e}(\hat{y}, J_w \mid x_i)$ denotes the smoothed absolute cross-validated residuals for the *woofer* span. Here we define $\hat{e}(\hat{y}, J(x_i) \mid x_i)$ as $\underset{J \in (J_t, J_m, J_w)}{min}(\hat{e}(\hat{y}, J \mid x_i))$. The ratio $0 \le R_i \le 1$ measures the ratio of estimated error variance at x_i when using a neighborhood of size J_w to that of the $\underset{J_t, J_m, J_w}{min}$ estimated error variance at x_i. If a local increase in the ratio occurs then smoother estimates are obtained by increasing $J_v(x_i)$ without significant bias increases. If no Bass Enhancement is desired then $J_v(x_i) = J(x_i)$.

Since the resulting span estimates $J_v(x_i)$ are not constrained to vary smoothly from one observation to the next, they often do not as a result of the variance of the estimates $\hat{e}(\hat{y}, J \mid x_i)$. For this reason, we obtain more smoothly varying optimal span estimates $J_o(x_i)$ by smoothing $J_v(x_i)$ against x_i. Final smoothed values for each y_i are determined from the interpolation:

$$D = \left[\frac{J_o(x_i) - J_m}{J_w - J_m}, \frac{J_o(x_i) - J_m}{J_m - J_t}\right]^+ \tag{9}$$

$$\hat{y}_{s_i} = \begin{cases} (1-D)\hat{y}_{s_m} + D\hat{y}_{s_w} & \text{if } J_o(x_i) \ge J_m, \\ (1-D)\hat{y}_{s_m} + D\hat{y}_{s_t} & \text{if } J_o(x_i) < J_m. \end{cases} \tag{10}$$

Further discussion of the variable span smoother can be found in Friedman [1].



(a) $\alpha = (0, \dots, 10)$ (b) No Bass Enhancement

Figure 1: Variable Span Smooths

2 Example

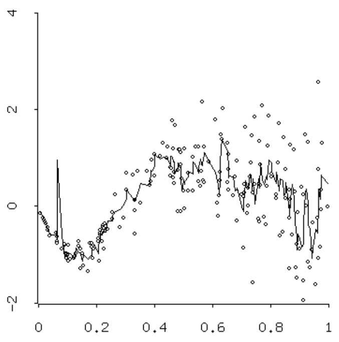
In this section the same example is considered as in Friedman [1]. The data is a simulation of $n = 200$ pairs $(x_i,\ y_i,\ w_i = 1)$ where:

$$\begin{aligned} y_i &= sin(2\pi(1 - x_i)^2) + x_i\epsilon_i && (11) \\ x_i &\overset{iid}{\sim} U[0, 1] && (12) \\ \epsilon_i &\overset{iid}{\sim} N(0, 1) && (13) \end{aligned}$$

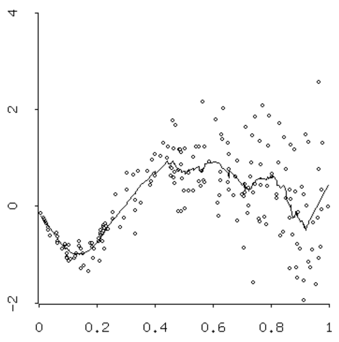
Figure 1(a) shows the variable span curve estimates using all base enhancement values $\alpha = (0, \dots, 10)$. Larger Bass Enhancement values tend to produce more bias but also tend to allow a smoother curve. It can be seen that the variable span smoother with $\alpha = 10$ is equivalent to the $woofer$ span smoother. Figure 1(b) displays the variable span smoother without bass enhancement. Figure 2 displays smoothed curves for different fixed span values.

As expected, smaller spans tend to produce more erratic curve behavior and larger spans push the curve toward strict linearity. The fixed span estimates with small neighborhoods show significantly less bias for smaller values of x but result in much more variability for medium and larger x values. The midrange smoother behaves well for a wide range of x but is biased for smaller x values. For larger spans, the smoothed curve is highly biased for small values of x but behaves quite well for x values closer to 1.

