Package 'UMR'

September 2, 2020

Title Unmatched Monotone Regression

Version 1.0.0

Description Unmatched regression refers to the regression setting where
covariates and predictors are collected separately/independently and so are not paired
together, as in the usual regression setting. Balab-
daoui, Doss, and Durot (2020) <arXiv:2007.00830>
study the unmatched regression setting where the univariate regression function is known to
be monotone. This package implements (two) gradient descent method(s) to
compute the estimator developed in Balabdaoui, Doss, and Durot (2020).

License GPL (>= 3)

Encoding UTF-8

LazyData true

Depends decon

Suggests purrr, distr, Iso

RoxygenNote 7.1.1

NeedsCompilation no

Author Charles Doss [aut, cre] (<https://orcid.org/0000-0003-1364-5222>)

Maintainer Charles Doss <cdoss@stat.umn.edu>

Repository CRAN

Date/Publication 2020-09-02 09:00:07 UTC

R topics documented:

AA *Helper functions for calculating gradient of least-squares Shuffled Isotonic Regression criterion, for Laplace or for Gaussian errors*

Description

Helper functions for calculating gradient of least-squares Shuffled Isotonic Regression criterion, for Laplace or for Gaussian errors

Usage

```
AA(yy, mm, func)

BB(mm, func)

AAfunc_Laplace_generic(dd, LL)

AAfunc_Gauss_generic(dd, sig)

BBfunc_Laplace_generic(dd, LL)

BBfunc_Gauss_generic(dd, sig)

getAAfunc_est_outer(eps, ww = 1/length(eps))

getBBfunc_est_outer(eps, ww = 1/length(eps))
```

Arguments

yy	Y (response) observation vector (numeric). Will apply as.vector() so it may be a matrix or array with all dimensions trivial except 1.
mm	Current (unsorted) estimate/iterate at which to compute gradient. (Length equals length of yy). Will apply as.vector() so it may be a matrix or array with all dimensions trivial except 1.
func	This is a function; should be the actual "A" or "B" function from the paper; AA and BB are just wrappers that call outer() with func(). func() should accept vector or matrix arguments.
dd	generic argument to the "A" function; usually of the form m - mmhat, where m is just some value of the regression function
LL	Double Exponential "mean" parameter: corresponding density is $exp(-|d|/LL) / (2LL)$.
sig	is standard deviation of the normal distribution.
eps	is a vector of residuals (or estimated residuals). In current coding, it should have been preprocessed to be *unique*. (If there are repeats this should be encoded in ww).

ww — is vector of weights of same length as eps, and summing to 1. Default is a weight of 1/length(eps) for each value of eps; if eps has been pre-binned then ww is the weights from binning.

Details

See helper functions "A" and "B" in paper.

For getAAfunc_est: returns a function(yy,mm) which is analogous to passing in an estimated 'func' argument to the AA function. (Reason to not do it that way relates to making sure matrix arguments are handled correctly.)

getBBfunc_est returns a function which as of this coding *MUST* take only a numeric vector of length 1; longer vectors will not work. Be careful! Note that ecdf objects are not intended to be stored permanently so storing functions returned by getBBfunc_est_outer or getAAfunc_est_outer may cause issues.

