Package 'FDboost'

August 23, 2025

Type Package

Title Boosting Functional Regression Models

Version 1.1-3

Date 2025-08-08

Description Regression models for functional data, i.e., scalar-on-function, function-on-scalar and function-on-function regression models, are fitted by a component-wise gradient boosting algorithm. For a manual on how to use 'FDboost', see Brockhaus, Ruegamer, Greven (2017) <doi:10.18637/jss.v094.i10>.

License GPL-2

URL https://github.com/boost-R/FDboost

BugReports https://github.com/boost-R/FDboost/issues

Depends mboost (>= 2.9-0), R (>= 3.5.0)

Imports gamboostLSS (>= 2.0-0), graphics, grDevices, MASS, Matrix, methods, mgcv, stabs, stats, utils, zoo

Suggests fda, fields, ggplot2, knitr, mapdata, maps, refund, testthat

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.2

Collate 'aaa.R' 'FDboost-package.R' 'FDboost.R' 'baselearners.R' 'baselearnersX.R' 'bootstrapCIs.R' 'clr_functions.R' 'constrainedX.R' 'crossvalidation.R' 'factorize.R' 'FDboostLSS.R' 'hmatrix.R' 'methods.R' 'stabsel.R' 'utilityFunctions.R'

NeedsCompilation no

Author Sarah Brockhaus [aut] (ORCID: <https://orcid.org/0000-0001-9484-7488>), David Ruegamer [aut, cre] (ORCID: <https://orcid.org/0000-0002-8772-9202>), Almond Stoecker [aut] (ORCID: <https://orcid.org/0000-0001-9160-2397>), Torsten Hothorn [ctb] (ORCID: <https://orcid.org/0000-0001-8301-0471>), with contributions by many others (see inst/CONTRIBUTIONS) [ctb]

Maintainer David Ruegamer <david.ruegamer@gmail.com>

Repository CRAN

Date/Publication 2025-08-23 11:00:02 UTC

Contents

`FDboost-package` *FDboost: Boosting Functional Regression Models*

Description

Regression models for functional data, i.e., scalar-on-function, function-on-scalar and function-on-function regression models, are fitted by a component-wise gradient boosting algorithm.

Details

This package is intended to fit regression models with functional variables. It is possible to fit models with functional response and/or functional covariates, resulting in scalar-on-function, function-on-scalar and function-on-function regression. Furthermore, the package can be used to fit density-on-scalar regression models. Details on the functional regression models that can be fitted with **FDboost** can be found in Brockhaus et al. (2015, 2017, 2018) and Ruegamer et al. (2018). A hands-on tutorial for the package can be found in Brockhaus, Ruegamer and Greven (2020), see <doi:10.18637/jss.v094.i10>. For density-on-scalar regression models see Maier et al. (2021).

Using component-wise gradient boosting as fitting procedure, **FDboost** relies on the R package **mboost** (Hothorn et al., 2017). A comprehensive tutorial to **mboost** is given in Hofner et al. (2014).

The main fitting function is `FDboost`. The model complexity is controlled by the number of boosting iterations (mstop). Like the fitting procedures in **mboost**, the function `FDboost` DOES NOT select an appropriate stopping iteration. This must be chosen by the user. The user can determine an adequate stopping iteration by resampling methods like cross-validation or bootstrap. This can be done using the function `applyFolds`.

Aside from common effect surface plots, tensor product factorization via the function `factorize` presents an alternative tool for visualization of estimated effects for non-linear function-on-scalar models (Stoecker, Steyer and Greven (2022), https://arxiv.org/abs/2109.02624). After factorization, effects are decomposed multiple scalar effects into functional main effect directions, which can be separately plotted allowing to visualize more complex effect structures.

Author(s)

Sarah Brockhaus, David Ruegamer and Almond Stoecker

References

Brockhaus, S., Ruegamer, D. and Greven, S. (2020): Boosting Functional Regression Models with FDboost. Journal of Statistical Software, 94(10), 1–50. <doi:10.18637/jss.v094.i10>

Brockhaus, S., Scheipl, F., Hothorn, T. and Greven, S. (2015): The functional linear array model. Statistical Modelling, 15(3), 279-300.

Brockhaus, S., Melcher, M., Leisch, F. and Greven, S. (2017): Boosting flexible functional regression models with a high number of functional historical effects, Statistics and Computing, 27(4), 913-926.

Brockhaus, S., Fuest, A., Mayr, A. and Greven, S. (2018): Signal regression models for location, scale and shape with an application to stock returns. Journal of the Royal Statistical Society: Series C (Applied Statistics), 67, 665-686.

Hothorn T., Buehlmann P., Kneib T., Schmid M., and Hofner B. (2017). mboost: Model-Based Boosting, R package version 2.8-1, https://cran.r-project.org/package=mboost

Hofner, B., Mayr, A., Robinzonov, N., Schmid, M. (2014). Model-based Boosting in R: A Hands-on Tutorial Using the R Package mboost. Computational Statistics, 29, 3-35. https://cran.r-project.org/package=mboost/vignettes/mboost_tutorial.pdf

Maier, E.-M., Stoecker, A., Fitzenberger, B., Greven, S. (2021): Additive Density-on-Scalar Regression in Bayes Hilbert Spaces with an Application to Gender Economics. arXiv preprint arXiv:2110.11771.

Ruegamer D., Brockhaus, S., Gentsch K., Scherer, K., Greven, S. (2018). Boosting factor-specific functional historical models for the detection of synchronization in bioelectrical signals. Journal of the Royal Statistical Society: Series C (Applied Statistics), 67, 621-642.

Stoecker A., Steyer L., Greven S. (2022): Functional Additive Models on Manifolds of Planar Shapes and Forms. arXiv preprint arXiv:2109.02624.

See Also

FDboost for the main fitting function and applyFolds for model tuning via resampling methods.

`anisotropic_Kronecker` *Kronecker product or row tensor product of two base-learners with anisotropic penalty*

Description

Kronecker product or row tensor product of two base-learners allowing for anisotropic penalties. For the Kronecker product, `%A%` works in the general case, `%A0%` for the special case where the penalty is zero in one direction. For the row tensor product, `%Xa0%` works for the special case where the penalty is zero in one direction.

Usage

```
bl1 %A% bl2

bl1 %A0% bl2

bl1 %Xa0% bl2
```

Arguments

bl1	base-learner 1, e.g. bbs(x1)
bl2	base-learner 2, e.g. bbs(x2)

Details

When %O% is called with a specification of df in both base-learners, e.g. bbs(x1, df = df1) %O% bbs(t, df = df2), the global df for the Kroneckered base-learner is computed as df = df1 * df2. And thus the penalty has only one smoothness parameter lambda resulting in an isotropic penalty,

$$P = lambda * [(P1 o I) + (I o P2)],$$

with overall penalty P, Kronecker product o, marginal penalty matrices $P1, P2$ and identity matrices I. (Currie et al. (2006) introduced the generalized linear array model, which has a design matrix that is composed of the Kronecker product of two marginal design matrices, which was implemented in mboost as %O%. See Brockhaus et al. (2015) for the application of array models to functional data.)

In contrast, a Kronecker product with anisotropic penalty is obtained by %A%, which allows for a different amount of smoothness in the two directions. For example bbs(x1, df = df1) %A% bbs(t, df = df2) results in computing two different values for lambda for the two marginal design matrices and a global value of lambda to adjust for the global df, i.e.

$$P = lambda * [(lambda1 * P1 o I) + (I o lambda2 * P2)],$$

with Kronecker product o, where $lambda1$ is computed individually for $df1$ and $P1$, $lambda2$ is computed individually for $df2$ and $P2$, and $lambda$ is computed such that the global df hold $df = df1 * df2$. For the computation of $lambda1$ and $lambda2$ weights specified in the model call can only be used when the weights, are such that they are specified on the level of rows and columns of the response matrix Y, e.g. resampling weights on the level of rows of Y and integration weights on the columns of Y are possible. If this the weights cannot be separated to blg1 and blg2 all weights are set to 1 for the computation of $lambda1$ and $lambda2$ which implies that $lambda1$ and $lambda2$ are equal over folds of cvrisk. The computation of the global $lambda$ considers the specified weights, such the global df are correct.

The operator %A0% treats the important special case where $lambda1 = 0$ or $lambda2 = 0$. In this case it suffices to compute the global lambda and computation gets faster and arbitrary weights can be specified. Consider $lambda1 = 0$ then the penalty becomes

$$P = lambda * [(1 * P1 o I) + (I o lambda2 * P2)] = lambda * lambda2 * (I o P2),$$

and only one global $lambda$ is computed which is then $lambda * lambda2$.

If the formula in FDboost contains base-learners connected by %O%, %A% or %A0%, those effects are not expanded with timeformula, allowing for model specifications with different effects in time-direction.

%Xa0% computes like %X% the row tensor product of two base-learners, with the difference that it sets the penalty for one direction to zero. Thus, %Xa0% behaves to %X% analogously like %A0% to %O%.

Value

An object of class blg (base-learner generator) with a dpp function as for other baselearners.

References

Brockhaus, S., Scheipl, F., Hothorn, T. and Greven, S. (2015): The functional linear array model. Statistical Modelling, 15(3), 279-300.

Currie, I.D., Durban, M. and Eilers P.H.C. (2006): Generalized linear array models with applications to multidimensional smoothing. Journal of the Royal Statistical Society, Series B-Statistical Methodology, 68(2), 259-280.

Examples

```
######## Example for anisotropic penalty
data("viscosity", package = "FDboost")
## set time-interval that should be modeled
interval <- "101"

## model time until "interval" and take log() of viscosity
end <- which(viscosity$timeAll == as.numeric(interval))
viscosity$vis <- log(viscosity$visAll[,1:end])
viscosity$time <- viscosity$timeAll[1:end]
# with(viscosity, funplot(time, vis, pch = 16, cex = 0.2))

## isotropic penalty, as timeformula is kroneckered to each effect using %O%
## only for the smooth intercept %A0% is used, as 1-direction should not be penalized
mod1 <- FDboost(vis ~ 1 +
                bolsc(T_C, df = 1) +
                bolsc(T_A, df = 1) +
                bols(T_C, df = 1) %Xc% bols(T_A, df = 1),
                timeformula = ~ bbs(time, df = 3),
                numInt = "equal", family = QuantReg(),
                offset = NULL, offset_control = o_control(k_min = 9),
                data = viscosity, control=boost_control(mstop = 100, nu = 0.4))
## cf. the formula that is passed to mboost
mod1$formulaMboost

## anisotropic effects using %A0%, as lambda1 = 0 for all base-learners
## in this case using %A% gives the same model, but three lambdas are computed explicitly
mod1a <- FDboost(vis ~ 1 +
                bolsc(T_C, df = 1) %A0% bbs(time, df = 3) +
                bolsc(T_A, df = 1) %A0% bbs(time, df = 3) +
                bols(T_C, df = 1) %Xc% bols(T_A, df = 1) %A0% bbs(time, df = 3),
                timeformula = ~ bbs(time, df = 3),
                numInt = "equal", family = QuantReg(),
                offset = NULL, offset_control = o_control(k_min = 9),
                data = viscosity, control=boost_control(mstop = 100, nu = 0.4))
## cf. the formula that is passed to mboost
mod1a$formulaMboost

## alternative model specification by using a 0-matrix as penalty
## only works for bolsc() as in bols() one cannot specify K
## -> model without interaction term
K0 <- matrix(0, ncol = 2, nrow = 2)
mod1k0 <- FDboost(vis ~ 1 +
```

```
                  bolsc(T_C, df = 1, K = K0) +
                  bolsc(T_A, df = 1, K = K0),
                  timeformula = ~ bbs(time, df = 3),
                  numInt = "equal", family = QuantReg(),
                  offset = NULL, offset_control = o_control(k_min = 9),
                  data = viscosity, control=boost_control(mstop = 100, nu = 0.4))
## cf. the formula that is passed to mboost
mod1k0$formulaMboost

## optimize mstop for mod1, mod1a and mod1k0
## ...

## compare estimated coefficients

oldpar <- par(mfrow=c(4, 2))
plot(mod1, which = 1)
plot(mod1a, which = 1)
plot(mod1, which = 2)
plot(mod1a, which = 2)
plot(mod1, which = 3)
plot(mod1a, which = 3)
funplot(mod1$yind, predict(mod1, which=4))
funplot(mod1$yind, predict(mod1a, which=4))
par(oldpar)
```

---

applyFolds *Cross-Validation and Bootstrapping over Curves*

Description

Cross-validation and bootstrapping over curves to compute the empirical risk for hyper-parameter selection.

Usage

```
applyFolds(
  object,
  folds = cv(rep(1, length(unique(object$id))), type = "bootstrap"),
  grid = 1:mstop(object),
  fun = NULL,
  riskFun = NULL,
  numInt = object$numInt,
  papply = mclapply,
  mc.preschedule = FALSE,
  showProgress = TRUE,
  compress = FALSE,
  ...
```

```
)

## S3 method for class 'FDboost'
cvrisk(
  object,
  folds = cvLong(id = object$id, weights = model.weights(object)),
  grid = 1:mstop(object),
  papply = mclapply,
  fun = NULL,
  mc.preschedule = FALSE,
  ...
)

cvLong(
  id,
  weights = rep(1, l = length(id)),
  type = c("bootstrap", "kfold", "subsampling", "curves"),
  B = ifelse(type == "kfold", 10, 25),
  prob = 0.5,
  strata = NULL
)

cvMa(
  ydim,
  weights = rep(1, l = ydim[1] * ydim[2]),
  type = c("bootstrap", "kfold", "subsampling", "curves"),
  B = ifelse(type == "kfold", 10, 25),
  prob = 0.5,
  strata = NULL,
  ...
)
```

Arguments

object	fitted FDboost-object
folds	a weight matrix with number of rows equal to the number of observed trajectories.
grid	the grid over which the optimal number of boosting iterations (mstop) is searched.
fun	if fun is NULL, the out-of-bag risk is returned. fun, as a function of object, may extract any other characteristic of the cross-validated models. These are returned as is.
riskFun	only exists in applyFolds; allows to compute other risk functions than the risk of the family that was specified in object. Must be specified as function of arguments (y, f, w = 1), where y is the observed response, f is the prediction from the model and w is the weight. The risk function must return a scalar numeric value for vector valued input.
numInt	only exists in applyFolds; the scheme for numerical integration, see numInt in FDboost.

papply	(parallel) apply function, defaults to mclapply from R package parallel, see cvrisk for details.
mc.preschedule	Defaults to FALSE. Preschedule tasks if they are parallelized using mclapply. For details see mclapply.
showProgress	logical, defaults to TRUE.
compress	logical, defaults to FALSE. Only used to force a meaningful behaviour of applyFolds with hmatrix objects when using nested resampling.
...	further arguments passed to the (parallel) apply function.
id	the id-vector as integers 1, 2, ... specifying which observations belong to the same curve, deprecated in cvMa().
weights	a numeric vector of (integration) weights, defaults to 1.
type	character argument for specifying the cross-validation method. Currently (stratified) bootstrap, k-fold cross-validation, subsampling and leaving-one-curve-out cross validation (i.e. jack knife on curves) are implemented.
B	number of folds, per default 25 for bootstrap and subsampling and 10 for kfold.
prob	percentage of observations to be included in the learning samples for subsampling.
strata	a factor of the same length as weights for stratification.
ydim	dimensions of response-matrix

Details

The number of boosting iterations is an important hyper-parameter of boosting. It be chosen using the functions applyFolds or cvrisk.FDboost. Those functions compute honest, i.e., out-of-bag, estimates of the empirical risk for different numbers of boosting iterations. The weights (zero weights correspond to test cases) are defined via the folds matrix, see cvrisk in package mboost.

In case of functional response, we recommend to use applyFolds. It recomputes the model in each fold using FDboost. Thus, all parameters are recomputed, including the smooth offset (if present) and the identifiability constraints (if present, only relevant for bolsc, brandomc and bbsc). Note, that the function applyFolds expects folds that give weights per curve without considering integration weights.

The function cvrisk.FDboost is a wrapper for cvrisk in package mboost. It overrides the default for the folds, so that the folds are sampled on the level of curves (not on the level of single observations, which does not make sense for functional response). Note that the smooth offset and the computation of the identifiability constraints are not part of the refitting if cvrisk is used. Per default the integration weights of the model fit are used to compute the prediction errors (as the integration weights are part of the default folds). Note that in cvrisk the weights are rescaled to sum up to one.

The functions cvMa and cvLong can be used to build an appropriate weight matrix for functional response to be used with cvrisk as sampling is done on the level of curves. The probability for each curve to enter a fold is equal over all curves. The function cvMa takes the dimensions of the response matrix as input argument and thus can only be used for regularly observed response. The function cvLong takes the id variable and the weights as arguments and thus can be used for responses in long format that are potentially observed irregularly.

If `strata` is defined sampling is performed in each stratum separately thus preserving the distribution of the `strata` variable in each fold.

Value

`cvMa` and `cvLong` return a matrix of sampling weights to be used in `cvrisk`.

The functions `applyFolds` and `cvrisk.FDboost` return a `cvrisk`-object, which is a matrix of the computed out-of-bag risk. The matrix has the folds in rows and the number of boosting iteratins in columns. Furhtermore, the matrix has attributes including:

`risk`	name of the applied risk function
`call`	model call of the model object
`mstop`	gird of stopping iterations that is used
`type`	name for the type of folds

Note

Use argument `mc.cores = 1L` to set the numbers of cores that is used in parallel computation. On Windows only 1 core is possible, `mc.cores = 1`, which is the default.

See Also

`cvrisk` to perform cross-validation with scalar response.

