R documentation

of ‘fda.la.Rd’

April 1, 2023

fda.la *Locally Adaptive Functional Data Analysis*

Description

fda.la() uses a variety of nonparametric methods to adaptively learn key unknown local parameters for a class of irregular functions in a Functional Data Analysis (FDA) framework. Local kernel smoothing methods are used, and key unknown parameters in the pointwise-optimal bandwidth formulas are learned adaptively from supplied data. The class of irregular functions lie in the so-called local Hölder class (the typical Hölder class $f : \mathbb{R} \to \mathbb{R}$, and f is Hölder continuous if there exist constant $C > 0$ and exponent $\alpha > 0$ such that $|f(x) - f(y)| \leq C|x - y|^{\alpha}$ for all x and y - note that when $\alpha = 1$ we have the class of Lipschitz functions). The approach is fully data-driven and automatic, requiring nothing more than an object containing a set of functional data.

Smoothing adapts to the unknown regularity of the object being estimated which is characterized by H_t and L_t, the regularity parameters associated with the fractional derivative $(X_u - X_v)/|u - v|^{H_t}$ where $\mathbb{E}((X_u - X_v)^2) \simeq L_t^2|u - v|^{2H_t}$ and where X is functional data.

In particular, for a pointwise-optimal bandwidth for estimating the unknown H_t, we use data-driven methods for obtaining key constants based on the fractional derivative and kernel function adopted (here the Epanechnikov kernel) which deliver estimates of H_t and L_t (these are obtained by a convergent iterative procedure). These same constants appear in the pointwise MSE-optimal bandwidths for estimating a) each online functional curve $X^{(i)}$, $i = 1, \dots, N$, b) the mean functional curve $\mu(t)$, and c) the covariance matrix for the online functional curves $\Gamma(s, t)$. The difference between the smoothing parameters used for estimating H_t and L_t, on one hand, and $\mu(t)$ and $\Gamma(s, t)$, on the other, are the factors of proportionality and rate of convergence. The factors of proportionality can include $1/M_i$, the number of observations for the ith online curve, $1/\sum_{i=1}^{N} M_i$, or $\log(\sum_{i=1}^{N} M_i)/\sum_{i=1}^{N} M_i$, or $1/N$ where N is the number of curves at hand, while the rates of convergence can include $1/(2H_t + 1)$ or $1/(2H_t + 2)$ (the former for estimating $X^{(i)}$ and $\mu(t)$, both the former and latter for estimating $\Gamma(s, t)$).

This function can be used for optimal smoothing for a batch of curves simultaneously, or for optimal smoothing of a sequence of new online curves using a dynamic recursive updating algorithm. When used for estimating newly acquired online curve data, the method is computationally efficient from both CPU time and memory requirements.

Usage

```
fda.la(common.design = FALSE,
       compute.lp.estimator = TRUE,
       curves_batch = NULL,
       curves_online = NULL,
       delta = NULL,
       degenerate.method.HL=c("skip","1nn","lp"),
       degenerate.method.mu=c("skip","1nn","lp"),
       degenerate.method.Gamma=c("skip","1nn","lp"),
       f.hat.grid = NULL,
       fda.la.previous = NULL,
       Gamma.bandwidth = c("logNMsq", "NMsq"),
       Gamma.hat.colSums.vec = NULL,
       Gamma.hat.crossprod.mat = NULL,
       Gamma.hat = NULL,
       h.Gamma = NULL,
       h.mu = NULL,
       h.HL.starting = NULL,
       h.HL = NULL,
       H.lb = 0.1,
       H.ub = 1,
       issue.warnings = TRUE,
       L.sq.lb = .Machine$double.eps,
       mean.correct.HL=TRUE,
       Mi.mean = NULL,
       Mi.sq.mean = NULL
       mu.bandwidth = c("logNM", "M", "NM"),
       mu.hat = NULL,
       mu.method = c("default", "Bspline.cv", "kernel.cv"),
       N = NULL,
       num.iter = 25,
       plot.batch.results = FALSE,
       plot.online.results = FALSE,
       sigma.hat = NULL,
       t.grid = NULL,
       t.lb = 0,
       t.ub = 1,
       theta.hat.t1.t2 = NULL,
       theta.hat.t1.t3 = NULL,
       theta.hat.t2.t3 = NULL,
       X.hat.count=NULL,
       X.hat.mu.count=NULL,
       X.hat.Gamma.count=NULL,
       X.hat.Gamma.count.mat=NULL)
```