Examples

```
 ## the "!!" de-quote (see ?partial) so e.g., can save mygradSIR for future runs.

####### gradient settings/setup for Gaussian


## set.seed(501)
library(distr)
mysig <- 1 ##  std dev
errdist <- Norm(0, sd=mysig)
mm0 <- function(xx){xx}
nn <- 300
xx <- sort(runif(n=nn, 0, 7))
yy <- mm0(xx) + errdist@r(nn)
## plot(xx,yy)

myScale <- mysig

AAfunc_Gauss <- purrr::partial(AAfunc_Gauss_generic, sig=!!mysig)
AA_Gauss <- purrr::partial(AA, func=!!AAfunc_Gauss)
BBfunc_Gauss <- purrr::partial(BBfunc_Gauss_generic, sig=!!mysig)
BB_Gauss <- purrr::partial(BB, func=!!BBfunc_Gauss)
mygradSIR <-
    grad_SIR_Gauss <- ## just for ease of reference
        purrr::partial(grad_SIR_generic,
                       rescale=TRUE, ## factor of nn/2
                       AAfunc=!!AA_Gauss, BBfunc=!!BB_Gauss)

####### gradient settings/setup for Laplace
```

```
set.seed(501)
library(distr)
myLL <- .7 ## (1/"rate") parameter, aka "mean" parameter (except Laplace mean is 0)
errdist <- DExp(1/myLL)


nn <- 200
mm0 <- function(xx){
   (xx<=0)*0 + (0<=xx & xx<=2)*1 +
       (2<xx & xx<=3)*3 +
       (3<xx)*6
}
xx <- sort(runif(n=nn, 0, 7))
yy <- mm0(xx) + errdist@r(nn)

myScale <- myLL;

## CS settings
#'mysig <- sqrt(2) * myLL;
#'
AAfunc_Laplace <- purrr::partial(AAfunc_Laplace_generic, LL=!!myLL)
AA_Laplace <- purrr::partial(AA, func=!!AAfunc_Laplace)
BBfunc_Laplace <- purrr::partial(BBfunc_Laplace_generic, LL=!!myLL)
BB_Laplace <- purrr::partial(BB, func=!!BBfunc_Laplace)
mygradSIR <-
    grad_SIR_Laplace <- purrr::partial(grad_SIR_generic,
                                       rescale=TRUE, ## factor of nn/2
                                      AAfunc=!!AA_Laplace, BBfunc=!!BB_Laplace)
```

`gradDesc` *Basic gradient descent implementation*

Description

Basic gradient descent implementation

Usage

```
gradDesc(yy, grad, init, stepsize, MM, printevery, filename)
```

Arguments

`yy`	Y (response) observation vector (numeric)
`grad`	a function(yy, mm) where mm is same length of yy and is the previous iterate value (i.e., the estimate vector).
`init`	Initial value of estimate ('mm'). I.e., numeric vector of same length as yy.
`stepsize`	Gradient descent stepsize. Set carefully! (I often use nn^2 / 2 where nn = length(yy), or nn if gradient is 'rescaled'.)

MM	Number of iterations
printevery	integer value (generally « MM). Every 'printevery' iterations, a count will be printed and the output saved.
filename	filename (path) to save output to.

Details

Implements a very basic gradient descent. Right now stepsize is fixed.

Examples

```
#### Set up the gradient function
 mysig <- 1 ##  std dev
errdist <- distr::Norm(0, sd=mysig)
modeldistname <- truedistname <- "Gauss" ## used for savefile name
mm0 <- function(xx){xx}
nn <- 300
xx <- sort(runif(n=nn, 0, 7))
yy <- mm0(xx) + errdist@r(nn)
## plot(xx,yy)

myScale <- mysig

AAfunc_Gauss <- purrr::partial(AAfunc_Gauss_generic, sig=!!mysig)
AA_Gauss <- purrr::partial(AA, func=!!AAfunc_Gauss)
BBfunc_Gauss <- purrr::partial(BBfunc_Gauss_generic, sig=!!mysig)
BB_Gauss <- purrr::partial(BB, func=!!BBfunc_Gauss)
mygradSIR <-
    grad_SIR_Gauss <- ## just for ease of reference
        purrr::partial(grad_SIR_generic,
                       rescale=TRUE, ## factor of nn/2
                       AAfunc=!!AA_Gauss, BBfunc=!!BB_Gauss)

 ## Now run the gradient descent
savefilenameUnique <- paste("graddesc_", modeldistname, "_", truedistname,
                            "_n", nn,
                            "_", format(Sys.time(), "%Y-%m-%d-%T"), ".rsav", sep="")
print(paste("The unique save file name for this run is", savefilenameUnique))
stepsize <- nn^(1/2) ## Has to be tuned
MM <-  100 ## Total number iterations is MM * JJ
JJ <- 2
eps <- (max(yy)-min(yy)) / (1000 * nn^(1/5) * myScale)
## print *and* SAVE every 'printevery' iterations.
## here no save occurs, printevery > MM
printevery <- 1000
init <- yy

mmhat <- gradDesc(yy=yy, grad=mygradSIR, ## from settings file
                      init=init,
                      stepsize=stepsize, MM=MM,
```

```
                    printevery=printevery,
                    filename=paste0("../saves/", savefilenameUnique))
#### some classical/matched [oracle] estimators
isoreg_std <- Iso::ufit(y=yy, x=xx, lmode=Inf)
mmhat_std = isoreg_std$y ## Isotonic regression
linreg_std <- lm(yy~xx)
```