Examples

```
Ytest <- matrix(rnorm(15), ncol = 3) # 5 trajectories, each with 3 observations
Ylong <- as.vector(Ytest)
## 4-folds for bootstrap for the response in long format without integration weights
cvMa(ydim = c(5,3), type = "bootstrap", B = 4)
cvLong(id = rep(1:5, times = 3), type = "bootstrap", B = 4)

if(require(fda)){
 ## load the data
 data("CanadianWeather", package = "fda")

 ## use data on a daily basis
 canada <- with(CanadianWeather,
                list(temp = t(dailyAv[ , , "Temperature.C"]),
                     l10precip = t(dailyAv[ , , "log10precip"]),
               l10precip_mean = log(colMeans(dailyAv[ , , "Precipitation.mm"]), base = 10),
                     lat = coordinates[ , "N.latitude"],
                     lon = coordinates[ , "W.longitude"],
                     region = factor(region),
                     place = factor(place),
                  day = 1:365,  ## corresponds to t: evaluation points of the fun. response
              day_s = 1:365))  ## corresponds to s: evaluation points of the fun. covariate

## center temperature curves per day
canada$tempRaw <- canada$temp
canada$temp <- scale(canada$temp, scale = FALSE)
```

```
rownames(canada$temp) <- NULL ## delete row-names

## fit the model
mod <- FDboost(l10precip ~ 1 + bolsc(region, df = 4) +
                bsignal(temp, s = day_s, cyclic = TRUE, boundary.knots = c(0.5, 365.5)),
               timeformula = ~ bbs(day, cyclic = TRUE, boundary.knots = c(0.5, 365.5)),
               data = canada)
mod <- mod[75]


  #### create folds for 3-fold bootstrap: one weight for each curve
  set.seed(123)
  folds_bs <- cv(weights = rep(1, mod$ydim[1]), type = "bootstrap", B = 3)

  ## compute out-of-bag risk on the 3 folds for 1 to 75 boosting iterations
  cvr <- applyFolds(mod, folds = folds_bs, grid = 1:75)

  ## weights per observation point
  folds_bs_long <- folds_bs[rep(seq_len(nrow(folds_bs)), times = mod$ydim[2]), ]
  attr(folds_bs_long, "type") <- "3-fold bootstrap"
  ## compute out-of-bag risk on the 3 folds for 1 to 75 boosting iterations
  cvr3 <- cvrisk(mod, folds = folds_bs_long, grid = 1:75)



  ## plot the out-of-bag risk
  oldpar <- par(mfrow = c(1,3))
  plot(cvr); legend("topright", lty=2, paste(mstop(cvr)))
  plot(cvr3); legend("topright", lty=2, paste(mstop(cvr3)))
  par(oldpar)


}
```

---

bbsc *Constrained Base-learners for Scalar Covariates*

Description

Constrained base-learners for fitting effects of scalar covariates in models with functional response

Usage

```
bbsc(
  ...,
  by = NULL,
  index = NULL,
  knots = 10,
  boundary.knots = NULL,
```

```
  degree = 3,
  differences = 2,
  df = 4,
  lambda = NULL,
  center = FALSE,
  cyclic = FALSE
)

bolsc(
  ...,
  by = NULL,
  index = NULL,
  intercept = TRUE,
  df = NULL,
  lambda = 0,
  K = NULL,
  weights = NULL,
  contrasts.arg = "contr.treatment"
)

brandomc(..., contrasts.arg = "contr.dummy", df = 4)
```

Arguments

`...`	one or more predictor variables or one matrix or data frame of predictor variables.
`by`	an optional variable defining varying coefficients, either a factor or numeric variable.
`index`	a vector of integers for expanding the variables in `...`.
`knots`	either the number of knots or a vector of the positions of the interior knots (for more details see `bbs`).
`boundary.knots`	boundary points at which to anchor the B-spline basis (default the range of the data). A vector (of length 2) for the lower and the upper boundary knot can be specified.
`degree`	degree of the regression spline.
`differences`	a non-negative integer, typically 1, 2 or 3. If `differences` = k, k-th-order differences are used as a penalty (0-th order differences specify a ridge penalty).
`df`	trace of the hat matrix for the base-learner defining the base-learner complexity. Low values of `df` correspond to a large amount of smoothing and thus to "weaker" base-learners.
`lambda`	smoothing parameter of the penalty, computed from `df` when `df` is specified.
`center`	See `bbs`.
`cyclic`	if `cyclic` = `TRUE` the fitted values coincide at the boundaries (useful for cyclic covariates such as day time etc.).
`intercept`	if `intercept` = `TRUE` an intercept is added to the design matrix of a linear base-learner.

K	in bolsc it is possible to specify the penalty matrix K
weights	experiemtnal! weights that are used for the computation of the transformation matrix Z.
contrasts.arg	Note that a special contrasts.arg exists in package mboost, namely "contr.dummy". This contrast is used per default in brandomc. It leads to a dummy coding as returned by model.matrix(~ x - 1) were the intercept is implicitly included but each factor level gets a separate effect estimate (for more details see brandom).

Details

The base-learners bbsc, bolsc and brandomc are the base-learners bbs, bols and brandom with additional identifiability constraints. The constraints enforce that $\sum_i \hat{h}(x_i, t) = 0$ for all t, so that effects varying over t can be interpreted as deviations from the global functional intercept, see Web Appendix A of Scheipl et al. (2015). The constraint is enforced by a basis transformation of the design and penalty matrix. In particular, it is sufficient to apply the constraint on the covariate-part of the design and penalty matrix and thus, it is not necessary to change the basis in t-direction. See Appendix A of Brockhaus et al. (2015) for technical details on how to enforce this sum-to-zero constraint.

Cannot deal with any missing values in the covariates.

Value

Equally to the base-learners of package mboost:

An object of class blg (base-learner generator) with a dpp function (data pre-processing) and other functions.

The call to dpp returns an object of class bl (base-learner) with a fit function. The call to fit finally returns an object of class bm (base-model).

Author(s)

Sarah Brockhaus, Almond Stoecker

References

Brockhaus, S., Scheipl, F., Hothorn, T. and Greven, S. (2015): The functional linear array model. Statistical Modelling, 15(3), 279-300.

Scheipl, F., Staicu, A.-M. and Greven, S. (2015): Functional Additive Mixed Models, Journal of Computational and Graphical Statistics, 24(2), 477-501.

See Also

FDboost for the model fit. bbs, bols and brandom for the corresponding base-learners in mboost.

Examples

```
#### simulate data with functional response and scalar covariate (functional ANOVA)
n <- 60   ## number of cases
Gy <- 27  ## number of observation poionts per response curve
```

```
dat <- list()
dat$t <- (1:Gy-1)^2/(Gy-1)^2
set.seed(123)
dat$z1 <- rep(c(-1, 1), length = n)
dat$z1_fac <- factor(dat$z1, levels = c(-1, 1), labels = c("1", "2"))
# dat$z1 <- runif(n)
# dat$z1 <- dat$z1 - mean(dat$z1)

# mean and standard deviation for the functional response
mut <- matrix(2*sin(pi*dat$t), ncol = Gy, nrow = n, byrow = TRUE) +
        outer(dat$z1, dat$t, function(z1, t) z1*cos(pi*t) ) # true linear predictor
sigma <- 0.1

# draw respone y_i(t) ~ N(mu_i(t), sigma)
dat$y <- apply(mut, 2, function(x) rnorm(mean = x, sd = sigma, n = n))

## fit function-on-scalar model with a linear effect of z1
m1 <- FDboost(y ~ 1 + bolsc(z1_fac, df = 1), timeformula = ~ bbs(t, df = 6), data = dat)

# look for optimal mSTOP using cvrisk() or validateFDboost()

cvm <- cvrisk(m1, grid = 1:500)
m1[mstop(cvm)]

m1[200] # use 200 boosting iterations

# plot true and estimated coefficients
plot(dat$t, 2*sin(pi*dat$t), col = 2, type = "l", main = "intercept")
plot(m1, which = 1, lty = 2, add = TRUE)

plot(dat$t, 1*cos(pi*dat$t), col = 2, type = "l", main = "effect of z1")
lines(dat$t, -1*cos(pi*dat$t), col = 2, type = "l")
plot(m1, which = 2, lty = 2, col = 1, add = TRUE)
```

`bhistx` *Base-learners for Functional Covariates*

Description

Base-learners that fit historical functional effects that can be used with the tensor product, as, e.g., `hbistx(...) %X% bolsc(...)`, to form interaction effects (Ruegamer et al., 2018). For expert use only! May show unexpected behavior compared to other base-learners for functional data!

Usage

```
bhistx(
  x,
  limits = "s<=t",
```

```
  standard = c("no", "time", "length"),
  intFun = integrationWeightsLeft,
  inS = c("smooth", "linear", "constant"),
  inTime = c("smooth", "linear", "constant"),
  knots = 10,
  boundary.knots = NULL,
  degree = 3,
  differences = 1,
  df = 4,
  lambda = NULL,
  penalty = c("ps", "pss"),
  check.ident = FALSE
)
```

Arguments

x	object of type hmatrix containing time, index and functional covariate; note that timeLab in the hmatrix-object must be equal to the name of the time-variable in timeformula in the FDboost-call
limits	defaults to "s<=t" for an historical effect with s<=t; either one of "s<t" or "s<=t" for [l(t), u(t)] = [T1, t]; otherwise specify limits as a function for integration limits [l(t), u(t)]: function that takes s as the first and t as the second argument and returns TRUE for combinations of values (s,t) if s falls into the integration range for the given t.
standard	the historical effect can be standardized with a factor. "no" means no standardization, "time" standardizes with the current value of time and "lenght" standardizes with the lenght of the integral
intFun	specify the function that is used to compute integration weights in s over the functional covariate $x(s)$
inS	historical effect can be smooth, linear or constant in s, which is the index of the functional covariates x(s).
inTime	historical effect can be smooth, linear or constant in time, which is the index of the functional response y(time).
knots	either the number of knots or a vector of the positions of the interior knots (for more details see bbs).
boundary.knots	boundary points at which to anchor the B-spline basis (default the range of the data). A vector (of length 2) for the lower and the upper boundary knot can be specified.
degree	degree of the regression spline.
differences	a non-negative integer, typically 1, 2 or 3. Defaults to 1. If differences = *k*, *k*-th-order differences are used as a penalty (*0*-th order differences specify a ridge penalty).
df	trace of the hat matrix for the base-learner defining the base-learner complexity. Low values of df correspond to a large amount of smoothing and thus to "weaker" base-learners.
lambda	smoothing parameter of the penalty, computed from df when df is specified.

penalty	by default, penalty="ps", the difference penalty for P-splines is used, for penalty="pss" the penalty matrix is transformed to have full rank, so called shrinkage approach by Marra and Wood (2011)
check.ident	use checks for identifiability of the effect, based on Scheipl and Greven (2016); see Brockhaus et al. (2017) for identifiability checks that take into account the integration limits

Details

bhistx implements a base-learner for functional covariates with flexible integration limits l(t), r(t) and the possibility to standardize the effect by 1/t or the length of the integration interval. The effect is stand * int_{l(t)}^{r_{t}} x(s)beta(t,s) ds. The base-learner defaults to a historical effect of the form $\int_{T1}^{t} x_i(s)beta(t,s)ds$, where $T1$ is the minimal index of t of the response $Y(t)$. bhistx can only be used if $Y(t)$ and $x(s)$ are observd over the same domain $s, t \in [T1, T2]$. The base-learner bhistx can be used to set up complex interaction effects like factor-specific historical effects as discussed in Ruegamer et al. (2018).

Note that the data has to be supplied as a hmatrix object for model fit and predictions.

Value

Equally to the base-learners of package mboost:

An object of class blg (base-learner generator) with a dpp function (dpp, data pre-processing).

The call of dpp returns an object of class bl (base-learner) with a fit function. The call to fit finally returns an object of class bm (base-model).

References

Brockhaus, S., Melcher, M., Leisch, F. and Greven, S. (2017): Boosting flexible functional regression models with a high number of functional historical effects, Statistics and Computing, 27(4), 913-926.

Marra, G. and Wood, S.N. (2011): Practical variable selection for generalized additive models. Computational Statistics & Data Analysis, 55, 2372-2387.

Ruegamer D., Brockhaus, S., Gentsch K., Scherer, K., Greven, S. (2018). Boosting factor-specific functional historical models for the detection of synchronization in bioelectrical signals. Journal of the Royal Statistical Society: Series C (Applied Statistics), 67, 621-642.

Scheipl, F., Staicu, A.-M. and Greven, S. (2015): Functional Additive Mixed Models, Journal of Computational and Graphical Statistics, 24(2), 477-501. https://arxiv.org/abs/1207.5947

Scheipl, F. and Greven, S. (2016): Identifiability in penalized function-on-function regression models. Electronic Journal of Statistics, 10(1), 495-526.

See Also

FDboost for the model fit and bhist for simple hisotorical effects.

Examples

```
if(require(refund)){
## simulate some data from a historical model
## the interaction effect is in this case not necessary
n <- 100
nygrid <- 35
data1 <- pffrSim(scenario = c("int", "ff"), limits = function(s,t){ s <= t },
                 n = n, nygrid = nygrid)
data1$X1 <- scale(data1$X1, scale = FALSE) ## center functional covariate
dataList <- as.list(data1)
dataList$tvals <- attr(data1, "yindex")

## create the hmatrix-object
X1h <- with(dataList, hmatrix(time = rep(tvals, each = n), id = rep(1:n, nygrid),
                              x = X1, argvals = attr(data1, "xindex"),
                              timeLab = "tvals", idLab = "wideIndex",
                              xLab = "myX", argvalsLab = "svals"))
dataList$X1h <- I(X1h)
dataList$svals <- attr(data1, "xindex")
## add a factor variable
dataList$zlong <- factor(gl(n = 2, k = n/2, length = n*nygrid), levels = 1:2)
dataList$z <- factor(gl(n = 2, k = n/2, length = n), levels = 1:2)

## do the model fit with main effect of bhistx() and interaction of bhistx() and bolsc()
mod <- FDboost(Y ~ 1 + bhistx(x = X1h, df = 5, knots = 5) +
               bhistx(x = X1h, df = 5, knots = 5) %X% bolsc(zlong),
              timeformula = ~ bbs(tvals, knots = 10), data = dataList)

## alternative parameterization: interaction of bhistx() and bols()
mod <- FDboost(Y ~ 1 + bhistx(x = X1h, df = 5, knots = 5) %X% bols(zlong),
              timeformula = ~ bbs(tvals, knots = 10), data = dataList)


  # find the optimal mstop over 5-fold bootstrap (small example to reduce run time)
  cv <- cvrisk(mod, folds = cv(model.weights(mod), B = 5))
  mstop(cv)
  mod[mstop(cv)]

 appl1 <- applyFolds(mod, folds = cv(rep(1, length(unique(mod$id))), type = "bootstrap", B = 5))

 # plot(mod)

}
```

`birthDistribution` *Densities of live births in Germany*

Description

`birthDistribution` contains densities of live births in Germany over the months per year (1950 to 2019) and sex (male and female), resulting in 140 densities.

Usage

```
data(birthDistribution, package = "FDboost")
```

Format

A list in the correct format to be passed to `FDboost` for density-on-scalar regression:

`birth_densities` A 140 x 12 matrix containing the birth densities in its rows. The first 70 rows correspond to male newborns, the second 70 rows to female ones. Within both of these, the years are ordered increasingly (1950-2019), see also `sex` and `year`.

`birth_densities_clr` A 140 x 12 matrix containing the clr transformed densities in its rows. Same structure as `birth_densities`.

`sex` A factor vector of length 140 with levels `"m"` (male) and `"f"` (female), corresponding to the sex of the newborns for the rows of `birth_densities` and `birth_densities_clr`. The first 70 elements are `"m"`, the second 70 `"f"`.

`year` A vector of length 140 containing the integers from 1950 to 2019 two times (`c(1950:2019, 1950:2019)`), corresponding to the years for the rows of `birth_densities` and `birth_densities_clr`.

`month` A vector containing the integers from 1 to 12, corresponding to the months for the columns of `birth_densities` and `birth_densities_clr` (domain $\mathcal{T}$ of the (clr-)densities).

Note that for estimating a density-on-scalar model with `FDboost`, the clr transformed densities (`birth_densities_clr`) serve as response, see also the vignette "FDboost_density-on-scalar_births". The original densities (`birth_densities`) are not needed for estimation, but still included for the sake of completeness.

Details

To compensate for the different lengths of the months, the average number of births per day for each month (by sex and year) was used to compute the birth shares from the absolute birth counts. The 12 shares corresponding to one year and sex form one density in the Bayes Hilbert space $B^2(\delta) = B^2(\mathcal{T}, \mathcal{A}, \delta)$, where $\mathcal{T} = \{1, \ldots, 12\}$ corresponds to the set of the 12 months, $\mathcal{A} := \mathcal{P}(\mathcal{T})$ corresponds to the power set of $\mathcal{T}$, and the reference measure $\delta := \sum_{t=1}^{12} \delta_t$ corresponds to the sum of dirac measures at $t \in \mathcal{T}$.

Source

Statistisches Bundesamt (Destatis), Genesis-Online, data set 12612-0002 (01/18/2021); dl-de/by-2-0; processed by Eva-Maria Maier

References

Maier, E.-M., Stoecker, A., Fitzenberger, B., Greven, S. (2021): Additive Density-on-Scalar Regression in Bayes Hilbert Spaces with an Application to Gender Economics. arXiv preprint arXiv:2110.11771.

See Also

clr for the (inverse) clr transformation.