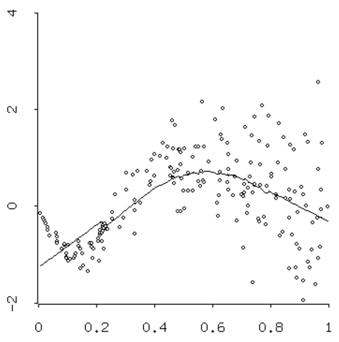
The fixed span local linear smoothers therefore cannot attain small variability and small bias simultaneously. The tweeter smoother performs best for small values of x, the midrange for medium values of x, and the woofer



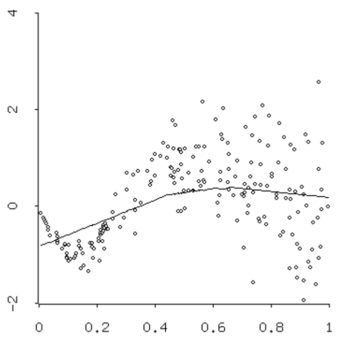
(a) $J = 5$



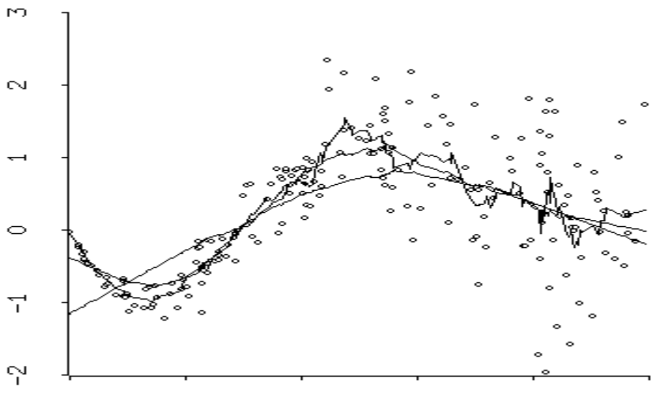
(b) $J = 20$



(c) $J = 80$



(d) $J = 150$



(e) $J = (.05n, .2n, .5n)$

Figure 2: Smooths with J=(5, 20, 80, 150, $(tweeter, midrange, woofer)$)

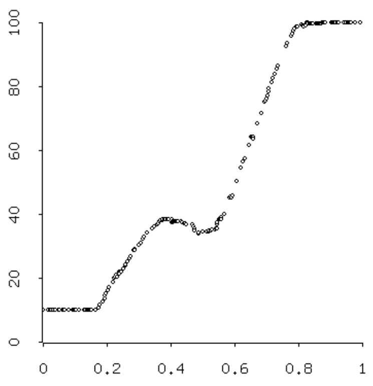


Figure 3: $J_o(x_i)$ vs. x_i

for larger x values. For optimal variable span selection, the spans should be chosen in such a way that the resulting curve behaves as well as any fixed span smoother in a given neighborhood. In our example, the variable span smoother should decrease neighborhood size for small values of x where $\left|\frac{\partial^2 f}{\partial x^2}\right| \gg 0$. Span sizes should increase with x since error variance also increases with x. This is, in fact, what can be seen in Figure 3 where the smoothed span estimates $J_o(x_i)$ are plotted against x_i for the variable span smoother.

References

[1] Friedman, Jerome. "A Variable Span Smoother". Technical Report No. 5, 1984. Stanford University, Ca.