`gradDesc_PC` *Gradient Descent implemented for Piecewise Constant functions*

Description

Gradient Descent implemented for Piecewise Constant functions

Usage

```
gradDesc_PC(yy, grad, init, stepsize, MM, eps, JJ = 50, printevery, filename)
```

Arguments

`yy`	Y (response) observation vector (numeric)
`grad`	a function(yy, mm) where mm may be shorter length than yy and is the previous iterate value (i.e., the estimate vector).
`init`	Initial value of estimate ('mm'). I.e., numeric vector usually of same length as yy.
`stepsize`	Gradient descent stepsize. Set carefully!
`MM`	Number of iterations in which "support reduction" (combining of approximately equal values into a region of constancy) is done (see details and paper).
`eps`	Roughly, points that are eps apart are considered to be equal and are thus collapsed into a single region of piecewise constancy of the output. (This is not precisely true because one can have a long sorted-increasing vector of points that are each eps from their two neighboring points but such that the first and last points are not eps apart. See algorithm description in paper for details.)
`JJ`	Total number of gradient steps is MM*JJ. JJ gradient steps are taken for each of the MM steps.
`printevery`	integer value (generally « MM). Every 'printevery' iterations, a count will be printed and the output saved.
`filename`	path1/path2/filename to save output to.

Details

Implements a gradient descent. See paper for details. Right now stepsize is fixed. Right now: init gets sorted in gradDesc_PC so does not need to be sorted on input. Roughly, the difference between this algorithm and gradDesc() (which is just vanilla gradient descent on this problem) is that: if mm is the current value of the output estimate, then gradDesc_PC 'collapses' or combines values of mm that are (roughly, up to tolerance 'eps') equal. Because the solution is generally piecewise constant with a relatively small number of constant regions this enormously speeds up the later stages of the algorithm. Note that once points are combined/collapsed they contribute identically to the objective function, so they will never be "uncombined".

Examples

```
#'
#### Set up the gradient function
 mysig <- 1 ##  std dev
errdist <- distr::Norm(0, sd=mysig)
modeldistname <- truedistname <- "Gauss" ## used for savefile name
mm0 <- function(xx){xx}
nn <- 300
xx <- sort(runif(n=nn, 0, 7))
yy <- mm0(xx) + errdist@r(nn)
## plot(xx,yy)

myScale <- mysig

AAfunc_Gauss <- purrr::partial(AAfunc_Gauss_generic, sig=!!mysig)
AA_Gauss <- purrr::partial(AA, func=!!AAfunc_Gauss)
BBfunc_Gauss <- purrr::partial(BBfunc_Gauss_generic, sig=!!mysig)
BB_Gauss <- purrr::partial(BB, func=!!BBfunc_Gauss)
mygradSIR <-
    grad_SIR_Gauss <- ## just for ease of reference
        purrr::partial(grad_SIR_generic,
                       rescale=TRUE, ## factor of nn/2
                       AAfunc=!!AA_Gauss, BBfunc=!!BB_Gauss)

 ## Now run the gradient descent
savefilenameUnique <- paste("graddesc_", modeldistname, "_", truedistname,
                            "_n", nn,
                            "_", format(Sys.time(), "%Y-%m-%d-%T"), ".rsav", sep="")
print(paste("The unique save file name for this run is", savefilenameUnique))
stepsize <- nn^(1/2) ## Has to be tuned
MM <-  200 ## Total number iterations is MM * JJ
JJ <- 2
eps <- (max(yy)-min(yy)) / (1000 * nn^(1/5) * myScale)
## print *and* SAVE every 'printevery' iterations;
## here no save occurs, printevery > MM
printevery <- 1000
init <- yy
```

```
 mmhat <- gradDesc_PC(yy=yy, grad=mygradSIR, ## from settings file
                      init=init,
                      stepsize=stepsize, MM=MM,
                      JJ=JJ, eps=eps,
                      printevery=printevery,
                      filename=paste0("../saves/", savefilenameUnique))
####  some classical/matched [oracle] estimators
isoreg_std <- Iso::ufit(y=yy, x=xx, lmode=Inf)
mmhat_std = isoreg_std$y ## Isotonic regression
linreg_std <- lm(yy~xx)
```