Examples

```
data("birthDistribution", package = "FDboost")

# Plot densities
year_col <- rainbow(70, start = 0.5, end = 1)
year_lty <- c(1, 2, 4, 5)
oldpar <- par(mfrow = c(1, 2))
funplot(1:12, birthDistribution$birth_densities[1:70, ], ylab = "densities", xlab = "month",
        xaxp = c(1, 12, 11), pch = 20, col = year_col, lty = year_lty, main = "Male")
funplot(1:12, birthDistribution$birth_densities[71:140, ], ylab = "densities", xlab = "month",
        xaxp = c(1, 12, 11), pch = 20, col = year_col, lty = year_lty, main = "Female")
par(mfrow = c(1, 1))

# fit density-on-scalar model with effects for sex and year
model <- FDboost(birth_densities_clr ~ 1 + bolsc(sex, df = 1) +
                   bbsc(year, df = 1, differences = 1),
                 # use bbsc() in timeformula to ensure integrate-to-zero constraint
                 timeformula = ~bbsc(month, df = 4,
                                     # December is followed by January of subsequent year
                                      cyclic = TRUE,
                                     # knots = {1, ..., 12} with additional boundary knot
                                      # 0 (coinciding with 12) due to cyclic = TRUE
                                      knots = 1:11, boundary.knots = c(0, 12),
                                     # degree = 1 with these knots yields identity matrix
                                      # as design matrix
                                      degree = 1),
                 data = birthDistribution, offset = 0,
                 control = boost_control(mstop = 1000))

# Plotting 'model' yields the clr-transformed effects
par(mfrow = c(1, 3))
plot(model, n1 = 12, n2 = 12)

# Use inverse clr transformation to get effects in Bayes Hilbert space, e.g. for intercept
intercept_clr <- predict(model, which = 1)[1, ]
intercept <- clr(intercept_clr, w = 1, inverse = TRUE)
funplot(1:12, intercept, xlab = "month", xaxp = c(1, 12, 11), pch = 20,
        main = "Intercept", ylab = expression(hat(beta)[0]), id = rep(1, 12))

# Same with predictions
predictions_clr <- predict(model)
predictions <- t(apply(predictions_clr, 1, clr, inverse = TRUE))
pred_ylim <- range(birthDistribution$birth_densities)
par(mfrow = c(1, 2))
funplot(1:12, predictions[1:70, ], ylab = "predictions", xlab = "month", ylim = pred_ylim,
        xaxp = c(1, 12, 11), pch = 20, col = year_col, lty = year_lty, main = "Male")
funplot(1:12, predictions[71:140, ], ylab = "predictions", xlab = "month", ylim = pred_ylim,
        xaxp = c(1, 12, 11), pch = 20, col = year_col, lty = year_lty, main = "Female")
```

```
par(oldpar)
```

`bootstrapCI` *Function to compute bootstrap confidence intervals*

Description

The model is fitted on bootstrapped samples of the data to compute bootstrapped coefficient estimates. To determine the optimal stopping iteration an inner bootstrap is run within each bootstrap fold. As estimation by boosting shrinks the coefficient estimates towards zero, to bootstrap confidence intervals are biased towards zero.

Usage

```
bootstrapCI(
  object,
  which = NULL,
  resampling_fun_outer = NULL,
  resampling_fun_inner = NULL,
  B_outer = 100,
  B_inner = 25,
  type_inner = c("bootstrap", "kfold", "subsampling"),
  levels = c(0.05, 0.95),
  verbose = TRUE,
  ...
)
```

Arguments

`object`	a fitted model object of class `FDboost`, for which the confidence intervals should be computed.
`which`	a subset of base-learners to take into account for computing confidence intervals.
`resampling_fun_outer`	function for the outer resampling procedure. `resampling_fun_outer` must be a function with arguments `object` and `fun`, where `object` corresponds to the fitted `FDboost` object and `fun` is passed to the `fun` argument of the resampling function (see examples). If `NULL`, `applyFolds` is used with 100-fold boostrap. Further arguments to `applyFolds` can be passed via `...`. Although the function can be defined very flexible, it is recommended to use `applyFolds` and, in particular, not `cvrisk`, as in this case, weights of the inner and outer fold will interact, probably causing the inner resampling to crash. For bootstrapped confidence intervals the outer function should usually be a bootstrap type of resampling.
`resampling_fun_inner`	function for the inner resampling procudure, which determines the optimal stopping iteration in each fold of the outer resampling procedure. Should be a function with one argument `object` for the fitted `FDboost` object. If `NULL`, `cvrisk` is used with 25-fold bootstrap.

B_outer	Number of resampling folds in the outer loop. Argument is overwritten, when a custom resampling_fun_outer is supplied.
B_inner	Number of resampling folds in the inner loop. Argument is overwritten, when a custom resampling_fun_inner is supplied.
type_inner	character argument for specifying the cross-validation method for the inner resampling level. Default is "bootstrap". Currently bootstrap, k-fold cross-validation and subsampling are implemented.
levels	the confidence levels required. If NULL, the raw results are returned.
verbose	if TRUE, information will be printed in the console
...	further arguments passed to applyFolds if the default for resampling_fun_outer is used

Value

A list containing the elements raw_results, the quantiles and mstops. In raw_results and quantiles, each baselearner selected with which in turn corresponds to a list element. The quantiles are given as vector, matrix or list of matrices depending on the nature of the effect. In case of functional effects the list element inquantiles is a length(levels) times length(effect) matrix, i.e. the rows correspond to the quantiles. In case of coefficient surfaces, quantiles comprises a list of matrices, where each list element corresponds to a quantile.

Note

Note that parallelization can be achieved by defining the resampling_fun_outer or _inner accordingly. See, e.g., cvrisk on how to parallelize resampling functions or the examples below. Also note that by defining a custum inner or outer resampling function the respective argument B_inner or B_outer is ignored. For models with complex baselearners, e.g., created by combining several baselearners with the Kronecker or row-wise tensor product, it is also recommended to use levels = NULL in order to let the function return the raw results and then manually compute confidence intervals. If a baselearner is not selected in any fold, the function treats its effect as constantly zero.

Author(s)

David Ruegamer, Sarah Brockhaus

Examples

```
if(require(refund)){
#########
# model with linear functional effect, use bsignal()
# Y(t) = f(t) + \int X1(s)\beta(s,t)ds + eps
set.seed(2121)
data1 <- pffrSim(scenario = "ff", n = 40)
data1$X1 <- scale(data1$X1, scale = FALSE)
dat_list <- as.list(data1)
dat_list$t <- attr(data1, "yindex")
dat_list$s <- attr(data1, "xindex")
```

```
## model fit by FDboost
m1 <- FDboost(Y ~ 1 + bsignal(x = X1, s = s, knots = 8, df = 3),
              timeformula = ~ bbs(t, knots = 8), data = dat_list)

}


# a short toy example with to few folds
# and up to 200 boosting iterations
bootCIs <- bootstrapCI(m1[200], B_inner = 2, B_outer = 5)

# look at stopping iterations
bootCIs$mstops

# plot bootstrapped coefficient estimates
plot(bootCIs, ask = FALSE)


my_inner_fun <- function(object){
cvrisk(object, folds = cvLong(id = object$id, weights =
model.weights(object), B = 2) # 10-fold for inner resampling
)
}


bootCIs <- bootstrapCI(m1, resampling_fun_inner = my_inner_fun,
                       B_outer = 5) # small B_outer to speed up


## We can also use the ... argument to parallelize the applyFolds
## function in the outer resampling


bootCIs <- bootstrapCI(m1, B_inner = 5, B_outer = 3)


## Now let's parallelize the outer resampling and use
## crossvalidation instead of bootstrap for the inner resampling

my_inner_fun <- function(object){
cvrisk(object, folds = cvLong(id = object$id, weights =
model.weights(object), type = "kfold", # use CV
B = 5, # 5-fold for inner resampling
)) # use five cores
}

# use applyFolds for outer function to avoid messing up weights
my_outer_fun <- function(object, fun){
applyFolds(object = object,
folds = cv(rep(1, length(unique(object$id)))),
type = "bootstrap", B = 10), fun = fun) # parallelize on 10 cores
}
```

```
bootCIs <- bootstrapCI(m1, resampling_fun_inner = my_inner_fun,
                       resampling_fun_outer = my_outer_fun,
                       B_inner = 5, B_outer = 10)


######## Example for scalar-on-function-regression with bsignal()
data("fuelSubset", package = "FDboost")

## center the functional covariates per observed wavelength
fuelSubset$UVVIS <- scale(fuelSubset$UVVIS, scale = FALSE)
fuelSubset$NIR <- scale(fuelSubset$NIR, scale = FALSE)

## to make mboost:::df2lambda() happy (all design matrix entries < 10)
## reduce range of argvals to [0,1] to get smaller integration weights
fuelSubset$uvvis.lambda <- with(fuelSubset, (uvvis.lambda - min(uvvis.lambda)) /
(max(uvvis.lambda) - min(uvvis.lambda) ))
fuelSubset$nir.lambda <- with(fuelSubset, (nir.lambda - min(nir.lambda)) /
(max(nir.lambda) - min(nir.lambda) ))

## model fit with scalar response and two functional linear effects
## include no intercept as all base-learners are centered around 0

mod2 <- FDboost(heatan ~ bsignal(UVVIS, uvvis.lambda, knots = 40, df = 4, check.ident = FALSE)
               + bsignal(NIR, nir.lambda, knots = 40, df=4, check.ident = FALSE),
               timeformula = NULL, data = fuelSubset)



# takes some time, because of defaults: B_outer = 100, B_inner = 25
bootCIs <- bootstrapCI(mod2, B_outer = 10, B_inner = 5)
           # in practice, rather set B_outer = 1000
```

---

bsignal *Base-learners for Functional Covariates*

Description

Base-learners that fit effects of functional covariates.

Usage

```
bsignal(
  x,
  s,
  index = NULL,
  inS = c("smooth", "linear", "constant"),
```

```
  knots = 10,
  boundary.knots = NULL,
  degree = 3,
  differences = 1,
  df = 4,
  lambda = NULL,
  center = FALSE,
  cyclic = FALSE,
  Z = NULL,
  penalty = c("ps", "pss"),
  check.ident = FALSE
)

bconcurrent(
  x,
  s,
  time,
  index = NULL,
  knots = 10,
  boundary.knots = NULL,
  degree = 3,
  differences = 1,
  df = 4,
  lambda = NULL,
  cyclic = FALSE
)

bhist(
  x,
  s,
  time,
  index = NULL,
  limits = "s<=t",
  standard = c("no", "time", "length"),
  intFun = integrationWeightsLeft,
  inS = c("smooth", "linear", "constant"),
  inTime = c("smooth", "linear", "constant"),
  knots = 10,
  boundary.knots = NULL,
  degree = 3,
  differences = 1,
  df = 4,
  lambda = NULL,
  penalty = c("ps", "pss"),
  check.ident = FALSE
)

bfpc(
```

```
  x,
  s,
  index = NULL,
  df = 4,
  lambda = NULL,
  penalty = c("identity", "inverse", "no"),
  pve = 0.99,
  npc = NULL,
  npc.max = 15,
  getEigen = TRUE
)
```

Arguments

x	matrix of functional variable x(s). The functional covariate has to be supplied as n by <no. of evaluations> matrix, i.e., each row is one functional observation.
s	vector for the index of the functional variable x(s) giving the measurement points of the functional covariate.
index	a vector of integers for expanding the covariate in x For example, `bsignal(X, s, index = index)` is equal to `bsignal(X[index,], s)`, where index is an integer of length greater or equal to `NROW(x)`.
inS	the functional effect can be smooth, linear or constant in s, which is the index of the functional covariates x(s).
knots	either the number of knots or a vector of the positions of the interior knots (for more details see bbs).
boundary.knots	boundary points at which to anchor the B-spline basis (default the range of the data). A vector (of length 2) for the lower and the upper boundary knot can be specified.
degree	degree of the regression spline.
differences	a non-negative integer, typically 1, 2 or 3. Defaults to 1. If `differences` = *k*, *k*-th-order differences are used as a penalty (*0*-th order differences specify a ridge penalty).
df	trace of the hat matrix for the base-learner defining the base-learner complexity. Low values of `df` correspond to a large amount of smoothing and thus to "weaker" base-learners.
lambda	smoothing parameter of the penalty, computed from `df` when `df` is specified.
center	See bbs. The effect is re-parameterized such that the unpenalized part of the fit is subtracted and only the penalized effect is fitted, using a spectral decomposition of the penalty matrix. The unpenalized, parametric part has then to be included in separate base-learners using `bsignal(..., inS = 'constant')` or `bsignal(..., inS = 'linear')` for first (`difference = 1`) and second (`difference = 2`) order difference penalty respectively. See the help on the argument `center` of bbs.
cyclic	if `cyclic = TRUE` the fitted coefficient function coincides at the boundaries (useful for cyclic covariates such as day time etc.).

Z	a transformation matrix for the design-matrix over the index of the covariate. Z can be calculated as the transformation matrix for a sum-to-zero constraint in the case that all trajectories have the same mean (then a shift in the coefficient function is not identifiable).
penalty	for bsignal, by default, penalty = "ps", the difference penalty for P-splines is used, for penalty = "pss" the penalty matrix is transformed to have full rank, so called shrinkage approach by Marra and Wood (2011). For bfpc the penalty can be either "identity" for a ridge penalty (the default) or "inverse" to use the matrix with the inverse eigenvalues on the diagonal as penalty matrix or "no" for no penalty.
check.ident	use checks for identifiability of the effect, based on Scheipl and Greven (2016) for linear functional effect using bsignal and based on Brockhaus et al. (2017) for historical effects using bhist
time	vector for the index of the functional response y(time) giving the measurement points of the functional response.
limits	defaults to "s<=t" for an historical effect with s<=t; either one of "s<t" or "s<=t" for [l(t), u(t)] = [T1, t]; otherwise specify limits as a function for integration limits [l(t), u(t)]: function that takes s as the first and t as the second argument and returns TRUE for combinations of values (s,t) if s falls into the integration range for the given t.
standard	the historical effect can be standardized with a factor. "no" means no standardization, "time" standardizes with the current value of time and "length" standardizes with the length of the integral
intFun	specify the function that is used to compute integration weights in s over the functional covariate $x(s)$
inTime	the historical effect can be smooth, linear or constant in time, which is the index of the functional response y(time).
pve	proportion of variance explained by the first K functional principal components (FPCs): used to choose the number of functional principal components (FPCs).
npc	prespecified value for the number K of FPCs (if given, this overrides pve).
npc.max	maximal number K of FPCs to use; defaults to 15.
getEigen	save the eigenvalues and eigenvectors, defaults to TRUE.

Details

bsignal() implements a base-learner for functional covariates to estimate an effect of the form $\int x_i(s)\beta(s)ds$. Defaults to a cubic B-spline basis with first difference penalties for $\beta(s)$ and numerical integration over the entire range by using trapezoidal Riemann weights. If bsignal() is used within FDboost(), the base-learner of timeformula is attached, resulting in an effect varying over the index of the response $\int x_i(s)\beta(s,t)ds$ if timeformula = bbs(t). The functional variable must be observed on one common grid s.

bconcurrent() implements a concurrent effect for a functional covariate on a functional response, i.e., an effect of the form $x_i(t)\beta(t)$ for a functional response $Y_i(t)$ and concurrently observed covariate $x_i(t)$. bconcurrent() can only be used if $Y(t)$ and $x(s)$ are observed over the same domain $s, t \in [T1, T2]$.

`bhist()` implements a base-learner for functional covariates with flexible integration limits `l(t)`, `r(t)` and the possibility to standardize the effect by `1/t` or the length of the integration interval. The effect is $stand * \int_{l(t)}^{r_t} x(s)\beta(t,s)ds$, where $stand$ is the chosen standardization which defaults to 1. The base-learner defaults to a historical effect of the form $\int_{T1}^{t} x_i(s)\beta(t,s)ds$, where $T1$ is the minimal index of t of the response $Y(t)$. The functional covariate must be observed on one common grid `s`. See Brockhaus et al. (2017) for details on historical effects.

`bfpc()` is a base-learner for a linear effect of functional covariates based on functional principal component analysis (FPCA). For the functional linear effect $\int x_i(s)\beta(s)ds$ the functional covariate and the coefficient function are both represented by a FPC basis. The functional covariate $x(s)$ is decomposed into $x(s) \approx \sum_{k=1}^{K} \xi_{ik}\Phi_k(s)$ using `fpca.sc` for the truncated Karhunen-Loeve decomposition. Then $\beta(s)$ is represented in the function space spanned by $\Phi_k(s)$, k=1,...,K, see Scheipl et al. (2015) for details. As penalty matrix, the identity matrix is used. The implementation is similar to `ffpc`.

It is recommended to use centered functional covariates with $\sum_i x_i(s) = 0$ for all s in `bsignal()`-, `bhist()`- and `bconcurrent()`-terms. For centered covariates, the effects are centered per time-point of the response. If all effects are centered, the functional intercept can be interpreted as the global mean function.

The base-learners for functional covariates cannot deal with any missing values in the covariates.

Value

Equally to the base-learners of package `mboost`:

An object of class `blg` (base-learner generator) with a `dpp()` function (dpp, data pre-processing).

The call of `dpp()` returns an object of class `bl` (base-learner) with a `fit()` function. The call to `fit()` finally returns an object of class `bm` (base-model).

References

Brockhaus, S., Scheipl, F., Hothorn, T. and Greven, S. (2015): The functional linear array model. Statistical Modelling, 15(3), 279-300.

Brockhaus, S., Melcher, M., Leisch, F. and Greven, S. (2017): Boosting flexible functional regression models with a high number of functional historical effects, Statistics and Computing, 27(4), 913-926.

Marra, G. and Wood, S.N. (2011): Practical variable selection for generalized additive models. Computational Statistics & Data Analysis, 55, 2372-2387.

Scheipl, F., Staicu, A.-M. and Greven, S. (2015): Functional Additive Mixed Models, Journal of Computational and Graphical Statistics, 24(2), 477-501.

Scheipl, F. and Greven, S. (2016): Identifiability in penalized function-on-function regression models. Electronic Journal of Statistics, 10(1), 495-526.

See Also

`FDboost` for the model fit.