3 Xlisp-Stat Super-smoother Code

```
(defproto supersmoother-proto '(x y w n r1 r2 r3 tsmo msmo wsmo jcv
                                alpha span smoothed-y resid))

(defmeth supersmoother-proto :isnew (x y w span alpha)
"Args: X Y W SPAN ALPHA
X is the independant variable.  Y is the sequence to be smoothed with
weights W.  Span should be either a fixed span width to be used for all
X's or NIL for a Variable Span smooth.  If SPAN is non-NIL then ALPHA
will be ignored.  Otherwise alpha is the Bass tone control parameter
in the Variable Span smoother."
   (let* (
         (rx (rank x))
         (n (length x))
         (newx (repeat 0 n))
         (newy (repeat 0 n))
         (neww (repeat 0 n))
        )
    (mapcar #'(lambda (i j) (setf (elt newx i) (elt x j))) rx (iseq n))
    (mapcar #'(lambda (i j) (setf (elt newy i) (elt y j))) rx (iseq n))
    (mapcar #'(lambda (i j) (setf (elt neww i) (elt w j))) rx (iseq n))
    (send self :x newx)
    (send self :y newy)
    (send self :w neww)
    (send self :alpha alpha)
    (send self :n n)
    (send self :span span)
    (send self :start)
   )
)


(defmeth supersmoother-proto :smooth (var2 j res)
  (let* (
         (x (send self :x))
         (var2 (cond ((string= var2 "y") (send self :y))
                     ((string= var2 "r1") (abs (send self :r1)))
                     ((string= var2 "r2") (abs (send self :r2)))
                     ((string= var2 "r3") (abs (send self :r3)))
                     ((string= var2 "J") (send self :jcv))))
         (w (send self :w))
         (n (send self :n))
         (bottom (floor (/ j 2)))
         (top (- n bottom 1))
        (xseq (select x (iseq 0 j)))
        (var2seq (select var2 (iseq 0 j)))
        (wseq (select w (iseq 0 j)))
        (wsum (sum wseq))
        (xbar (/ (sum (* xseq wseq)) wsum))
        (var2bar (/ (sum (* var2seq wseq)) wsum))
        (cov (sum (* wseq (- xseq xbar) (- var2seq var2bar))))
        (xvar (sum (* wseq (^ (- xseq xbar) 2))))
        (i 0)
        (smooth-vals (list ))
        (residuals (list ))
        (beta 0)
        (alpha 0)
        (smooth 0)
       )
    (loop
      (setf beta (/ cov xvar))
      (setf alpha (- var2bar (* beta xbar)))
      (setf smooth (+ alpha (* beta (elt x i))))
      (if res (progn
                (setf residuals (append residuals
                  (list (/ (- (elt var2 i) smooth)
                           (- 1 (/ wsum)
                                (/ (^ (- (elt x i) xbar) 2) xvar))))))))
      (setf smooth-vals (append smooth-vals (list smooth)))
      (setf i (1+ i))
      (if (= i n)
             (return (if res (list smooth-vals residuals) smooth-vals)))
      (cond ((and (< i top) (> i bottom))
            (let* (
                  (wtop (elt w (+ i bottom)))
                  (wbot (elt w (- i bottom 1)))
                  (xt (elt x (+ i bottom)))
                  (xb (elt x (- i bottom 1)))
                  (vt (elt var2 (+ i bottom)))
                  (vb (elt var2 (- i bottom 1)))
                  (xtop (* xt wtop))
                  (xbot (* xb wbot))
                  (v2top (* vt wtop))
                  (v2bot (* vb wbot))
                  (wsum (sum wseq))
                 )
              (setf xbar (/ (+ (* wsum xbar) xtop (- xbot))
                            (+ wsum wtop (- wbot))))
```

```
               (setf var2bar (/ (+ (* wsum var2bar) v2top (- v2bot))
                             (+ wsum wtop (- wbot))))

               (setf xbef (- xt (/ (+ (* xbar (+ wsum wtop (- wbot))) xbot)
                                   (+ wsum wtop))))

               (setf vbef (- vt (/ (+ (* var2bar (+ wsum wtop (- wbot))) v2bot)
                                   (+ wsum wtop))))

               (setf cov (+ cov (* (/ (+ wtop wsum) wsum) wtop xbef vbef)

                            (- (* (/ (+ wsum wtop (- wbot)) (+ wsum wtop))
                                   wbot (- xb xbar) (- vb var2bar)))))

               (setf xvar (+ xvar (* (/ (+ wtop wsum) wsum) (^ xbef 2) wtop)
                             (- (* (/ (+ wsum wtop (- wbot)) (+ wsum wtop))
                                   wbot (^ (- xb xbar) 2)))))
             )
            (setf xseq (select x (iseq (- i bottom) (+ i bottom))))
            (setf var2seq (select var2 (iseq (- i bottom) (+ i bottom))))
            (setf wseq (select w (iseq (- i bottom) (+ i bottom))))
           )
        )
     )
 )
)



(defmeth supersmoother-proto :start ()
(let (
      (span (send self :span))
     )
 (cond ((not span)
  (let* (
         (n (send self :n))
         (alpha (if (= (send self :alpha) 11) nil (send self :alpha)))
         (tweet (max 2 (floor (* .05 n))))
         (mid (max 2 (floor (* .2 n))))
         (woof (max 2 (floor (* .5 n))))
         (tweetsmo (send self :smooth "y" tweet t))
         (midsmo (send self :smooth "y" mid t))
         (woofsmo (send self :smooth "y" woof t))