`grad_SIR_generic` *@title Gradient of least-squares Shuffled Isotonic Regression criterion*

Description

@title Gradient of least-squares Shuffled Isotonic Regression criterion

Usage

```
grad_SIR_generic(
  yy,
  mm,
  counts = rep(1, length(mm)),
  AAfunc,
  BBfunc,
  rescale = FALSE
)
```

Arguments

`yy`	Y (response) observation vector (numeric)
`mm`	Current (unsorted) estimate/iterate at which to compute gradient. (Length equals length of yy).
`counts`	If the function that mm represents is piecewise constant, then mm may be passed in as only the unique entries. In that case counts contains the number of times each element of mm is repeated. Thus length(counts)==length(mm). (Default for counts is thus a vector of all 1's.)
`AAfunc`	This is the function "A" defined in the gradient calculations in the paper (Balabdaoui, Doss, Durot (2020+).
`BBfunc`	This is the function "B" defined in the gradient calculations in the paper (Balabdaoui, Doss, Durot (2020+). @details Returns gradient. See calculations in ShuffReg.pdf. @examples #### See help for gradDesc_PC, gradDesc, or grad_helpers
`rescale`	Boolean: if False then the final return value is the gradient; if True the final return value is gradient * length(yy) / 2.

UMR *UMR: For computing an estimator in Unlinked Monotone Regression.*

Description

A package for computing (via first order, gradient descent type algorithms) an estimator in the problem of univariate Unlinked Monotone Regression. See Balabdaoui, Doss, and Durot (2020+).

UMR functions

The main functions are gradDesc_PC (for Gradient Descent for Piecwise Constant functions) and gradDesc. The former is faster and recommended. The latter is the more naive vanilla gradient descent method (can be used for instance to double check results from gradDesc_PC).

umr_deconv *Carpentier and Schluter 2016 deconvolution method for unmatched monotone regression*

Description

Carpentier and Schluter 2016 deconvolution method for unmatched monotone regression

Usage

```
umr_deconv(xx, yy, sig, error = "normal", bw = "dboot1", adjust = 1, n = 512)

quant_deconv(
  yy,
  sig,
  error = "normal",
  bw = "dboot1",
  adjust = 1,
  n = 512,
  monotonize = base::cummax
)
```

Arguments

xx	X (covariate or predictor) observation vector
yy	Y (response) observation vector (numeric)
sig	standard deviation of epsilon (passed to DeconCdf)
error	Must be "normal" or "laplacian" or "snormal"; see help("DeconCdf")
bw	Bandwidth choice or method for kernel estimator; see help("DeconCdf")
adjust	See help("DeconCdf")

n	See help("DeconCdf")
monotonize	is a function taking a numeric vector argument which returns an increasing numeric vector of the same length. This is used to monotonize the output of the CDF from deconvolution, which is not guaranteed to be a "bona-fide" CDF in the sense that it may not be monotone.

Details

quant_deconv implements Carpentier and Schluter 2016 deconvolution method for unmatched monotone regression, using deconv package. Note that because the DeconCdf() function computes the CDF but there is no direct code for computing the quantile function, we use approxfun to create the quantile function; this may be slow. quant_deconv() returns a vector of length length(yy). Then umr_deconv is a wrapper for quant_deconv. NOTE: It returns the output of approxfun, which is may change over time. The output value is of type function. We linearly interpolate between the points i/n.

Examples

```
library(distr)
mysig <- 1 ##  std dev
errdist <- Norm(0, sd=mysig)
mm0 <- function(xx){xx}
 nn <- 300
 xx <- sort(runif(n=nn, 0, 7))
 yy <- mm0(xx) + errdist@r(nn)
 ## plot(xx,yy)
 modeldistname <- truedistname <- "Gauss" ## used for savefile name
myScale <- mysig

xx <- sort(runif(n=nn, 0, 7))
mmtrue <- mm0(xx)
yy <- mmtrue + errdist@r(nn)
plot(xx,yy)
qq <- quant_deconv(yy, sig=1, error="normal")
lines(xx, ## already sorted
     qq)
```

Index