Examples

```
######## Example for scalar-on-function-regression with bsignal()
data("fuelSubset", package = "FDboost")

## center the functional covariates per observed wavelength
fuelSubset$UVVIS <- scale(fuelSubset$UVVIS, scale = FALSE)
fuelSubset$NIR <- scale(fuelSubset$NIR, scale = FALSE)

## to make mboost:::df2lambda() happy (all design matrix entries < 10)
## reduce range of argvals to [0,1] to get smaller integration weights
fuelSubset$uvvis.lambda <- with(fuelSubset, (uvvis.lambda - min(uvvis.lambda)) /
                                  (max(uvvis.lambda) - min(uvvis.lambda) ))
fuelSubset$nir.lambda <- with(fuelSubset, (nir.lambda - min(nir.lambda)) /
                                (max(nir.lambda) - min(nir.lambda) ))

## model fit with scalar response and two functional linear effects
## include no intercept
## as all base-learners are centered around 0
mod2 <- FDboost(heatan ~ bsignal(UVVIS, uvvis.lambda, knots = 40, df = 4, check.ident = FALSE)
               + bsignal(NIR, nir.lambda, knots = 40, df=4, check.ident = FALSE),
               timeformula = NULL, data = fuelSubset)
summary(mod2)


##############################################
### data simulation like in manual of pffr::ff

if(require(refund)){

#########
# model with linear functional effect, use bsignal()
# Y(t) = f(t) + \int X1(s)\beta(s,t)ds + eps
set.seed(2121)
data1 <- pffrSim(scenario = "ff", n = 40)
data1$X1 <- scale(data1$X1, scale = FALSE)
dat_list <- as.list(data1)
dat_list$t <- attr(data1, "yindex")
dat_list$s <- attr(data1, "xindex")

## model fit by FDboost
m1 <- FDboost(Y ~ 1 + bsignal(x = X1, s = s, knots = 5),
              timeformula = ~ bbs(t, knots = 5), data = dat_list,
              control = boost_control(mstop = 21))

## search optimal mSTOP

  set.seed(123)
  cv <- validateFDboost(m1, grid = 1:100) # 21 iterations


## model fit by pffr
t <- attr(data1, "yindex")
```

```
s <- attr(data1, "xindex")
m1_pffr <- pffr(Y ~ ff(X1, xind = s), yind = t, data = data1)


  oldpar <- par(mfrow = c(2, 2))
  plot(m1, which = 1); plot(m1, which = 2)
  plot(m1_pffr, select = 1, shift = m1_pffr$coefficients["(Intercept)"])
  plot(m1_pffr, select = 2)
  par(oldpar)



##############################################
# model with functional historical effect, use bhist()
# Y(t) = f(t)  + \int_0^t X1(s)\beta(s,t)ds + eps
set.seed(2121)
mylimits <- function(s, t){
  (s < t) | (s == t)
}
data2 <- pffrSim(scenario = "ff", n = 40, limits = mylimits)
data2$X1 <- scale(data2$X1, scale = FALSE)
dat2_list <- as.list(data2)
dat2_list$t <- attr(data2, "yindex")
dat2_list$s <- attr(data2, "xindex")

## model fit by FDboost
m2 <- FDboost(Y ~ 1 + bhist(x = X1, s = s, time = t, knots = 5),
              timeformula = ~ bbs(t, knots = 5), data = dat2_list,
              control = boost_control(mstop = 40))

## search optimal mSTOP

  set.seed(123)
  cv2 <- validateFDboost(m2, grid = 1:100) # 40 iterations


## model fit by pffr
t <- attr(data2, "yindex")
s <- attr(data2, "xindex")
m2_pffr <- pffr(Y ~ ff(X1, xind = s, limits = "s<=t"), yind = t, data = data2)


oldpar <- par(mfrow = c(2, 2))
plot(m2, which = 1); plot(m2, which = 2)
## plot of smooth intercept does not contain m1_pffr$coefficients["(Intercept)"]
plot(m2_pffr, select = 1, shift = m2_pffr$coefficients["(Intercept)"])
plot(m2_pffr, select = 2)
par(oldpar)



}
```

clr *Clr and inverse clr transformation*

Description

`clr` computes the clr or inverse clr transformation of a vector `f` with respect to integration weights `w`, corresponding to a Bayes Hilbert space $B^2(\mu) = B^2(\mathcal{T}, \mathcal{A}, \mu)$.

Usage

```
clr(f, w = 1, inverse = FALSE)
```

Arguments

`f`	a vector containing the function values (evaluated on a grid) of the function f to transform. If `inverse = TRUE`, `f` must be a density, i.e., all entries must be positive and usually `f` integrates to one. If `inverse = FALSE`, `f` should integrate to zero, see Details.
`w`	a vector of length one or of the same length as `f` containing positive integration weights. If `w` has length one, this weight is used for all function values. The integral of f is approximated via $\int_{\mathcal{T}} f \,\mathrm{d}\mu \approx \sum_{j=1}^{m} \mathtt{w}_j \, \mathtt{f}_j$, where m equals the length of `f`.
`inverse`	if `TRUE`, the inverse clr transformation is computed.

Details

The clr transformation maps a density f from $B^2(\mu)$ to $L_0^2(\mu) := \{f \in L^2(\mu) \mid \int_{\mathcal{T}} f \,\mathrm{d}\mu = 0\}$ via

$$\mathrm{clr}(f) := \log f - \frac{1}{\mu(\mathcal{T})} \int_{\mathcal{T}} \log f \,\mathrm{d}\mu.$$

The inverse clr transformation maps a function f from $L_0^2(\mu)$ to $B^2(\mu)$ via

$$\mathrm{clr}^{-1}(f) := \frac{\exp f}{\int_{\mathcal{T}} \exp f \,\mathrm{d}\mu}.$$

Note that in contrast to Maier et al. (2021), this definition of the inverse clr transformation includes normalization, yielding the respective probability density function (representative of the equivalence class of proportional functions in $B^2(\mu)$).

The (inverse) clr transformation depends not only on f, but also on the underlying measure space $(\mathcal{T}, \mathcal{A}, \mu)$, which determines the integral. In `clr` this is specified via the integration weights `w`. E.g., for a discrete set $\mathcal{T}$ with $\mathcal{A} = \mathcal{P}(\mathcal{T})$ the power set of $\mathcal{T}$ and $\mu = \sum_{t \in T} \delta_t$ the sum of dirac measures at $t \in \mathcal{T}$, the default `w = 1` is the correct choice. In this case, integrals are indeed computed exactly, not only approximately. For an interval $\mathcal{T} = [a, b]$ with $\mathcal{A} = \mathcal{B}$ the Borel σ-algebra restricted to $\mathcal{T}$ and $\mu = \lambda$ the Lebesgue measure, the choice of `w` depends on the grid on which the function was

evaluated: w_j must correspond to the length of the subinterval of $[a, b]$, which f_j represents. E.g., for a grid with equidistant distance d, where the boundary grid values are $a + \frac{d}{2}$ and $b - \frac{d}{2}$ (i.e., the grid points are centers of intervals of size d), equal weights d should be chosen for w.

The clr transformation is crucial for density-on-scalar regression since estimating the clr transformed model in $L_0^2(\mu)$ is equivalent to estimating the original model in $B^2(\mu)$ (as the clr transformation is an isometric isomorphism), see also the vignette "FDboost_density-on-scalar_births" and Maier et al. (2021).

Value

A vector of the same length as f containing the (inverse) clr transformation of f.

Author(s)

Eva-Maria Maier

References

Maier, E.-M., Stoecker, A., Fitzenberger, B., Greven, S. (2021): Additive Density-on-Scalar Regression in Bayes Hilbert Spaces with an Application to Gender Economics. arXiv preprint arXiv:2110.11771.

Examples

```
### Continuous case (T = [0, 1] with Lebesgue measure):
# evaluate density of a Beta distribution on an equidistant grid
g <- seq(from = 0.005, to = 0.995, by = 0.01)
f <- dbeta(g, 2, 5)
# compute clr transformation with distance of two grid points as integration weight
f_clr <- clr(f, w = 0.01)
# visualize result
plot(g, f_clr , type = "l")
abline(h = 0, col = "grey")
# compute inverse clr transformation (w as above)
f_clr_inv <- clr(f_clr, w = 0.01, inverse = TRUE)
# visualize result
plot(g, f, type = "l")
lines(g, f_clr_inv, lty = 2, col = "red")

### Discrete case (T = {1, ..., 12} with sum of dirac measures at t in T):
data("birthDistribution", package = "FDboost")
# fit density-on-scalar model with effects for sex and year
model <- FDboost(birth_densities_clr ~ 1 + bolsc(sex, df = 1) +
                   bbsc(year, df = 1, differences = 1),
                 # use bbsc() in timeformula to ensure integrate-to-zero constraint
                 timeformula = ~bbsc(month, df = 4,
                                     # December is followed by January of subsequent year
                                      cyclic = TRUE,
                                     # knots = {1, ..., 12} with additional boundary knot
                                      # 0 (coinciding with 12) due to cyclic = TRUE
                                      knots = 1:11, boundary.knots = c(0, 12),
                                     # degree = 1 with these knots yields identity matrix
```

```
                                    # as design matrix
                                    degree = 1),
                  data = birthDistribution, offset = 0,
                  control = boost_control(mstop = 1000))
# Extract predictions (clr-transformed!) and transform them to Bayes Hilbert space
predictions_clr <- predict(model)
predictions <- t(apply(predictions_clr, 1, clr, inverse = TRUE))
```

`coef.FDboost` *Coefficients of boosted functional regression model*

Description

Takes a fitted FDboost-object produced by FDboost() and returns estimated coefficient functions/surfaces $\beta(t), \beta(s,t)$ and estimated smooth effects $f(z), f(x,z)$ or $f(x,z,t)$. Not implemented for smooths in more than 3 dimensions.

Usage

```
## S3 method for class 'FDboost'
coef(
  object,
  raw = FALSE,
  which = NULL,
  computeCoef = TRUE,
  returnData = FALSE,
  n1 = 40,
  n2 = 40,
  n3 = 20,
  n4 = 10,
  ...
)
```

Arguments

object	a fitted FDboost-object
raw	logical defaults to FALSE. If raw = FALSE for each effect the estimated function/surface is calculated. If raw = TRUE the coefficients of the model are returned.
which	a subset of base-learners for which the coefficients should be computed (numeric vector), defaults to NULL which is the same as which=seq_along(object$baselearner). In the special case of which=0, only the coefficients of the offset are returned.
computeCoef	defaults to TRUE, if FALSE only the names of the terms are returned
returnData	return the dataset which is used to get the coefficient estimates as predictions, see Details.

n1	see below
n2	see below
n3	n1, n2, n3 give the number of grid-points for 1-/2-/3-dimensional smooth terms used in the marginal equidistant grids over the range of the covariates at which the estimated effects are evaluated.
n4	gives the number of points for the third dimension in a 3-dimensional smooth term
...	other arguments, not used.

Details

If raw = FALSE the function coef.FDboost generates adequate dummy data and uses the function predict.FDboost to compute the estimated coefficient functions.

Value

If raw = FALSE, a list containing

- offset a list with plot information for the offset.
- smterms a named list with one entry for each smooth term in the model. Each entry contains
  - x, y, z the unique grid-points used to evaluate the smooth/coefficient function/coefficient surface
  - xlim, ylim, zlim the extent of the x/y/z-axes
  - xlab, ylab, zlab the names of the covariates for the x/y/z-axes
  - value a vector/matrix/list of matrices containing the coefficient values
  - dim the dimensionality of the effect
  - main the label of the smooth term (a short label)

If raw = TRUE, a list containing the estimated spline coefficients.

---

cvrisk.FDboostLSS *Cross-validation for FDboostLSS*

Description

Multidimensional cross-validated estimation of the empirical risk for hyper-parameter selection, for an object of class FDboostLSS setting the folds per default to resampling curves.

Usage

```
## S3 method for class 'FDboostLSS'
cvrisk(
  object,
  folds = cvLong(id = object[[1]]$id, weights = model.weights(object[[1]])),
  grid = NULL,
```

```
  papply = mclapply,
  trace = TRUE,
  fun = NULL,
  ...
)
```

Arguments

object	an object of class FDboostLSS.
folds	a weight matrix a weight matrix with number of rows equal to the number of observations. The number of columns corresponds to the number of cross-validation runs, defaults to 25 bootstrap samples, resampling whole curves
grid	defaults to a grid up to the current number of boosting iterations. The default generates the grid according to the defaults of cvrisk.mboostLSS which are different for models with cyclic or noncyclic fitting.
papply	(parallel) apply function, defaults to mclapply, see cvrisk.mboostLSS for details.
trace	print status information during cross-validation? Defaults to TRUE.
fun	if fun is NULL, the out-of-sample risk is returned. fun, as a function of object, may extract any other characteristic of the cross-validated models. These are returned as is.
...	additional arguments passed to mclapply.

Details

The function cvrisk.FDboostLSS is a wrapper for cvrisk.mboostLSS in package gamboostLSS. It overrides the default for the folds, so that the folds are sampled on the level of curves (not on the level of single observations, which does not make sense for functional response).

Value

An object of class cvriskLSS (when fun was not specified), basically a matrix containing estimates of the empirical risk for a varying number of bootstrap iterations. plot and print methods are available as well as an mstop method, see cvrisk.mboostLSS.

See Also

cvrisk.mboostLSS in package gamboostLSS.

`emotion` *EEG and EMG recordings in a computerised gambling study*

Description

To analyse the functional relationship between electroencephalography (EEG) and facial electromyography (EMG), Gentsch et al. (2014) simultaneously recorded EEG and EMG signals from 24 participants while they were playing a computerised gambling task. The given subset contains aggregated observations of 23 participants. Curves were averaged over each subject and each of the 8 study settings, resulting in 23 times 8 curves.

Usage

```
data("emotion")
```

Format

A list with the following 10 variables.

`power` factor variable with levels *high* and *low*

`game_outcome` factor variable with levels *gain* and *loss*

`control` factor variable with levels *high* and *low*

`subject` factor variable with 23 levels

`EEG` matrix; EEG signal in wide format

`EMG` matrix; EMG signal in wide format

`s` time points for the functional covariate

`t` time points for the functional response

Details

The aim is to explain potentials in the EMG signal by study settings as well as the EEG signal (see Ruegamer et al., 2018).

Source

Gentsch, K., Grandjean, D. and Scherer, K. R. (2014) Coherence explored between emotion components: Evidence from event-related potentials and facial electromyography. Biological Psychology, 98, 70-81.

Ruegamer D., Brockhaus, S., Gentsch K., Scherer, K., Greven, S. (2018). Boosting factor-specific functional historical models for the detection of synchronization in bioelectrical signals. Journal of the Royal Statistical Society: Series C (Applied Statistics), 67, 621-642.

Examples

```
data("emotion", package = "FDboost")

# fit function-on-scalar model with random effect and power effect
fos_random_power <- FDboost(EMG ~ 1 + brandomc(subject, df = 2)
                            + bolsc(power, df = 2),
                            timeformula = ~ bbs(t, df = 3),
                            data = emotion)
## Not run:
# fit function-on-function model with intercept and historical EEG effect
# where limits specifies the used lag between EMG and EEG signal
fof_historical <- FDboost(EMG ~ 1 + bhist(EEG, s = s, time = t,
                          limits = function(s,t) s < t - 3),
                          timeformula = ~ bbs(t, df = 3), data = emotion,
                          control = boost_control(mstop = 200))

## End(Not run)
```

`extract.blg` *Extract information of a base-learner*

Description

Takes a base-learner and extracts information.

Usage

```
## S3 method for class 'blg'
extract(
  object,
  what = c("design", "penalty", "index"),
  asmatrix = FALSE,
  expand = FALSE,
  ...
)
```

Arguments

`object`	a base-learner
`what`	a character specifying the quantities to extract. This can be a subset of "design" (default; design matrix), "penalty" (penalty matrix) and "index" (index of ties used to expand the design matrix)
`asmatrix`	a logical indicating whether the the returned matrix should be coerced to a matrix (default) or if the returned object stays as it is (i.e., potentially a sparse matrix). This option is only applicable if `extract` returns matrices, i.e., `what = "design"` or `what = "penalty"`.

expand	a logical indicating whether the design matrix should be expanded (default: FALSE). This is useful if ties were taken into account either manually (via argument index in a base-learner) or automatically for data sets with many observations. expand = TRUE is equivalent to extract(B)[extract(B, what = "index"),] for a base-learner B.
...	currently not used

See Also

extract for the extract function of the package mboost.

`factorize` *Factorize tensor product model*

Description

Factorize an FDboost tensor product model into the response and covariate parts

$$h_j(x,t) = \sum_k v_j^{(k)}(t) h_j^{(k)}(x), j = 1, ..., J,$$

for effect visualization as proposed in Stoecker, Steyer and Greven (2022).

Usage

```
factorize(x, ...)

## S3 method for class 'FDboost'
factorize(x, newdata = NULL, newweights = 1, blwise = TRUE, ...)
```

Arguments

x	a model object of class FDboost.
...	other arguments passed to methods.
newdata	new data the factorization is based on. By default (NULL), the factorization is carried out on the data used for fitting.
newweights	vector of the length of the data or length one, containing new weights used for factorization.
blwise	logical, should the factorization be carried out base-learner-wise (TRUE, default) or for the whole model simultaneously.

Details

The mboost infrastructure is used for handling the orthogonal response directions $v_j^{(k)}(t)$ in one mboost-object (with k running over iteration indices) and the effects into the respective directions $h_j^{(k)}(t)$ in another mboost-object, both of subclass FDboost_fac. The number of boosting iterations of FDboost_fac-objects cannot be further increased as in regular mboost-objects.

Value

a list of two mboost models of class `FDboost_fac` containing basis functions for response and covariates, respectively, as base-learners.