Arguments

`common.design`	A logical value indicating whether the curves have common design (`common.design=TRUE` or independent design `common.design=FALSE`)
`compute.lp.estimator`	A logical value that determines whether to compute the linear projection estimator for each curve using the estimated $\Gamma(s,t)$ and sample points for each curve

`curves_batch`	A list, each element containing elements `curves_batch[[i]]$t` and `curves_batch[[i]]$x`, $i = 1, ..., N$, where N is the number of curves in the batch, `t` the design points for the ith curve, and `x` the sample points
`curves_online`	A list, each element containing elements `curves_online[[i]]$t` and `curves_online[[i]]$x`, $i = 1, ..., N.online$, where $N.online$ is the number of online curves, `t` the design points for the ith curve, and `x` the sample points
`delta`	A parameter used to estimate H_t that defaults to $\Delta^*/4$ where $\Delta = e^{-\log(M)^{1/3}}$
`degenerate.method.HL`	A character string that determines how degenerate points (i.e., no observations in $t \pm h$) are handled when computing H_t and L_t for the online curves: `degenerate.method="1nn"` indicates 1st nearest neighbor, `degenerate.method="skip"` indicates to discard the online curve's contribution at each degenerate point (in the "vertical" sense) and return `NA`, and codedegenerate.method="lp" indicates to use a linear curve reconstruction method based on the estimated $\Gamma(s, t)$
`degenerate.method.mu`	A character string that determines how degenerate points (i.e., no observations in $t \pm h$) are handled when computing $\mu(t)$ for the online curves: `degenerate.method="1nn"` indicates 1st nearest neighbor, `degenerate.method="skip"` indicates to discard the online curve's contribution at each degenerate point (in the "vertical" sense) and return `NA`, and codedegenerate.method="lp" indicates to use a linear curve reconstruction method based on the estimated $\Gamma(s, t)$
`degenerate.method.Gamma`	A character string that determines how degenerate points (i.e., no observations in $t \pm h$) are handled when computing $\Gamma(s, t)$ for the online curves: `degenerate.method="1nn"` indicates 1st nearest neighbor, `degenerate.method="skip"` indicates to discard the online curve's contribution at each degenerate point (in the "vertical" sense) and return `NA`, and codedegenerate.method="lp" indicates to use a linear curve reconstruction method based on the estimated $\Gamma(s, t)$
`f.hat.grid`	When called recursively, the density estimates for the design points t evaluated on the grid of points t_0
`fda.la.previous`	When called recursively, output from previous invocation of `fda.la` stored in object via `fda.la.previous <- fda.la(...)` is used to retrieve all necessary objects for performing recursion, alternately one can provide them manually
`Gamma.bandwidth`	When called recursively, the vector of bandwidths used to compute $\Gamma(s, t)$ evaluated on the grid of points t_0
`Gamma.hat.colSums.vec`	When called recursively, the vector of column sums used to compute $\Gamma(s, t)$ evaluated on the grid of points t_0
`Gamma.hat.crossprod.mat`	When called recursively, the matrix of cross product terms used to compute $\Gamma(s, t)$ evaluated on the grid of points t_0
`Gamma.hat`	The estimate of the covariance matrix $\Gamma(s, t)$ evaluated on the grid of points t_0
`h.Gamma`	The vector of bandwidths used to compute the covariance matrix $\Gamma(s, t)$ evaluated on the 2×2 matrix of grid points t_0