         (temp (progn (send self :tsmo (first tweetsmo))
                      (send self :msmo (first midsmo))
                      (send self :wsmo (first woofsmo))
                      (send self :r1 (second tweetsmo))
                      (send self :r2 (second midsmo))
                      (send self :r3 (second woofsmo))))
             (tresidsmo (send self :smooth "r1" mid nil))
             (mresidsmo (send self :smooth "r2" mid nil))
             (wresidsmo (send self :smooth "r3" mid nil))
             (temp (progn (send self :r1 tresidsmo)
                          (send self :r2 mresidsmo)
                          (send self :r3 wresidsmo)))
             (bspanest (map-elements #'(lambda (x y z)
                                (cond ((= x (min x y z)) tweet)
                                      ((= y (min x y z)) mid)
                                      (t woof))) tresidsmo mresidsmo wresidsmo))
             (minresid (mapcar #'min tresidsmo mresidsmo wresidsmo))
             (jcv (if alpha (send self :jcv (map-elements #'(lambda (x y)
                                      (+ x (* (- woof x) (^ y (- 10 alpha)))))
                                    bspanest (/ minresid wresidsmo)))
                         (send self :jcv bspanest)))

             (jcvsmo (send self :smooth "J" mid nil))
            )
        (send self :resid minresid)
        (send self :jcv (mapcar #'(lambda (x) (cond ((> x woof) woof)
                                                    ((< x tweet) tweet)
                                                    (t x))) jcvsmo))
        (def output (send self :jcv))
        (send self :interpolate-smooth tweet mid woof)
      )
     )
     (t  (let* (
                (n (send self :n))
                (mid (max 2 (floor (* .2 n))))
                (smoothed-y (send self :smooth "y" span t))
                (temp (send self :r1 (second smoothed-y)))
                (smoothed-resid (send self :smooth "r1" mid nil))
               )
           (send self :smoothed-y (first smoothed-y))
           (send self :resid smoothed-resid)
          ))
     )
    )
    )
```

```
(defmeth supersmoother-proto :interpolate-smooth (tweet mid woof)
  (let* (
         (n (send self :n))
         (tsmooth (send self :tsmo))
         (msmooth (send self :msmo))
         (wsmooth (send self :wsmo))
         (jcvsmo (send self :jcv))
         (f-list (- jcvsmo mid))
         (smoothed-y (list ))
        )
     (dotimes (i n)
       (setf smoothed-y (append smoothed-y
          (if (> (elt f-list i) 0)
             (interpolate (elt f-list i) mid woof
                          (elt msmooth i) (elt wsmooth i))
             (interpolate (- (elt f-list i)) tweet mid
                          (elt msmooth i) (elt tsmooth i)))))
     )
   (send self :smoothed-y smoothed-y)
  )
)


(defun interpolate (f span1 span2 smooth1 smooth2)
   (let (
        (f (/ f (- span2 span1)))
       )
    (list (+ (* (- 1 f) smooth1) (* f smooth2)))
  )
)




(defmeth supersmoother-proto :x (&optional (val nil set))
   (if set (setf (slot-value 'x) val)
   (slot-value 'x))
)

(defmeth supersmoother-proto :y (&optional (val nil set))
   (if set (setf (slot-value 'y) val)
   (slot-value 'y))
)
```

```
(defmeth supersmoother-proto :w (&optional (val nil set))
   (if set (setf (slot-value 'w) val)
   (slot-value 'w))
)

(defmeth supersmoother-proto :r1 (&optional (val nil set))
   (if set (setf (slot-value 'r1) val)
   (slot-value 'r1))
)

(defmeth supersmoother-proto :r2 (&optional (val nil set))
   (if set (setf (slot-value 'r2) val)
   (slot-value 'r2))
)

(defmeth supersmoother-proto :r3 (&optional (val nil set))
   (if set (setf (slot-value 'r3) val)
   (slot-value 'r3))
)

(defmeth supersmoother-proto :tsmo (&optional (val nil set))
   (if set (setf (slot-value 'tsmo) val)
   (slot-value 'tsmo))
)

(defmeth supersmoother-proto :msmo (&optional (val nil set))
   (if set (setf (slot-value 'msmo) val)
   (slot-value 'msmo))
)

(defmeth supersmoother-proto :wsmo (&optional (val nil set))
   (if set (setf (slot-value 'wsmo) val)
   (slot-value 'wsmo))
)

(defmeth supersmoother-proto :jcv (&optional (val nil set))
   (if set (setf (slot-value 'jcv) val)
   (slot-value 'jcv))
)


(defmeth supersmoother-proto :alpha (&optional (val nil set))
   (if set (setf (slot-value 'alpha) val)
   (slot-value 'alpha))
)
```

```
(defmeth supersmoother-proto :span (&optional (val nil set))
   (if set (setf (slot-value 'span) val)
   (slot-value 'span))
)

(defmeth supersmoother-proto :smoothed-y (&optional (val nil set))
   (if set (setf (slot-value 'smoothed-y) val)
   (slot-value 'smoothed-y))
)

(defmeth supersmoother-proto :n (&optional (val nil set))
   (if set (setf (slot-value 'n) val)
   (slot-value 'n))
)

(defmeth supersmoother-proto :resid (&optional (val nil set))
   (if set (setf (slot-value 'resid) val)
   (slot-value 'resid))
)
```