A factorized model

References

Stoecker, A., Steyer L. and Greven, S. (2022): Functional additive models on manifolds of planar shapes and forms <arXiv:2109.02624>

See Also

[FDboost_fac-class]

Examples

```
library(FDboost)

# generate irregular toy data ----------------------------------------------------

n <- 100
m <- 40
# covariates
x <- seq(0,2,len = n)
# time & id
set.seed(90384)
t <- runif(n = n*m, -pi,pi)
id <- sample(1:n, size = n*m, replace = TRUE)

# generate components
fx <- ft <- list()
fx[[1]] <- exp(x)
d <- numeric(2)
d[1] <- sqrt(c(crossprod(fx[[1]])))
fx[[1]] <- fx[[1]] / d[1]
fx[[2]] <- -5*x^2
fx[[2]] <- fx[[2]] - fx[[1]] * c(crossprod(fx[[1]], fx[[2]])) # orthogonalize fx[[2]]
d[2] <- sqrt(c(crossprod(fx[[2]])))
fx[[2]] <- fx[[2]] / d[2]
ft[[1]] <- sin(t)
ft[[2]] <- cos(t)
ft[[1]] <- ft[[1]] / sqrt(sum(ft[[1]]^2))
ft[[2]] <- ft[[2]] / sqrt(sum(ft[[2]]^2))

mu1 <- d[1] * fx[[1]][id] * ft[[1]]
mu2 <- d[2] * fx[[2]][id] * ft[[2]]
# add linear covariate
ft[[3]] <- t^2 * sin(4*t)
ft[[3]] <- ft[[3]] - ft[[1]] * c(crossprod(ft[[1]], ft[[3]]))
ft[[3]] <- ft[[3]] - ft[[2]] * c(crossprod(ft[[2]], ft[[3]]))
ft[[3]] <- ft[[3]] / sqrt(sum(ft[[3]]^2))
```

```
set.seed(9234)
fx[[3]] <- runif(0,3, n = length(x))
fx[[3]] <- fx[[3]] - fx[[1]] * c(crossprod(fx[[1]], fx[[3]]))
fx[[3]] <- fx[[3]] - fx[[2]] * c(crossprod(fx[[2]], fx[[3]]))
d[3] <- sqrt(sum(fx[[3]]^2))
fx[[3]] <- fx[[3]] / d[3]

mu3 <- d[3] * fx[[3]][id] * ft[[3]]

mu <- mu1 + mu2 + mu3
# add some noise
y <- mu + rnorm(length(mu), 0, .01)
# and noise covariate
z <- rnorm(n)

# fit FDboost model ------------------------------------------------------

dat <- list(y = y, x = x, t = t, x_lin = fx[[3]], id = id)
m <- FDboost(y ~ bbs(x, knots = 5, df = 2, differences = 0) +
               # bbs(z, knots = 2, df = 2, differences = 0) +
               bols(x_lin, intercept = FALSE, df = 2)
               , ~ bbs(t),
             id = ~ id,
             offset = 0, #numInt = "Riemann",
             control = boost_control(nu = 1),
             data = dat)
MU <- split(mu, id)
PRED <- split(predict(m), id)
Ti <- split(t, id)
t0 <- seq(-pi, pi, length.out = 40)
MU <- do.call(cbind, Map(function(mu, t) approx(t, mu, t0)$y,
          MU, Ti))
PRED <- do.call(cbind, Map(function(mu, t) approx(t, mu, t0)$y,
            PRED, Ti))

opar <- par(mfrow = c(2,2))
image(t0, x, MU)
contour(t0, x, MU, add = TRUE)
image(t0, x, PRED)
contour(t0, x, PRED, add = TRUE)
persp(t0, x, MU, zlim = range(c(MU, PRED), na.rm = TRUE))
persp(t0, x, PRED, zlim = range(c(MU, PRED), na.rm = TRUE))
par(opar)

# factorize model --------------------------------------------------------

fac <- factorize(m)

vi <- as.data.frame(varimp(fac$cov))
# if(require(lattice))
#   barchart(variable ~ reduction, group = blearner, vi, stack = TRUE)

cbind(d^2, sort(vi$reduction, decreasing = TRUE)[1:3])
```

```
x_plot <- list(x, x, fx[[3]])

cols <- c("cornflowerblue", "darkseagreen", "darkred")
opar <- par(mfrow = c(3,2))
wch <- c(1,2,10)
for(w in seq_along(wch)) {
  plot.mboost(fac$resp, which = wch[w], col = "darkgrey", ask = FALSE,
       main = names(fac$resp$baselearner[wch[w]]))
  lines(sort(t), ft[[w]][order(t)]*max(d), col = cols[w], lty = 2)
  plot(fac$cov, which = wch[w],
       main = names(fac$cov$baselearner[wch[w]]))
  points(x_plot[[w]], d[w] * fx[[w]] / max(d), col = cols[w], pch = 3)
}
par(opar)

# re-compose predictions
preds <- lapply(fac, predict)
predf <- rowSums(preds$resp * preds$cov[id, ])
PREDf <- split(predf, id)
PREDf <- do.call(cbind, Map(function(mu, t) approx(t, mu, t0)$y,
                            PREDf, Ti))
opar <- par(mfrow = c(1,2))
image(t0,x, PRED, main = "original prediction")
contour(t0,x, PRED, add = TRUE)
image(t0,x,PREDf, main = "recomposed")
contour(t0,x, PREDf, add = TRUE)
par(opar)

stopifnot(all.equal(PRED, PREDf))

# check out other methods
set.seed(8399)
newdata_resp <- list(t = sort(runif(60, min(t), max(t))))
a <- predict(fac$resp, newdata = newdata_resp, which = 1:5)
plot(newdata_resp$t, a[, 1])
# coef method
cf <- coef(fac$resp, which = 1)


# check factorization on a new dataset -------------------------------------

t_grid <- seq(-pi,pi,len = 30)
x_grid <- seq(0,2,len = 30)
x_lin_grid <- seq(min(dat$x_lin), max(dat$x_lin), len = 30)

# use grid data for factorization
griddata <- expand.grid(
  # time
  t = t_grid,
  # covariates
  x = x_grid,
```

```
  x_lin = 0
)

griddata_lin <- expand.grid(
  t = seq(-pi, pi, len = 30),
  x = 0,
  x_lin = x_lin_grid
)

griddata <- rbind(griddata, griddata_lin)

griddata$id <- as.numeric(factor(paste(griddata$x, griddata$x_lin, sep = ":")))

fac2 <- factorize(m, newdata = griddata)

ratio <- -max(abs(predict(fac$resp, which = 1))) / max(abs(predict(fac2$resp, which = 1)))

opar <- par(mfrow = c(3,2))
wch <- c(1,2,10)
for(w in seq_along(wch)) {
  plot.mboost(fac$resp, which = wch[w], col = "darkgrey", ask = FALSE,
                                main = names(fac$resp$baselearner[wch[w]]))

  lines(sort(griddata$t),
        ratio*predict(fac2$resp, which = wch[w])[order(griddata$t)],
        col = cols[w], lty = 2)
  plot(fac$cov, which = wch[w],
       main = names(fac$cov$baselearner[wch[w]]))
  this_x <- fac2$cov$model.frame(which = wch[w])[[1]][[1]]
    lines(sort(this_x), 1/ratio*predict(fac2$cov, which = wch[w])[order(this_x)],
          col = cols[w], lty = 1)
}
par(opar)

# check predictions
p <- predict(fac2$resp, which = 1)
library(FDboost)

# generate regular toy data ------------------------------------------------

n <- 100
m <- 40
# covariates
x <- seq(0,2,len = n)
# time
t <- seq(-pi,pi,len = m)
# generate components
fx <- ft <- list()
fx[[1]] <- exp(x)
d <- numeric(2)
d[1] <- sqrt(c(crossprod(fx[[1]])))
fx[[1]] <- fx[[1]] / d[1]
fx[[2]] <- -5*x^2
```

```
fx[[2]] <- fx[[2]] - fx[[1]] * c(crossprod(fx[[1]], fx[[2]])) # orthogonalize fx[[2]]
d[2] <- sqrt(c(crossprod(fx[[2]])))
fx[[2]] <- fx[[2]] / d[2]
ft[[1]] <- sin(t)
ft[[2]] <- cos(t)
ft[[1]] <- ft[[1]] / sqrt(sum(ft[[1]]^2))
ft[[2]] <- ft[[2]] / sqrt(sum(ft[[2]]^2))
mu1 <- d[1] * fx[[1]] %*% t(ft[[1]])
mu2 <- d[2] * fx[[2]] %*% t(ft[[2]])
# add linear covariate
ft[[3]] <- t^2 * sin(4*t)
ft[[3]] <- ft[[3]] - ft[[1]] * c(crossprod(ft[[1]], ft[[3]]))
ft[[3]] <- ft[[3]] - ft[[2]] * c(crossprod(ft[[2]], ft[[3]]))
ft[[3]] <- ft[[3]] / sqrt(sum(ft[[3]]^2))
set.seed(9234)
fx[[3]] <- runif(0,3, n = length(x))
fx[[3]] <- fx[[3]] - fx[[1]] * c(crossprod(fx[[1]], fx[[3]]))
fx[[3]] <- fx[[3]] - fx[[2]] * c(crossprod(fx[[2]], fx[[3]]))
d[3] <- sqrt(sum(fx[[3]]^2))
fx[[3]] <- fx[[3]] / d[3]
mu3 <- d[3] * fx[[3]] %*% t(ft[[3]])

mu <- mu1 + mu2 + mu3
# add some noise
y <- mu + rnorm(length(mu), 0, .01)
# and noise covariate
z <- rnorm(n)

# fit FDboost model ------------------------------------------------------

dat <- list(y = y, x = x, t = t, x_lin = fx[[3]])
m <- FDboost(y ~ bbs(x, knots = 5, df = 2, differences = 0) +
               # bbs(z, knots = 2, df = 2, differences = 0) +
               bols(x_lin, intercept = FALSE, df = 2)
               , ~ bbs(t), offset = 0,
             control = boost_control(nu = 1),
             data = dat)

opar <- par(mfrow = c(1,2))
image(t, x, t(mu))
contour(t, x, t(mu), add = TRUE)
image(t, x, t(predict(m)))
contour(t, x, t(predict(m)), add = TRUE)
par(opar)

# factorize model ------------------------------------------------------

fac <- factorize(m)

vi <- as.data.frame(varimp(fac$cov))
# if(require(lattice))
#   barchart(variable ~ reduction, group = blearner, vi, stack = TRUE)
```

```
cbind(d^2, vi$reduction[c(1:2, 10)])


x_plot <- list(x, x, fx[[3]])

cols <- c("cornflowerblue", "darkseagreen", "darkred")
opar <- par(mfrow = c(3,2))
wch <- c(1,2,10)
for(w in seq_along(wch)) {
  plot.mboost(fac$resp, which = wch[w], col = "darkgrey", ask = FALSE,
       main = names(fac$resp$baselearner[wch[w]]))
  lines(t, ft[[w]]*max(d), col = cols[w], lty = 2)
  plot(fac$cov, which = wch[w],
       main = names(fac$cov$baselearner[wch[w]]))
  points(x_plot[[w]], d[w] * fx[[w]] / max(d), col = cols[w], pch = 3)
}
par(opar)

# re-compose prediction
preds <- lapply(fac, predict)
PREDSf <- array(0, dim = c(nrow(preds$resp),nrow(preds$cov)))
for(i in seq_len(ncol(preds$resp)))
  PREDSf <- PREDSf + preds$resp[,i] %*% t(preds$cov[,i])

opar <- par(mfrow = c(1,2))
image(t,x, t(predict(m)), main = "original prediction")
contour(t,x, t(predict(m)), add = TRUE)
image(t,x,PREDSf, main = "recomposed")
contour(t,x, PREDSf, add = TRUE)
par(opar)
# => matches
stopifnot(all.equal(as.numeric(t(predict(m))), as.numeric(PREDSf)))

# check out other methods
set.seed(8399)
newdata_resp <- list(t = sort(runif(60, min(t), max(t))))
a <- predict(fac$resp, newdata = newdata_resp, which = 1:5)
plot(newdata_resp$t, a[, 1])
# coef method
cf <- coef(fac$resp, which = 1)
```

FDboost *Model-based Gradient Boosting for Functional Response*

Description

Gradient boosting for optimizing arbitrary loss functions, where component-wise models are utilized as base-learners in the case of functional responses. Scalar responses are treated as the special

case where each functional response has only one observation. This function is a wrapper for mboost's mboost and its siblings to fit models of the general form

$$\xi(Y_i(t)|X_i = x_i) = \sum_j h_j(x_i, t), i = 1, ..., N,$$

with a functional (but not necessarily continuous) response $Y(t)$, transformation function ξ, e.g., the expectation, the median or some quantile, and partial effects $h_j(x_i, t)$ depending on covariates x_i and the current index of the response t. The index of the response can be for example time. Possible effects are, e.g., a smooth intercept $\beta_0(t)$, a linear functional effect $\int x_i(s)\beta(s,t)ds$, potentially with integration limits depending on t, smooth and linear effects of scalar covariates $f(z_i, t)$ or $z_i\beta(t)$. A hands-on tutorial for the package can be found at <doi:10.18637/jss.v094.i10>.

Usage

```
FDboost(
  formula,
  timeformula,
  id = NULL,
  numInt = "equal",
  data,
  weights = NULL,
  offset = NULL,
  offset_control = o_control(),
  check0 = FALSE,
  ...
)
```

Arguments

formula	a symbolic description of the model to be fit. Per default no intercept is added, only a smooth offset, see argument offset. To add a smooth intercept, use 1, e.g., y ~ 1 for a pure intercept model.
timeformula	one-sided formula for the specification of the effect over the index of the response. For functional response $Y_i(t)$ typically use ~ bbs(t) to obtain smooth effects over t. In the limiting case of Y_i being a scalar response, use ~ bols(1), which sets up a base-learner for the scalar 1. Or use timeformula = NULL, then the scalar response is treated as scalar.
id	defaults to NULL which means that all response trajectories are observed on a common grid allowing to represent the response as a matrix. If the response is given in long format for observation-specific grids, id contains the information which observations belong to the same trajectory and must be supplied as a formula, ~ nameid, where the variable nameid should contain integers 1, 2, 3, ..., N.
numInt	integration scheme for the integration of the loss function. One of c("equal", "Riemann") meaning equal weights of 1 or trapezoidal Riemann weights. Alternatively a vector of length ncol(response) containing positive weights can be specified.

data	a data frame or list containing the variables in the model.
weights	only for internal use to specify resampling weights; per default all weights are equal to 1.
offset	a numeric vector to be used as offset over the index of the response (optional). If no offset is specified, per default offset = NULL which means that a smooth time-specific offset is computed and used before the model fit to center the data. If you do not want to use a time-specific offset, set offset = "scalar" to get an overall scalar offset, like in mboost.
offset_control	parameters for the estimation of the offset, defaults to o_control(), see o_control.
check0	logical, for response in matrix form, i.e. response that is observed on a common grid, check the fitted effects for the sum-to-zero constraint $h_j(x_i)(t) = 0$ for all t and give a warning if it is not fulfilled. Defaults to FALSE.
...	additional arguments passed to mboost, including, family and control.

Details

In matrix representation of functional response and covariates each row represents one functional observation, e.g., Y[i,t_g] corresponds to $Y_i(t_g)$, giving a <number of curves> by <number of evaluations> matrix. For the model fit, the matrix of the functional response evaluations $Y_i(t_g)$ are stacked internally into one long vector.

If it is possible to represent the model as a generalized linear array model (Currie et al., 2006), the array structure is used for an efficient implementation, see mboost. This is only possible if the design matrix can be written as the Kronecker product of two marginal design matrices yielding a functional linear array model (FLAM), see Brockhaus et al. (2015) for details. The Kronecker product of two marginal bases is implemented in R-package mboost in the function %O%, see %O%.

When %O% is called with a specification of df in both base-learners, e.g., bbs(x1, df = df1) %O% bbs(t, df = df2), the global df for the Kroneckered base-learner is computed as df = df1 * df2. And thus the penalty has only one smoothness parameter lambda resulting in an isotropic penalty. A Kronecker product with anisotropic penalty is %A%, allowing for different amount of smoothness in the two directions, see %A%. If the formula contains base-learners connected by %O%, %A% or %A0%, those effects are not expanded with timeformula, allowing for model specifications with different effects in time-direction.

If the response is observed on curve-specific grids it must be supplied as a vector in long format and the argument id has to be specified (as formula!) to define which observations belong to which curve. In this case the base-learners are built as row tensor-products of marginal base-learners, see Scheipl et al. (2015) and Brockhaus et al. (2017), for details on how to set up the effects. The row tensor product of two marginal bases is implemented in R-package mboost in the function %X%, see %X%.

A scalar response can be seen as special case of a functional response with only one time-point, and thus it can be represented as FLAM with basis 1 in time-direction, use timeformula = ~bols(1). In this case, a penalty in the time-direction is used, see Brockhaus et al. (2015) for details. Alternatively, the scalar response is fitted as scalar response, like in the function mboost in package mboost. The advantage of using FDboost in that case is that methods for the functional base-learners are available, e.g., plot.

The desired regression type is specified by the family-argument, see the help-page of mboost. For example a mean regression model is obtained by family = Gaussian() which is the default or median regression by family = QuantReg(); see Family for a list of implemented families.

With FDboost the following covariate effects can be estimated by specifying the following effects in the formula (similar to function pffr in R-package refund. The timeformula is used to expand the effects in t-direction.

- Linear functional effect of scalar (numeric or factor) covariate z that varies smoothly over t, i.e. $z_i\beta(t)$, specified as bolsc(z), see bolsc, or for a group effect with mean zero use brandomc(z).
- Nonlinear effects of a scalar covariate that vary smoothly over t, i.e. $f(z_i, t)$, specified as bbsc(z), see bbsc.
- (Nonlinear) effects of scalar covariates that are constant over t, e.g., $f(z_i)$, specified as c(bbs(z)), or βz_i, specified as c(bols(z)).
- Interaction terms between two scalar covariates, e.g., $z_i 1 zi2\beta(t)$, are specified as bols(z1) %Xc% bols(z2) and an interaction $z_i 1 f(zi2, t)$ as bols(z1) %Xc% bbs(z2), as %Xc% applies the sum-to-zero constraint to the desgin matrix of the tensor product built by %Xc%, see %Xc%.
- Function-on-function regression terms of functional covariates x, e.g., $\int x_i(s)\beta(s,t)ds$, specified as bsignal(x, s = s), using P-splines, see bsignal. Terms given by bfpc provide FPC-based effects of functional covariates, see bfpc.
- Function-on-function regression terms of functional covariates x with integration limits $[l(t), u(t)]$ depending on t, e.g., $\int_{[} l(t), u(t)]x_i(s)\beta(s,t)ds$, specified as bhist(x, s = s, time = t, limits). The limits argument defaults to "s<=t" which yields a historical effect with limits $[min(t), t]$, see bhist.
- Concurrent effects of functional covariates x measured on the same grid as the response, i.e., $x_i(s)\beta(t)$, are specified as bconcurrent(x, s = s, time = t), see bconcurrent.
- Interaction effects can be estimated as tensor product smooth, e.g., $z \int x_i(s)\beta(s,t)ds$ as bsignal(x, s = s) %X% bolsc(z)
- For interaction effects with historical functional effects, e.g., $z_i \int_{[} l(t), u(t)]x_i(s)\beta(s,t)ds$ the base-learner bhistx should be used instead of bhist, e.g., bhistx(x, limits) %X% bolsc(z), see bhistx.
- Generally, the c()-notation can be used to get effects that are constant over the index of the functional response.
- If the formula in FDboost contains base-learners connected by %O%, %A% or %A0%, those effects are not expanded with timeformula, allowing for model specifications with different effects in time-direction.