`h.mu`	The vector of bandwidths used to compute the mean function $\mu(t)$ evaluated on the grid of points t_0
`h.HL.starting`	A scalar for uninitialized bandwidths used to generate a vector of starting values to compute the regularization vectors H_t and L_t evaluated on the grid of points t_0
`h.HL`	The vector of bandwidths used to compute the regularization vectors H_t and L_t evaluated on the grid of points t_0
`H.lb`	Lower bound (scalar) provided for the regularization vector H_t evaluated on the grid of points t_0
`H.ub`	Upper bound (scalar) provided for the regularization vector H_t evaluated on the grid of points t_0
`issue.warnings`	Logical that determines whether to issue warnings as they occur or not (`TRUE` issues them immediately)
`L.sq.lb`	Lower bound (scalar) provided for the regularization vector L_t^2 evaluated on the grid of points t_0
`mean.correct.HL`	A logical value that determines whether to mean correct individual curves used to construct H_t and L_t
`Mi.mean`	When called recursively, the mean sample size for all curves called previously, but also returned by invocation of the function
`Mi.sq.mean`	When called recursively, the mean of the squared sample size for all curves called previously, but also returned by invocation of the function
`mu.bandwidth`	The vector of bandwidths used to compute the mean function $\mu(t)$ evaluated on the grid of points t_0
`mu.hat`	The estimated mean function $\mu(t)$ evaluated on the grid of points t_0
`mu.method`	When invoked on a batch of curves, whether to use the default kernel smoothing method to estimate the mean function $\mu(t)$ or, alternatively, to use cross-validated B-splines or cross-validated kernel smoothing with a global (scalar) smoothing parameter
`N`	The number of curves involved from previous invocations of the function (used to inform subsequent online curve estimation)
`num.iter`	An integer provided that determines the number of iterations of the batch of curves used to determine the vector of smoothing parameters `h.HL` for determining the regularization parameter vectors H_t and L_t that are evaluated on the grid of points t_0
`plot.batch.results`	A logical value that determines whether to produce some simple graphical summaries when the function is invoked on a batch of curves
`plot.online.results`	A logical value that determines whether to produce some simple graphical summaries when the function is invoked on a set of online curves
`sigma.hat`	When called recursively, the estimated vector of σ_t evaluated on the grid of points t_0
`t.grid`	The grid of points t_0 provided by the user
`t.lb`	The lower bound of the support of the online and batch curves t_{lb} (if not provided will use minimum of the empirical support)

`t.ub`	The upper bound of the support of the online and batch curves t_{ub} (if not provided will use maximum of the empirical support)
`theta.hat.t1.t2`	When called recursively, a set of estimates used to compute the regularization parameters H_t and L_t
`theta.hat.t1.t3`	When called recursively, a set of estimates used to compute the regularization parameters H_t and L_t
`theta.hat.t2.t3`	When called recursively, a set of estimates used to compute the regularization parameters H_t and L_t
`X.hat.count`	A vector of counts representing the number of skipped curves (degenerate cases poitwise) at each grid point for the computation of H_t and L_t, which is non-zero when `degenerate.method.HT="skip"`
`X.hat.mu.count`	A vector of counts representing the number of skipped curves (degenerate cases poitwise) at each grid point for the computation of $\mu(t)$, which is non-zero when `degenerate.method.mu="skip"`
`X.hat.Gamma.count`	A vector of counts representing the number of skipped curves (degenerate cases poitwise) at each grid point for the computation of the mean vector used to compute $\Gamma(s,t)$, which is non-zero when `degenerate.method.Gamma="skip"`
`X.hat.Gamma.count.mat`	A matrix of counts representing the number of skipped curves (degenerate cases pointwise) at each grid point for the computation of the crossproduct vector used to compute $\Gamma(s,t)$, which is non-zero when `degenerate.method.Gamma="skip"`

Details

This function contains a variety of optional arguments which allow for a range of tasks to be accomplished, and there are two ways to think about using this function depending on whether you are interested in conducting analysis on a "batch" of curves or, instead, conducting "online" analysis of one (or more) newly arrived curves having already processed a batch of curves.

In batch mode, a set of N existing curves are used to compute various objects. The process is self-starting and uses sensible defaults for all objects (bandwidths etc.) save for the "mean curve" (it is recommended that the user provide some initial mean curve based on the pooled data, or use the function `mu.pooled.func()` - see the example below for an illustration). In batch mode all data is used, i.e., for curves with no sample points within a bandwidth's reach of a grid point the first nearest neighbour is used (`degenerate.method.*="1nn"`) rather than, say, discarding that curve's information. It is intended that batch mode is used for a relatively small number of existing curves where batch computation can be done within the existing memory footprint of your computer.