In order to obtain a fair selection of base-learners, the same degrees of freedom (df) should be specified for all baselearners. If the number of df differs among the base-learners, the selection is biased towards more flexible base-learners with higher df as they are more likely to yield larger improvements of the fit. It is recommended to use a rather small number of df for all base-learners. It is not possible to specify df larger than the rank of the design matrix. For base-learners with rank-deficient penalty, it is not possible to specify df smaller than the rank of the null space of the penalty (e.g., in bbs unpenalized part of P-splines). The df of the base-learners in an FDboost-object can be checked using extract(object, "df"), see extract.

The most important tuning parameter of component-wise gradient boosting is the number of boosting iterations. It is recommended to use the number of boosting iterations as only tuning parameter, fixing the step-length at a small value (e.g., nu = 0.1). Note that the default number of boosting iterations is 100 which is arbitrary and in most cases not adequate (the optimal number of boosting

iterations can considerably exceed 100). The optimal stopping iteration can be determined by resampling methods like cross-validation or bootstrapping, see the function `cvrisk.FDboost` which searches the optimal stopping iteration on a grid, which in many cases has to be extended.

Value

An object of class `FDboost` that inherits from `mboost`. Special `predict.FDboost`, `coef.FDboost` and `plot.FDboost` methods are available. The methods of `mboost` are available as well, e.g., `extract`. The `FDboost`-object is a named list containing:

`...`	all elements of an `mboost`-object
`yname`	the name of the response
`ydim`	dimension of the response matrix, if the response is represented as such
`yind`	the observation (time-)points of the response, i.e. the evaluation points, with its name as attribute
`data`	the data that was used for the model fit
`id`	the id variable of the response
`predictOffset`	the function to predict the smooth offset
`offsetFDboost`	offset as specified in call to FDboost
`offsetMboost`	offset as given to mboost
`call`	the call to `FDboost`
`callEval`	the evaluated function call to `FDboost` without data
`numInt`	value of argument `numInt` determining the numerical integration scheme
`timeformula`	the time-formula
`formulaFDboost`	the formula with which `FDboost` was called
`formulaMboost`	the formula with which `mboost` was called within `FDboost`

Author(s)

Sarah Brockhaus, Torsten Hothorn

References

Brockhaus, S., Ruegamer, D. and Greven, S. (2017): Boosting Functional Regression Models with FDboost. <doi:10.18637/jss.v094.i10>

Brockhaus, S., Scheipl, F., Hothorn, T. and Greven, S. (2015): The functional linear array model. Statistical Modelling, 15(3), 279-300.

Brockhaus, S., Melcher, M., Leisch, F. and Greven, S. (2017): Boosting flexible functional regression models with a high number of functional historical effects, Statistics and Computing, 27(4), 913-926.

Currie, I.D., Durban, M. and Eilers P.H.C. (2006): Generalized linear array models with applications to multidimensional smoothing. Journal of the Royal Statistical Society, Series B-Statistical Methodology, 68(2), 259-280.

Scheipl, F., Staicu, A.-M. and Greven, S. (2015): Functional additive mixed models, Journal of Computational and Graphical Statistics, 24(2), 477-501.

See Also

Note that FDboost calls mboost directly. See, e.g., bsignal and bbsc for possible base-learners.

Examples

```
######## Example for function-on-scalar-regression
data("viscosity", package = "FDboost")
## set time-interval that should be modeled
interval <- "101"

## model time until "interval" and take log() of viscosity
end <- which(viscosity$timeAll == as.numeric(interval))
viscosity$vis <- log(viscosity$visAll[,1:end])
viscosity$time <- viscosity$timeAll[1:end]
# with(viscosity, funplot(time, vis, pch = 16, cex = 0.2))

## fit median regression model with 100 boosting iterations,
## step-length 0.4 and smooth time-specific offset
## the factors are coded such that the effects are zero for each timepoint t
## no integration weights are used!
mod1 <- FDboost(vis ~ 1 + bolsc(T_C, df = 2) + bolsc(T_A, df = 2),
                timeformula = ~ bbs(time, df = 4),
                numInt = "equal", family = QuantReg(),
                offset = NULL, offset_control = o_control(k_min = 9),
                data = viscosity, control=boost_control(mstop = 100, nu = 0.4))


  #### find optimal mstop over 5-fold bootstrap, small number of folds for example
  #### do the resampling on the level of curves

  ## possibility 1: smooth offset and transformation matrices are refitted
  set.seed(123)
  appl1 <- applyFolds(mod1, folds = cv(rep(1, length(unique(mod1$id))), B = 5),
                      grid = 1:500)
  ## plot(appl1)
  mstop(appl1)
  mod1[mstop(appl1)]

  ## possibility 2: smooth offset is refitted,
  ## computes oob-risk and the estimated coefficients on the folds
  set.seed(123)
  val1 <- validateFDboost(mod1, folds = cv(rep(1, length(unique(mod1$id))), B = 5),
                          grid = 1:500)
  ## plot(val1)
  mstop(val1)
  mod1[mstop(val1)]

  ## possibility 3: very efficient
  ## using the function cvrisk; be careful to do the resampling on the level of curves
  folds1 <- cvLong(id = mod1$id, weights = model.weights(mod1), B = 5)
  cvm1 <- cvrisk(mod1, folds = folds1, grid = 1:500)
  ## plot(cvm1)
```

```
  mstop(cvm1)

## look at the model
summary(mod1)
coef(mod1)
plot(mod1)
plotPredicted(mod1, lwdPred = 2)


######## Example for scalar-on-function-regression
data("fuelSubset", package = "FDboost")

## center the functional covariates per observed wavelength
fuelSubset$UVVIS <- scale(fuelSubset$UVVIS, scale = FALSE)
fuelSubset$NIR <- scale(fuelSubset$NIR, scale = FALSE)

## to make mboost:::df2lambda() happy (all design matrix entries < 10)
## reduce range of argvals to [0,1] to get smaller integration weights
fuelSubset$uvvis.lambda <- with(fuelSubset, (uvvis.lambda - min(uvvis.lambda)) /
                                        (max(uvvis.lambda) - min(uvvis.lambda) ))
fuelSubset$nir.lambda <- with(fuelSubset, (nir.lambda - min(nir.lambda)) /
                                        (max(nir.lambda) - min(nir.lambda) ))

## model fit with scalar response
## include no intercept as all base-learners are centered around 0
mod2 <- FDboost(heatan ~ bsignal(UVVIS, uvvis.lambda, knots = 40, df = 4, check.ident = FALSE)
                + bsignal(NIR, nir.lambda, knots = 40, df = 4, check.ident = FALSE),
              timeformula = NULL, data = fuelSubset, control = boost_control(mstop = 200))

## additionally include a non-linear effect of the scalar variable h2o
mod2s <- FDboost(heatan ~ bsignal(UVVIS, uvvis.lambda, knots = 40, df = 4, check.ident = FALSE)
                + bsignal(NIR, nir.lambda, knots = 40, df = 4, check.ident = FALSE)
                + bbs(h2o, df = 4),
              timeformula = NULL, data = fuelSubset, control = boost_control(mstop = 200))

## alternative model fit as FLAM model with scalar response; as timeformula = ~ bols(1)
## adds a penalty over the index of the response, i.e., here a ridge penalty
## thus, mod2f and mod2 have different penalties
mod2f <- FDboost(heatan ~ bsignal(UVVIS, uvvis.lambda, knots = 40, df = 4, check.ident = FALSE)
                + bsignal(NIR, nir.lambda, knots = 40, df = 4, check.ident = FALSE),
            timeformula = ~ bols(1), data = fuelSubset, control = boost_control(mstop = 200))


  ## bootstrap to find optimal mstop takes some time
  set.seed(123)
  folds2 <- cv(weights = model.weights(mod2), B = 10)
  cvm2 <- cvrisk(mod2, folds = folds2, grid = 1:1000)
  mstop(cvm2) ## mod2[327]
  summary(mod2)
  ## plot(mod2)


## Example for function-on-function-regression
```

```
if(require(fda)){

  data("CanadianWeather", package = "fda")
  CanadianWeather$l10precip <- t(log(CanadianWeather$monthlyPrecip))
  CanadianWeather$temp <- t(CanadianWeather$monthlyTemp)
  CanadianWeather$region <- factor(CanadianWeather$region)
  CanadianWeather$month.s <- CanadianWeather$month.t <- 1:12

  ## center the temperature curves per time-point
  CanadianWeather$temp <- scale(CanadianWeather$temp, scale = FALSE)
  rownames(CanadianWeather$temp) <- NULL ## delete row-names

  ## fit model with cyclic splines over the year
  mod3 <- FDboost(l10precip ~ bols(region, df = 2.5, contrasts.arg = "contr.dummy")
                  + bsignal(temp, month.s, knots = 11, cyclic = TRUE,
                          df = 2.5, boundary.knots = c(0.5,12.5), check.ident = FALSE),
                  timeformula = ~ bbs(month.t, knots = 11, cyclic = TRUE,
                                      df = 3, boundary.knots = c(0.5, 12.5)),
                  offset = "scalar", offset_control = o_control(k_min = 5),
                  control = boost_control(mstop = 60),
                  data = CanadianWeather)


   #### find the optimal mstop over 5-fold bootstrap
   ## using the function applyFolds
   set.seed(123)
   folds3 <- cv(rep(1, length(unique(mod3$id))), B = 5)
   appl3 <- applyFolds(mod3, folds = folds3, grid = 1:200)

   ## use function cvrisk; be careful to do the resampling on the level of curves
   set.seed(123)
   folds3long <- cvLong(id = mod3$id, weights = model.weights(mod3), B = 5)
   cvm3 <- cvrisk(mod3, folds = folds3long, grid = 1:200)
   mstop(cvm3) ## mod3[64]

   summary(mod3)
   ## plot(mod3, pers = TRUE)

}

######## Example for functional response observed on irregular grid
######## Delete part of observations in viscosity data-set
data("viscosity", package = "FDboost")
## set time-interval that should be modeled
interval <- "101"

## model time until "interval" and take log() of viscosity
end <- which(viscosity$timeAll == as.numeric(interval))
viscosity$vis <- log(viscosity$visAll[,1:end])
viscosity$time <- viscosity$timeAll[1:end]
# with(viscosity, funplot(time, vis, pch = 16, cex = 0.2))

## only keep one eighth of the observation points
```

```
set.seed(123)
selectObs <- sort(sample(x = 1:(64*46), size = 64*46/4, replace = FALSE))
dataIrregular <- with(viscosity, list(vis = c(vis)[selectObs],
                                      T_A = T_A, T_C = T_C,
                                      time = rep(time, each = 64)[selectObs],
                                      id = rep(1:64, 46)[selectObs]))

## fit median regression model with 50 boosting iterations,
## step-length 0.4 and smooth time-specific offset
## the factors are in effect coding -1, 1 for the levels
## no integration weights are used!
mod4 <- FDboost(vis ~ 1 + bols(T_C, contrasts.arg = "contr.sum", intercept = FALSE)
                + bols(T_A, contrasts.arg = "contr.sum", intercept=FALSE),
                timeformula = ~ bbs(time, lambda = 100), id = ~id,
                numInt = "Riemann", family = QuantReg(),
                offset = NULL, offset_control = o_control(k_min = 9),
                data = dataIrregular, control = boost_control(mstop = 50, nu = 0.4))
## summary(mod4)
## plot(mod4)
## plotPredicted(mod4, lwdPred = 2)


  ## Find optimal mstop, small grid/low B for a fast example
  set.seed(123)
  folds4 <- cv(rep(1, length(unique(mod4$id))), B = 3)
  appl4 <- applyFolds(mod4, folds = folds4, grid = 1:50)
  ## val4 <- validateFDboost(mod4, folds = folds4, grid = 1:50)

  set.seed(123)
  folds4long <- cvLong(id = mod4$id, weights = model.weights(mod4), B = 3)
  cvm4 <- cvrisk(mod4, folds = folds4long, grid = 1:50)
  mstop(cvm4)


## Be careful if you want to predict newdata with irregular response,
## as the argument index is not considered in the prediction of newdata.
## Thus, all covariates have to be repeated according to the number of observations
## in each response trajectroy.
## Predict four response curves with full time-observations
## for the four combinations of T_A and T_C.
newd <- list(T_A = factor(c(1,1,2,2), levels = 1:2,
                          labels = c("low", "high"))[rep(1:4, length(viscosity$time))],
             T_C = factor(c(1,2,1,2), levels = 1:2,
                          labels = c("low", "high"))[rep(1:4, length(viscosity$time))],
             time = rep(viscosity$time, 4))

pred <- predict(mod4, newdata = newd)
## funplot(x = rep(viscosity$time, 4), y = pred, id = rep(1:4, length(viscosity$time)))
```

FDboostLSS *Model-based Gradient Boosting for Functional GAMLSS*

Description

Function for fitting generalized additive models for location, scale and shape (GAMLSS) with functional data using component-wise gradient boosting, for details see Brockhaus et al. (2018).

Usage

```
FDboostLSS(
  formula,
  timeformula,
  data = list(),
  families = GaussianLSS(),
  control = boost_control(),
  weights = NULL,
  method = c("cyclic", "noncyclic"),
  ...
)
```

Arguments

formula	a symbolic description of the model to be fit. If `formula` is a single formula, the same formula is used for all distribution parameters. `formula` can also be a (named) list, where each list element corresponds to one distribution parameter of the GAMLSS distribution. The names must be the same as in the `families`.
timeformula	one-sided formula for the expansion over the index of the response. For a functional response $Y_i(t)$ typically `~bbs(t)` to obtain a smooth expansion of the effects along `t`. In the limiting case that Y_i is a scalar response use `~bols(1)`, which sets up a base-learner for the scalar 1. Or you can use `timeformula=NULL`, then the scalar response is treated as scalar. Analogously to `formula`, `timeformula` can either be a one-sided formula or a named list of one-sided formulas.
data	a data frame or list containing the variables in the model.
families	an object of class `families`. It can be either one of the pre-defined distributions that come along with the package `gamboostLSS` or a new distribution specified by the user (see `Families` for details). Per default, the two-parametric `GaussianLSS` family is used.
control	a list of parameters controlling the algorithm. For more details see `boost_control`.
weights	does not work!
method	fitting method, currently two methods are supported: `"cyclic"` (see Mayr et al., 2012) and `"noncyclic"` (algorithm with inner loss of Thomas et al., 2018).
...	additional arguments passed to `FDboost`, including, `family` and `control`.

Details

For details on the theory of GAMLSS, see Rigby and Stasinopoulos (2005). FDboostLSS calls FDboost to fit the distribution parameters of a GAMLSS - a functional boosting model is fitted for each parameter of the response distribution. In mboostLSS, details on boosting of GAMLSS based on Mayr et al. (2012) and Thomas et al. (2018) are given. In FDboost, details on boosting regression models with functional variables are given (Brockhaus et al., 2015, Brockhaus et al., 2017).

Value

An object of class FDboostLSS that inherits from mboostLSS. The FDboostLSS-object is a named list containing one list entry per distribution parameter and some attributes. The list is named like the parameters, e.g. mu and sigma, if the parameters mu and sigma are modeled. Each list-element is an object of class FDboost.

Author(s)

Sarah Brockhaus

References

Brockhaus, S., Scheipl, F., Hothorn, T. and Greven, S. (2015). The functional linear array model. Statistical Modelling, 15(3), 279-300.

Brockhaus, S., Melcher, M., Leisch, F. and Greven, S. (2017): Boosting flexible functional regression models with a high number of functional historical effects, Statistics and Computing, 27(4), 913-926.

Brockhaus, S., Fuest, A., Mayr, A. and Greven, S. (2018): Signal regression models for location, scale and shape with an application to stock returns. Journal of the Royal Statistical Society: Series C (Applied Statistics), 67, 665-686.

Mayr, A., Fenske, N., Hofner, B., Kneib, T. and Schmid, M. (2012): Generalized additive models for location, scale and shape for high-dimensional data - a flexible approach based on boosting. Journal of the Royal Statistical Society: Series C (Applied Statistics), 61(3), 403-427.

Rigby, R. A. and D. M. Stasinopoulos (2005): Generalized additive models for location, scale and shape (with discussion). Journal of the Royal Statistical Society: Series C (Applied Statistics), 54(3), 507-554.

Thomas, J., Mayr, A., Bischl, B., Schmid, M., Smith, A., and Hofner, B. (2018), Gradient boosting for distributional regression - faster tuning and improved variable selection via noncyclical updates. Statistics and Computing, 28, 673-687.

Stoecker, A., Brockhaus, S., Schaffer, S., von Bronk, B., Opitz, M., and Greven, S. (2019): Boosting Functional Response Models for Location, Scale and Shape with an Application to Bacterial Competition. https://arxiv.org/abs/1809.09881

See Also

Note that FDboostLSS calls FDboost directly.

Examples

```
########### simulate Gaussian scalar-on-function data
n <- 500 ## number of observations
G <- 120 ## number of observations per functional covariate
set.seed(123) ## ensure reproducibility
z <- runif(n) ## scalar covariate
z <- z - mean(z)
s <- seq(0, 1, l=G) ## index of functional covariate
## generate functional covariate
if(require(splines)){
   x <- t(replicate(n, drop(bs(s, df = 5, int = TRUE) %*% runif(5, min = -1, max = 1))))
}else{
  x <- matrix(rnorm(n*G), ncol = G, nrow = n)
}
x <- scale(x, center = TRUE, scale = FALSE) ## center x per observation point

mu <- 2 + 0.5*z + (1/G*x) %*% sin(s*pi)*5 ## true functions for expectation
sigma <- exp(0.5*z - (1/G*x) %*% cos(s*pi)*2) ## for standard deviation

y <- rnorm(mean = mu, sd = sigma, n = n) ## draw respone y_i ~ N(mu_i, sigma_i)

## save data as list containing s as well
dat_list <- list(y = y, z = z, x = I(x), s = s)

## model fit with noncyclic algorithm assuming Gaussian location scale model
m_boost <- FDboostLSS(list(mu = y ~ bols(z, df = 2) + bsignal(x, s, df = 2, knots = 16),
                           sigma = y ~ bols(z, df = 2) + bsignal(x, s, df = 2, knots = 16)),
                              timeformula = NULL, data = dat_list, method = "noncyclic")
summary(m_boost)


 if(require(gamboostLSS)){
  ## find optimal number of boosting iterations on a grid in 1:1000
  ## using 5-fold bootstrap
  ## takes some time, easy to parallelize on Linux
  set.seed(123)
  cvr <- cvrisk(m_boost, folds = cv(model.weights(m_boost[[1]]), B = 5),
                grid = 1:1000, trace = FALSE)
  ## use model at optimal stopping iterations
  m_boost <- m_boost[mstop(cvr)] ## 832

  ## plot smooth effects of functional covariates for mu and sigma
  oldpar <- par(mfrow = c(1,2))
  plot(m_boost$mu, which = 2, ylim = c(0,5))
  lines(s, sin(s*pi)*5, col = 3, lwd = 2)
  plot(m_boost$sigma, which = 2, ylim = c(-2.5,2.5))
  lines(s, -cos(s*pi)*2, col = 3, lwd = 2)
  par(oldpar)
 }
```

FDboost_fac-class *'FDboost_fac' S3 class for factorized FDboost model components*

Description

Model factorization with 'factorize()' decomposes an 'FDboost' model into two objects of class 'FDboost_fac' - one for the response and one for the covariate predictor. The first is essentially an 'FDboost' object and the second an 'mboost' object, however, in a 'read-only' mode and slightly adjusted methods (method defaults).

See Also

[factorize(), factorize.FDboost()]

fitted.FDboost *Fitted values of a boosted functional regression model*

Description

Takes a fitted FDboost-object and computes the fitted values.

Usage

```
## S3 method for class 'FDboost'
fitted(object, toFDboost = TRUE, ...)
```

Arguments

object	a fitted FDboost-object
toFDboost	logical, defaults to TRUE. In case of regular response in wide format (i.e., response is supplied as matrix): should the predictions be returned as matrix, or list of matrices instead of vectors
...	additional arguments passed on to predict.FDboost

Value

matrix or vector of fitted values

See Also

FDboost for the model fit.

`fuelSubset` *Spectral data of fossil fuels*

Description

For 129 laboratory samples of fossil fuels the heat value and the humidity were determined together with two spectra. One spectrum is ultraviolet-visible (UV-VIS), measured at 1335 wavelengths in the range of 250.4 to 878.4 nanometer (nm), the other a near infrared spectrum (NIR) measured at 2307 wavelengths in the range of 800.4 to 2779.0 nm. `fuelSubset` is a subset of the original dataset containing only 10% of the original measures of the spectra, resulting in 231 measures of the NIR spectrum and 134 measures of the UVVIS spectrum.

Usage

```
data("fuelSubset")
```

Format

A data list with 129 observations on the following 7 variables.

`heatan` heat value in mega joule (mJ)

`h2o` humidity in percent

`NIR` near infrared spectrum (NIR)

`UVVIS` ultraviolet-visible spectrum (UV-VIS)

`nir.lambda` wavelength of NIR spectrum in nm

`uvvis.lambda` wavelength of UV-VIS spectrum in nm

`h2o.fit` predicted values of humidity

Details

The aim is to predict the heat value using the spectral data. The variable `h2o.fit` was generated by a functional linear regression model, using both spectra and their derivatives as predictors.

Source

Siemens AG

Fuchs, K., Scheipl, F. & Greven, S. (2015), Penalized scalar-on-functions regression with interaction term. Computational Statistics and Data Analysis. 81, 38-51.

Examples

```
data("fuelSubset", package = "FDboost")

## center the functional covariates per observed wavelength
fuelSubset$UVVIS <- scale(fuelSubset$UVVIS, scale = FALSE)
fuelSubset$NIR <- scale(fuelSubset$NIR, scale = FALSE)
```

```
## to make mboost::df2lambda() happy (all design matrix entries < 10)
## reduce range of argvals to [0,1] to get smaller integration weights
fuelSubset$uvvis.lambda <- with(fuelSubset, (uvvis.lambda - min(uvvis.lambda)) /
                                    (max(uvvis.lambda) - min(uvvis.lambda) ))
fuelSubset$nir.lambda <- with(fuelSubset, (nir.lambda - min(nir.lambda)) /
                                    (max(nir.lambda) - min(nir.lambda) ))


### fit mean regression model with 100 boosting iterations,
### step-length 0.1 and
mod <- FDboost(heatan ~ bsignal(UVVIS, uvvis.lambda, knots=40, df=4, check.ident=FALSE)
               + bsignal(NIR, nir.lambda, knots=40, df=4, check.ident=FALSE),
               timeformula = NULL, data = fuelSubset)
summary(mod)
## plot(mod)
```

---

funMRD *Functional MRD*

Description

Calculates the functional MRD for a fitted FDboost-object

Usage

```
funMRD(object, overTime = TRUE, breaks = object$yind, global = FALSE, ...)
```

Arguments

object	fitted FDboost-object with regular response
overTime	per default the functional MRD is calculated over time if overTime=FALSE, the MRD is calculated per curve
breaks	an optional vector or number giving the time-points at which the model is evaluated. Can be specified as number of equidistant time-points or as vector of time-points. Defaults to the index of the response in the model.
global	logical. defaults to FALSE, if TRUE the global MRD like in a normal linear model is calculated
...	currently not used

Details

Formula to calculate MRD over time, overTime=TRUE:
$MRD(t) = n^{-1} \sum_i |Y_i(t) - \hat{Y}_i(t)|/|Y_i(t)|$

Formula to calculate MRD over subjects, overTime=FALSE:
$MRD_i = \int |Y_i(t) - \hat{Y}_i(t)|/|Y_i(t)| dt \approx G^{-1} \sum_g |Y_i(t_g) - \hat{Y}_i(t_g)|/|Y_i(t)|$

Value

Returns a vector with the calculated MRD and some extra information in attributes.

Note

`breaks` cannot be changed in the case the `bsignal()` is used over the same domain as the response! In that case you would have to rename the index of the response or that of the covariates.

`funMSE` *Functional MSE*

Description

Calculates the functional MSE for a fitted FDboost-object

Usage

```
funMSE(
  object,
  overTime = TRUE,
  breaks = object$yind,
  global = FALSE,
  relative = FALSE,
  root = FALSE,
  ...
)
```

Arguments

`object`	fitted FDboost-object
`overTime`	per default the functional R-squared is calculated over time if `overTime=FALSE`, the R-squared is calculated per curve
`breaks`	an optional vector or number giving the time-points at which the model is evaluated. Can be specified as number of equidistant time-points or as vector of time-points. Defaults to the index of the response in the model.
`global`	logical. defaults to `FALSE`, if TRUE the global R-squared like in a normal linear model is calculated
`relative`	logical. defaults to `FALSE`. If `TRUE` the MSE is standardized by the global variance of the response $n^{-1} \int \sum_i (Y_i(t) - \bar{Y})^2 dt \approx G^{-1} n^{-1} \sum_g \sum_i (Y_i(t_g) - \bar{Y})^2$
`root`	take the square root of the MSE
`...`	currently not used

Details

Formula to calculate MSE over time, `overTime=TRUE`:
$MSE(t) = n^{-1} \sum_i (Y_i(t) - \hat{Y}_i(t))^2$

Formula to calculate MSE over subjects, `overTime=FALSE`:
$MSE_i = \int (Y_i(t) - \hat{Y}_i(t))^2 dt \approx G^{-1} \sum_g (Y_i(t_g) - \hat{Y}_i(t_g))^2$

Value

Returns a vector with the calculated MSE and some extra information in attributes.

Note

`breaks` cannot be changed in the case the `bsignal()` is used over the same domain as the response! In that case you would have to rename the index of the response or that of the covariates.

---

funplot *Plot functional data with linear interpolation of missing values*

Description

Plot functional data with linear interpolation of missing values

Usage

```
funplot(x, y, id = NULL, rug = TRUE, ...)
```

Arguments

`x`	optional, time-vector for plotting
`y`	matrix of functional data with functions in rows and measured times in columns; or vector or functional observations, in this case id has to be specified
`id`	defaults to NULL for y matrix, is id-variables for y in long format
`rug`	logical. Should rugs be plotted? Defaults to TRUE.
`...`	further arguments passed to `matplot`.

Details

All observations are marked by a small cross (`pch=3`). Missing values are imputed by linear interpolation. Parts that are interpolated are plotted by dotted lines, parts with non-missing values as solid lines.

Value

see `matplot`

Examples

```
### examples for regular data in wide format
data(viscosity)
with(viscosity, funplot(timeAll, visAll, pch=20))
if(require(fda)){
  with(fda::growth, funplot(age, t(hgtm)))
}
```

`funRsquared` *Functional R-squared*

Description

Calculates the functional R-squared for a fitted FDboost-object

Usage

```
funRsquared(object, overTime = TRUE, breaks = object$yind, global = FALSE, ...)
```

Arguments

`object`	fitted FDboost-object
`overTime`	per default the functional R-squared is calculated over time if `overTime=FALSE`, the R-squared is calculated per curve
`breaks`	an optional vector or number giving the time-points at which the model is evaluated. Can be specified as number of equidistant time-points or as vector of time-points. Defaults to the index of the response in the model.
`global`	logical. defaults to `FALSE`, if TRUE the global R-squared like in a normal linear model is calculated
`...`	currently not used

Details

`breaks` should be set to some grid, if there are many missing values or time-points with very few observations in the dataset. Otherwise at these points of t the variance will be almost 0 (or even 0 if there is only one observation at a time-point), and then the prediction by the local means $\mu(t)$ is locally very good. The observations are interpolated linearly if necessary.

Formula to calculate R-squared over time, `overTime=TRUE`:
$R^2(t) = 1 - \sum_i (Y_i(t) - \hat{Y}_i(t))^2 / \sum_i (Y_i(t) - \bar{Y}(t))^2$

Formula to calculate R-squared over subjects, `overTime=FALSE`:
$R_i^2 = 1 - \int (Y_i(t) - \hat{Y}_i(t))^2 dt / \int (Y_i(t) - \bar{Y}_i)^2 dt$

Value

Returns a vector with the calculated R-squared and some extra information in attributes.

Note

breaks cannot be changed in the case the bsignal() is used over the same domain as the response! In that case you would have to rename the index of the response or that of the covariates.

References

Ramsay, J., Silverman, B. (2006). Functional data analysis. Wiley Online Library. chapter 16.3

---

getTime *Generic functions to asses attributes of functional data objects*

Description

Extract attributes of an object.

Usage

```
getTime(object)

getId(object)

getX(object)

getArgvals(object)

getTimeLab(object)

getIdLab(object)

getXLab(object)

getArgvalsLab(object)
```

Arguments

object	an R-object, currently implemented for hmatrix and fmatrix

Details

Extract the time variable getTime, the idgetId, the functional covariate getX, its argument values getArgvals. Or the names of the different variables getTimeLab, getIdLab, getXLab, getArgvalsLab.

Value

properties of a hmatrix or fmatrix

See Also

hmatrix for the h.atrix class.

getTime.hmatrix *Extract attributes of hmatrix*

Description

Extract attributes of an object of class hmatrix.

Usage

```
## S3 method for class 'hmatrix'
getTime(object)

## S3 method for class 'hmatrix'
getId(object)

## S3 method for class 'hmatrix'
getX(object)

## S3 method for class 'hmatrix'
getArgvals(object)

## S3 method for class 'hmatrix'
getTimeLab(object)

## S3 method for class 'hmatrix'
getIdLab(object)

## S3 method for class 'hmatrix'
getXLab(object)

## S3 method for class 'hmatrix'
getArgvalsLab(object)
```

Arguments

object	object of class hmatrix

Details

Extract the time variable getTime, the idgetId, the functional covariate getX, its argument values getArgvals. Or the names of the different variables getTimeLab, getIdLab, getXLab, getArgvalsLab for an object of class hmatrix.

Value

properties of a hmatrix

`hmatrix` *A S3 class for univariate functional data on a common grid*

Description

The hmatrix class represents data for a functional historical effect. The class is basically a matrix containing the time and the id for the observations of the functional response. The functional covariate is contained as attribute.

Usage

```
hmatrix(
  time,
  id,
  x,
  argvals = seq_len(ncol(x)),
  timeLab = "t",
  idLab = "wideIndex",
  xLab = "x",
  argvalsLab = "s"
)
```

Arguments

`time`	set of argument values of the response in long format, i.e. at which `t` the response curve is observed
`id`	specify to which curve the point belongs to, id from 1, 2, ..., n.
`x`	matrix of functional covariate, each trajectory is in one row
`argvals`	set of argument values, i.e., the common gird at which the functional covariate is observed, by default `seq_len(ncol(x))`
`timeLab`	name of the time axis, by default `t`
`idLab`	name of the id variable, by default `wideIndex`
`xLab`	name of the functional variable, by default NULL
`argvalsLab`	name of the argument for the covariate by default `s`

Details

In the hmatrix class the id has to run from i=1, 2, ..., n including all integers from 1 to n. The rows of the functional covariate x correspond to those observations.

Value

An matrix object of type `"hmatrix"`

See Also

getTime.hmatrix to extract attributes, and ?"[.hmatrix" for the extract method.

Examples

```
## Example for a hmatrix object
t1 <- rep((1:5)/2, each = 3)
id1 <- rep(1:3, 5)
x1 <- matrix(1:15, ncol = 5)
s1 <- (1:5)/2
myhmatrix <- hmatrix(time = t1, id = id1, x = x1, argvals = s1,
                     timeLab = "t1", argvalsLab = "s1", xLab = "test")

# extract with [ keeps attributes
# select observations of subjects 2 and 3
myhmatrixSub <- myhmatrix[id1 %in% c(2, 3), ]
str(myhmatrixSub)
getX(myhmatrixSub)
getX(myhmatrix)

# get time
myhmatrix[ , 1] # as column matrix as drop = FALSE
getTime(myhmatrix) # as vector

# get id
myhmatrix[ , 2] # as column matrix as drop = FALSE
getId(myhmatrix) # as vector

# subset hmatrix on the basis of an index, which is defined on the curve level
reweightData(data = list(hmat = myhmatrix), vars = "hmat", index = c(1, 1, 2))
# this keeps only the unique x values in attr(,'x') but multiplies the corresponding
# ids and times in the time id matrix
# for bhistx baselearner, there may be an additional id variable for the tensor product
newdat <- reweightData(data = list(hmat = myhmatrix,
  repIDx = rep(seq_len(nrow(attr(myhmatrix,'x'))), length(attr(myhmatrix,"argvals")))),
  vars = "hmat", index = c(1,1,2), idvars="repIDx")
length(newdat$repIDx)

## use hmatrix within a data.frame
mydat <- data.frame(I(myhmatrix), z=rnorm(3)[id1])
str(mydat)
str(mydat[id1 %in% c(2, 3), ])
str(myhmatrix[id1 %in% c(2, 3), ])
```

---

integrationWeights *Functions to compute integration weights*

Description

Computes trapezoidal integration weights (Riemann sums) for a functional variable `X1` that has evaluation points `xind`.

Usage

```
integrationWeights(X1, xind, id = NULL)

integrationWeightsLeft(X1, xind, leftWeight = c("first", "mean", "zero"))
```

Arguments

`X1`	for functional data that is observed on one common grid, a matrix containing the observations of the functional variable. For a functional variable that is observed on curve specific grids, a long vector.
`xind`	evaluation points (index) of functional variable
`id`	defaults to `NULL`. Only necessary for response in long format. In this case `id` specifies which curves belong together.
`leftWeight`	one of `c("mean", "first", "zero")`. With left Riemann sums different assumptions for the weight of the first observation are possible. The default is to use the mean over all integration weights, `"mean"`. Alternatively one can use the first integration weight, `"first"`, or use the distance to zero, `"zero"`.

Details

The function `integrationWeights()` computes trapezoidal integration weights, that are symmetric. Per default those weights are used in the `bsignal`-base-learner. In the special case of evaluation points (`xind`) with equal distances, all integration weights are equal.

The function `integrationWeightsLeft()` computes weights, that take into account only the distance to the prior observation point. Thus one has to decide what to do with the first observation. The left weights are adequate for historical effects like in `bhist`.

Value

Matrix with integration

See Also

`bsignal` and `bhist` for the base-learners.

Examples

```
## Example for trapezoidal integration weights
xind0 <- seq(0,1,l = 5)
xind <- c(0, 0.1, 0.3, 0.7, 1)
X1 <- matrix(xind^2, ncol = length(xind0), nrow = 2)

# Regualar observation points
integrationWeights(X1, xind0)
```

```
# Irregular observation points
integrationWeights(X1, xind)

# with missing value
X1[1,2] <- NA
integrationWeights(X1, xind0)
integrationWeights(X1, xind)

## Example for left integration weights
xind0 <- seq(0,1,l = 5)
xind <- c(0, 0.1, 0.3, 0.7, 1)
X1 <- matrix(xind^2, ncol = length(xind0), nrow = 2)

# Regular observation points
integrationWeightsLeft(X1, xind0, leftWeight = "mean")
integrationWeightsLeft(X1, xind0, leftWeight = "first")
integrationWeightsLeft(X1, xind0, leftWeight = "zero")

# Irregular observation points
integrationWeightsLeft(X1, xind, leftWeight = "mean")
integrationWeightsLeft(X1, xind, leftWeight = "first")
integrationWeightsLeft(X1, xind, leftWeight = "zero")

# obervation points that do not start with 0
xind2 <- xind + 0.5
integrationWeightsLeft(X1, xind2, leftWeight = "zero")
```

`is.hmatrix` *Test to class of hmatrix*

Description

is.hmatrix tests if its argument is an object of class hmatrix.