Essentially, batch mode proceeds from arbitrary starting values for all objects and then iterates the process to produce counterparts to the arbitrary starting values that are obtained from optimally smoothed curves where the optimal bandwidths are obtained via an interative procedure that converges quite quickly (after roughly 10 or so iterations, experience would indicate).

Typically we envision that batch mode is used for feasibly computing a set of existing curves that will be used for subsequent updating as, for instance, when new curves become available either one at a time or as a set.

The salient difference between the two modes of calling this function is that in online mode one takes the results from the initial batch run and updates these results using recursion based on a newer set of curves drawn from the same stochastic process as that for the original batch. This way, one can call the function each time a new curve arises and recursively update the prior estimates. In this manner, the estimates are refined and as the number of curves processed increases the estimates will themselves stabilize. Alternatively, one could take a set of new curves and process them using online mode, and they will be treated recursively as if each curve was appearing in succession. This method of processing is computationally expedient requiring minimal memory and computation even when when the number of previously processed curves increases without limit.

Some very basic plotting can be done for batch or online invocation with plots being specific to each.

Value

This function returns...

`degenerate.method.HL`	per above
`degenerate.method.mu`	per above
`degenerate.method.Gamma`	per above
`delta`	per above
`f.hat.grid`	per above post recursion
`Gamma.bandwidth`	per above post recursion
`Gamma.hat.colSums.vec`	per above post recursion
`Gamma.hat.crossprod.mat`	per above post recursion
`Gamma.hat`	per above post recursion
`Gamma.hat.sigma.correction`	Gamma.hat with diagonal corrected by substraction of sigma.hat estimate post recursion
`Gamma.hat.flattop.correction`	Gamma.hat with diagonal band corrected by flattop method post recursion
`h.Gamma`	per above post recursion
`h.mu`	per above post recursion
`h.HL.updated.mat`	matrix of updated values of h.HL post-recursion over all recursions
`h.HL.updated`	updated value of h.HL post-recursion for last recursion
`h.HL`	per above post recursion
`h.HL.starting`	per above
`H.updated.mat`	matrix of updated values of H post-recursion over all recursions
`H.updated`	updated value of H post-recursion for last recursion
`issue.warnings`	per above
`length.grid`	per above
`L.sq.updated.mat`	matrix of updated values of L^2 post-recursion over all recursions

`L.sq.updated`	updated value of L^2 post-recursion for last recursion
`mean.correct.HL`	per above
`Mi.mean`	per above post recursion
`Mi.sq.mean`	per above post recursion
`mu.hat.mat`	per above post-recursion over all recursions
`mu.hat`	per above post recursion
`mu.bandwidth`	per above post recursion
`N`	per above
`plot.online.results`	per above
`sigma.hat`	per above post recursion
`t.grid`	per above
`theta.hat.t1.t2`	per above post recursion
`theta.hat.t1.t3`	per above post recursion
`theta.hat.t2.t3`	per above post recursion
`X.hat`	estimated curve matrix
`X.hat.lp`	estimated curve matrix using lp reconstruction
`X.hat.count`	per above post recursion
`X.hat.mu.count`	per above post recursion
`X.hat.Gamma.count`	per above post recursion
`X.hat.Gamma.count.mat`	per above post recursion

Note

There are some functions that operate on the list of curves provided: `plot.data(curves)`, `plot.curves(curves)` (for simulated data containing `curves$x_true` and `curves$grid_true`), and `summary.data(curves,kable=TRUE/FAL`
that provides summary information on the curves with regards to sample points etc.

Note that the option `common.design` triggers...

Author(s)

Valentin Patilea <`valentin.patilea@gmail.com`>, Jeffrey S. Racine <`racinej@mcmaster.ca`>

Examples

```
## Require libraries for 3D plotting and multivariate random number generation

require(GA)
require(MASS)

## Set parameters here
## Regularity parameters for generating data

h_first = 0.25
h_second = 0.75
h_slope = 10
change_point_t = 0.5

## Parameters for generating data from a Brownian motion

sigma = 0.25
tau = 1
L = 1

## Set size of batch (N), (average) number of observations for each online curve
## (M), number of online curves, and grid of points on which to estimate curves

N = 100
M = 100
N.online = 1
t0.grid = seq(0,1,length=25)

## Set the mean function

mu.func <- function(x) { sin(2*pi*x) }

## Generate batch data

points_dist <- functional::Curry(runif, min = 0, max = 1)

points_list <- generates_equal_length_points(N = N, m = M,
                                             distribution = points_dist)

batch <- generate_curves(points_list,
                         hurst = hurst_logistic,
                         grid = t0.grid,
                         sigma = sigma,
                         tau = tau,
                         L = L)

curves_batch <- lapply(batch$curves,
                       function(x) list(t = x$observed$t,
                                        x = x$observed$x + mu.func(x$observed$t),
                                        grid_true = x$ideal$t,
                                        x_true = x$ideal$x + mu.func(x$ideal$t)))

## Call the function on the batch using a preliminary estimator of the pooled mean
## (the preliminary estimator is replaced by the proposed estimator of mu(t), it is
## only used for computing H_t and L_t for the batch)

mu.starting <- mu.pooled.func(curves_batch,t0.grid,method="np")

fda.la.out <- fda.la(curves_batch=curves_batch,
                     t.grid=t0.grid,
                     mu.hat=mu.starting,
                     plot.batch.results=TRUE,
                     plot.online.results=TRUE)
```

```
## Generate online data

points_dist <- functional::Curry(runif, min = 0, max = 1)

points_list <- generates_increasing_length_points(N = N.online, m = M,
                                                  distribution = points_dist)

online <- generate_curves(points_list,
                          hurst = hurst_logistic,
                          grid = t0.grid,
                          sigma = sigma,
                          tau = tau,
                          L = L)

curves_online <- lapply(online$curves,
                        function(x) list(t = x$observed$t,
                                         x = x$observed$x + mu.func(x$observed$t),
                                         grid_true = x$ideal$t))

## Call the function on the online data (call with plot.online.results=TRUE to
## create a simple plot that overwrites the batch plot generated above)

fda.la.out <- fda.la(curves_online=curves_online,
                     fda.la.previous=fda.la.out)

## Or you can call the function on the online data and pass all required objects
## instead of simply inputting fda.la.out (previous invocation on batch)

## fda.la.out <- fda.la(curves_online=curves_online,
##                      plot.online.results=TRUE,
##                      t.grid=t0.grid,
##                      delta=fda.la.out$delta,
##                      N=fda.la.out$N,
##                      mu.hat=fda.la.out$mu.hat,
##                      Gamma.hat=fda.la.out$Gamma.hat,
##                      Gamma.hat.crossprod.mat=fda.la.out$Gamma.hat.crossprod.mat,
##                      Gamma.hat.colSums.vec=fda.la.out$Gamma.hat.colSums.vec,
##                      Mi.mean=fda.la.out$Mi.mean,
##                      Mi.sq.mean=fda.la.out$Mi.sq.mean,
##                      sigma.hat=fda.la.out$sigma.hat,
##                      f.hat.grid=fda.la.out$f.hat.grid,
##                      h.HL=fda.la.out$h.HL,
##                      h.mu=fda.la.out$h.mu,
##                      h.Gamma=fda.la.out$h.Gamma,
##                      theta.hat.t1.t3=fda.la.out$theta.hat.t1.t3,
##                      theta.hat.t1.t2=fda.la.out$theta.hat.t1.t2,
##                      theta.hat.t2.t3=fda.la.out$theta.hat.t2.t3,
##                      X.hat.count=fda.la.out$X.hat.count,
##                      X.hat.mu.count=fda.la.out$X.hat.mu.count,
##                      X.hat.Gamma.count=fda.la.out$X.hat.Gamma.count,
##                      X.hat.Gamma.count.mat=fda.la.out$X.hat.Gamma.count.mat)
```

Index

fda.la, 1