Usage

```
is.hmatrix(object)
```

Arguments

`object`	object of class hmatrix

Value

logical value

`mstop.validateFDboost` *Methods for objects of class validateFDboost*

Description

Methods for objects that are fitted to determine the optimal mstop and the prediction error of a model fitted by FDboost.

Usage

```
## S3 method for class 'validateFDboost'
mstop(object, riskopt = c("mean", "median"), ...)

## S3 method for class 'validateFDboost'
print(x, ...)

## S3 method for class 'validateFDboost'
plot(
  x,
  riskopt = c("mean", "median"),
  ylab = attr(x, "risk"),
  xlab = "Number of boosting iterations",
  ylim = range(x$oobrisk),
  which = 1,
  modObject = NULL,
  predictNA = FALSE,
  names.arg = NULL,
  ask = TRUE,
  ...
)

plotPredCoef(
  x,
  which = NULL,
  pers = TRUE,
  commonRange = TRUE,
  showNumbers = FALSE,
  showQuantiles = TRUE,
  ask = TRUE,
  terms = TRUE,
  probs = c(0.25, 0.5, 0.75),
  ylim = NULL,
  ...
)
```

Arguments

`object`	object of class `validateFDboost`

riskopt	how the risk is minimized to obtain the optimal stopping iteration; defaults to the mean, can be changed to the median.
...	additional arguments passed to callies.
x	an object of class validateFDboost.
ylab	label for y-axis
xlab	label for x-axis
ylim	values for limits of y-axis
which	In the case of plotPredCoef() the subset of base-learners to take into account for plotting. In the case of plot.validateFDboost() the diagnostic plots that are given (1: empirical risk per fold as a funciton of the boosting iterations, 2: empirical risk per fold, 3: MRD per fold, 4: observed and predicted values, 5: residuals; 2-5 for the model with the optimal number of boosting iterations).
modObject	if the original model object of class FDboost is given predicted values of the whole model can be compared to the predictions of the cross-validated models
predictNA	should missing values in the response be predicted? Defaults to FALSE.
names.arg	names of the observed curves
ask	defaults to TRUE, ask for next plot using par(ask = ask) ?
pers	plot coefficient surfaces as persp-plots? Defaults to TRUE.
commonRange	plot predicted coefficients on a common range, defaults to TRUE.
showNumbers	show number of curve in plot of predicted coefficients, defaults to FALSE
showQuantiles	plot the 0.05 and the 0.95 Quantile of coefficients in 1-dim effects.
terms	logical, defaults to TRUE; plot the added terms (default) or the coefficients?
probs	vector of quantiles to be used in the plotting of 2-dimensional coefficients surfaces, defaults to probs = c(0.25, 0.5, 0.75)

Details

The function mstop.validateFDboost extracts the optimal mstop by minimizing the mean (or the median) risk. plot.validateFDboost plots cross-validated risk, RMSE, MRD, measured and predicted values and residuals as determined by validateFDboost. The function plotPredCoef plots the coefficients that were estimated in the folds - only possible if the argument getCoefCV is TRUE in the call to validateFDboost.

Value

No return value (plot method) or the object itself (print method)

o_control *Function to control estimation of smooth offset*

Description

Function to control estimation of smooth offset

Usage

```
o_control(k_min = 20, rule = 2, silent = TRUE, cyclic = FALSE, knots = NULL)
```

Arguments

k_min	maximal number of k in s()
rule	which rule to use in approx() of the response before calculating the global mean, rule=1 means no extrapolation, rule=2 means to extrapolate the closest non-missing value, see approx
silent	print error messages of model fit?
cyclic	defaults to FALSE, if TRUE cyclic splines are used
knots	arguments knots passed to gam

Value

a list with controls

plot.bootstrapCI *Methods for objects of class bootstrapCI*

Description

Methods for objects that are fitted to compute bootstrap confidence intervals.

Usage

```
## S3 method for class 'bootstrapCI'
plot(
  x,
  which = NULL,
  pers = TRUE,
  commonRange = TRUE,
  showNumbers = FALSE,
  showQuantiles = TRUE,
  ask = TRUE,
  probs = c(0.25, 0.5, 0.75),
```

```
  ylim = NULL,
  ...
)

## S3 method for class 'bootstrapCI'
print(x, ...)
```

Arguments

x	an object of class bootstrapCI.
which	base-learners that are plotted
pers	plot coefficient surfaces as persp-plots? Defaults to TRUE.
commonRange	plot predicted coefficients on a common range, defaults to TRUE.
showNumbers	show number of curve in plot of predicted coefficients, defaults to FALSE
showQuantiles	plot the 0.05 and the 0.95 Quantile of coefficients in 1-dim effects.
ask	defaults to TRUE, ask for next plot using par(ask = ask)?
probs	vector of quantiles to be used in the plotting of 2-dimensional coefficients surfaces, defaults to probs = c(0.25, 0.5, 0.75)
ylim	values for limits of y-axis
...	additional arguments passed to callies.

Details

plot.bootstrapCI plots the bootstrapped coefficients.

Value

No return value (plot method) or x itself (print method)

---

plot.FDboost *Plot the fit or the coefficients of a boosted functional regression model*

Description

Takes a fitted FDboost-object produced by FDboost() and plots the fitted effects or the coefficient-functions/surfaces.

Usage

```
## S3 method for class 'FDboost'
plot(
  x,
  raw = FALSE,
  rug = TRUE,
  which = NULL,
```

```
  includeOffset = TRUE,
  ask = TRUE,
  n1 = 40,
  n2 = 40,
  n3 = 20,
  n4 = 11,
  onlySelected = TRUE,
  pers = FALSE,
  commonRange = FALSE,
  ...
)

plotPredicted(
  x,
  subset = NULL,
  posLegend = "topleft",
  lwdObs = 1,
  lwdPred = 1,
  ...
)

plotResiduals(x, subset = NULL, posLegend = "topleft", ...)
```

Arguments

x	a fitted FDboost-object
raw	logical defaults to FALSE. If raw = FALSE for each effect the estimated function/surface is calculated. If raw = TRUE the coefficients of the model are returned.
rug	when TRUE (default) then the covariate to which the plot applies is displayed as a rug plot at the foot of each plot of a 1-d smooth, and the locations of the covariates are plotted as points on the contour plot representing a 2-d smooth.
which	a subset of base-learners to take into account for plotting.
includeOffset	logical, defaults to TRUE. Should the offset be included in the plot of the intercept (default) or should it be plotted separately.
ask	logical, defaults to TRUE, if several effects are plotted the user has to hit Return to see next plot.
n1	see below
n2	see below
n3	n1, n2, n3 give the number of grid-points for 1-/2-/3-dimensional smooth terms used in the marginal equidistant grids over the range of the covariates at which the estimated effects are evaluated.
n4	gives the number of points for the third dimension in a 3-dimensional smooth term
onlySelected	logical, defaults to TRUE. Only plot effects that where selected in at least one boosting iteration.

pers	logical, defaults to FALSE, If TRUE, perspective plots (persp) for 2- and 3-dimensional effects are drawn. If FALSE, image/contour-plots (image, contour) are drawn for 2- and 3-dimensional effects.
commonRange	logical, defaults to FALSE, if TRUE the range over all effects is the same (does not affect perspecitve or image plots).
...	other arguments, passed to funplot (only used in plotPredicted)
subset	subset of the observed response curves and their predictions that is plotted. Per default all observations are plotted.
posLegend	location of the legend, if a legend is drawn automatically (only used in plotPredicted). The default is "topleft".
lwdObs	lwd of observed curves (only used in plotPredicted)
lwdPred	lwd of predicted curves (only used in plotPredicted)

Value

no return value (plot method)

See Also

FDboost for the model fit and coef.FDboost for the calculation of the coefficient functions.

---

predict.FDboost *Prediction for boosted functional regression model*

Description

Takes a fitted FDboost-object produced by FDboost() and produces predictions given a new set of values for the model covariates or the original values used for the model fit. This is a wrapper function for predict.mboost()

Usage

```
## S3 method for class 'FDboost'
predict(object, newdata = NULL, which = NULL, toFDboost = TRUE, ...)
```

Arguments

object	a fitted FDboost-object
newdata	a named list or a data frame containing the values of the model covariates at which predictions are required. If this is not provided then predictions corresponding to the original data are returned. If newdata is provided then it should contain all the variables needed for prediction, in the format supplied to FDboost, i.e., functional predictors must be supplied as matrices with each row corresponding to one observed function.

which	a subset of base-learners to take into account for computing predictions or coefficients. If which is given (as an integer vector corresponding to base-learners) a list is returned.
toFDboost	logical, defaults to TRUE. In case of regular response in wide format (i.e. response is supplied as matrix): should the predictions be returned as matrix, or list of matrices instead of vectors
...	additional arguments passed on to predict.mboost().

Value

a matrix or list of predictions depending on values of unlist and which

See Also

FDboost for the model fit and plotPredicted for a plot of the observed values and their predictions.

---

predict.FDboost_fac *Prediction and plotting for factorized FDboost model components*

Description

Prediction and plotting for factorized FDboost model components

Usage

```
## S3 method for class 'FDboost_fac'
predict(object, newdata = NULL, which = NULL, ...)

## S3 method for class 'FDboost_fac'
plot(x, which = NULL, main = NULL, ...)
```

Arguments

object, x	a model-factor given as a FDboost_fac object
newdata	optionally, a data frame or list in which to look for variables with which to predict. See predict.mboost.
which	a subset of base-learner components to take into account for computing predictions or coefficients. Different components are never aggregated to a joint prediction, but always returned as a matrix or list. Select the k-th component by name in the format bl(x, ...)[k] or all components of a base-learner by dropping the index or all base-learners of a variable by using the variable name.
...	additional arguments passed to underlying methods.
main	the plot title. By default, base-learner names are used with component numbers [k].

Value

A matrix of predictions (for predict method) or no return value (plot method)

See Also

[factorize(), factorize.FDboost()]

`residuals.FDboost` *Residual values of a boosted functional regression model*

Description

Takes a fitted `FDboost`-object and computes the residuals, more precisely the current value of the negative gradient is returned.

Usage

```
## S3 method for class 'FDboost'
residuals(object, ...)
```

Arguments

`object`	a fitted `FDboost`-object
`...`	not used

Details

The residual is missing if the corresponding value of the response was missing.

Value

matrix of residual values

See Also

`FDboost` for the model fit.

reweightData *Function to Reweight Data*

Description

Function to Reweight Data

Usage

```
reweightData(
  data,
  argvals,
  vars,
  longvars = NULL,
  weights,
  index,
  idvars = NULL,
  compress = FALSE
)
```

Arguments

data	a named list or data.frame.
argvals	character (vector); name(s) for entries in data giving the index for observed grid points; must be supplied if vars is not supplied.
vars	character (vector); name(s) for entries in data, which are subsetted according to weights or index. Must be supplied if argvals is not supplied.
longvars	variables in long format, e.g., a response that is observed at curve specific grids.
weights	vector of weights for observations. Must be supplied if index is not supplied.
index	vector of indices for observations. Must be supplied if weights is not supplied.
idvars	character (vector); index, which is needed to expand vars to be conform with the hmatrix structure when using bhistx-base-learners or to be conform with variables in long format specified in longvars.
compress	logical; whether hmatrix objects are saved in compressed form or not. Default is TRUE. Should be set to FALSE when using reweightData for nested resampling.

Details

reweightData indexes the rows of matrices and / or positions of vectors by using either the index or the weights-argument. To prevent the function from indexing the list entry / entries, which serve as time index for observed grid points of each trajectory of functional observations, the argvals argument (vector of character names for these list entries) can be supplied. If argvals is not supplied, vars must be supplied and it is assumed that argvals is equal to names(data)[!names(data) %in% vars].

When using `weights`, a weight vector of length N must be supplied, where N is the number of observations. When using `index`, the vector must contain the index of each row as many times as it shall be included in the new data set.

Value

A list with the reweighted or subsetted data.

Author(s)

David Ruegamer, Sarah Brockhaus

Examples

```
## load data
data("viscosity", package = "FDboost")
interval <- "101"
end <- which(viscosity$timeAll == as.numeric(interval))
viscosity$vis <- log(viscosity$visAll[ , 1:end])
viscosity$time <- viscosity$timeAll[1:end]

## what does data look like
str(viscosity)

## do some reweighting
# correct weights
str(reweightData(viscosity, vars=c("vis", "T_C", "T_A", "rspeed", "mflow"),
    argvals = "time", weights = c(0, 32, 32, rep(0, 61))))

str(visNew <- reweightData(viscosity, vars=c("vis", "T_C", "T_A", "rspeed", "mflow"),
    argvals = "time", weights = c(0, 32, 32, rep(0, 61))))
# check the result
# visNew$vis[1:5, 1:5] ## image(visNew$vis)

# incorrect weights
str(reweightData(viscosity, vars=c("vis", "T_C", "T_A", "rspeed", "mflow"),
    argvals = "time", weights = sample(1:64, replace = TRUE)), 1)

# supply meaningful index
str(visNew <- reweightData(viscosity, vars = c("vis", "T_C", "T_A", "rspeed", "mflow"),
              argvals = "time", index = rep(1:32, each = 2)))
# check the result
# visNew$vis[1:5, 1:5]

# errors
if(FALSE){
   reweightData(viscosity, argvals = "")
   reweightData(viscosity, argvals = "covThatDoesntExist", index = rep(1,64))
   }
```

`stabsel.FDboost` *Stability Selection*

Description

Function for stability selection with functional response. Per default the sampling is done on the level of curves and if the model contains a smooth functional intercept, this intercept is refittedn in each sampling fold.

Usage

```
## S3 method for class 'FDboost'
stabsel(
  x,
  refitSmoothOffset = TRUE,
  cutoff,
  q,
  PFER,
 folds = cvLong(x$id, weights = rep(1, l = length(x$id)), type = "subsampling", B = B),
  B = ifelse(sampling.type == "MB", 100, 50),
  assumption = c("unimodal", "r-concave", "none"),
  sampling.type = c("SS", "MB"),
  papply = mclapply,
  verbose = TRUE,
  eval = TRUE,
  ...
)
```

Arguments

`x`	fitted FDboost-object
`refitSmoothOffset`	logical, should the offset be refitted in each learning sample? Defaults to `TRUE`.
`cutoff`	cutoff between 0.5 and 1. Preferably a value between 0.6 and 0.9 should be used.
`q`	number of (unique) selected variables (or groups of variables depending on the model) that are selected on each subsample.
`PFER`	upper bound for the per-family error rate. This specifies the amount of falsely selected base-learners, which is tolerated. See details of `stabsel`.
`folds`	a weight matrix with number of rows equal to the number of observations, see `{cvLong}`. Usually one should not change the default here as subsampling with a fraction of 1/2 is needed for the error bounds to hold. One usage scenario where specifying the folds by hand might be the case when one has dependent data (e.g. clusters) and thus wants to draw clusters (i.e., multiple rows together) not individuals.

B	number of subsampling replicates. Per default, we use 50 complementary pairs for the error bounds of Shah & Samworth (2013) and 100 for the error bound derived in Meinshausen & Buehlmann (2010). As we use B complementary pairs in the former case this leads to 2B subsamples.
assumption	Defines the type of assumptions on the distributions of the selection probabilities and simultaneous selection probabilities. Only applicable for sampling.type = "SS". For sampling.type = "MB" we always use "none".
sampling.type	use sampling scheme of of Shah & Samworth (2013), i.e., with complementary pairs (sampling.type = "SS"), or the original sampling scheme of Meinshausen & Buehlmann (2010).
papply	(parallel) apply function, defaults to mclapply. Alternatively, parLapply can be used. In the latter case, usually more setup is needed (see example of cvrisk for some details).
verbose	logical (default: TRUE) that determines wether warnings should be issued.
eval	logical. Determines whether stability selection is evaluated (eval = TRUE; default) or if only the parameter combination is returned.
...	additional arguments to cvrisk or validateFDboost.

Details

The number of boosting iterations is an important hyper-parameter of the boosting algorithms and can be chosen using the functions cvrisk.FDboost and validateFDboost as they compute honest, i.e. out-of-bag, estimates of the empirical risk for different numbers of boosting iterations. The weights (zero weights correspond to test cases) are defined via the folds matrix, see cvrisk in package mboost. See Hofner et al. (2015) for the combination of stability selection and componentwise boosting.

Value

An object of class stabsel with a special print method. For the elements of the object, see stabsel

References

B. Hofner, L. Boccuto and M. Goeker (2015), Controlling false discoveries in high-dimensional situations: boosting with stability selection. BMC Bioinformatics, 16, 1-17.

N. Meinshausen and P. Buehlmann (2010), Stability selection. Journal of the Royal Statistical Society, Series B, 72, 417-473.

R.D. Shah and R.J. Samworth (2013), Variable selection with error control: another look at stability selection. Journal of the Royal Statistical Society, Series B, 75, 55-80.

See Also

stabsel to perform stability selection for a mboost-object.

Examples

```
######## Example for function-on-scalar-regression
data("viscosity", package = "FDboost")
## set time-interval that should be modeled
interval <- "101"

## model time until "interval" and take log() of viscosity
end <- which(viscosity$timeAll == as.numeric(interval))
viscosity$vis <- log(viscosity$visAll[,1:end])
viscosity$time <- viscosity$timeAll[1:end]
# with(viscosity, funplot(time, vis, pch = 16, cex = 0.2))

## fit a model cotaining all main effects
modAll <- FDboost(vis ~ 1
          + bolsc(T_C, df=1) %A0% bbs(time, df=5)
          + bolsc(T_A, df=1) %A0% bbs(time, df=5)
          + bolsc(T_B, df=1) %A0% bbs(time, df=5)
          + bolsc(rspeed, df=1) %A0% bbs(time, df=5)
          + bolsc(mflow, df=1) %A0% bbs(time, df=5),
       timeformula = ~bbs(time, df=5),
       numInt = "Riemann", family = QuantReg(),
       offset = NULL, offset_control = o_control(k_min = 10),
       data = viscosity,
       control = boost_control(mstop = 100, nu = 0.2))


## create folds for stability selection
## only 5 folds for a fast example, usually use 50 folds
set.seed(1911)
folds <- cvLong(modAll$id, weights = rep(1, l = length(modAll$id)),
                type = "subsampling", B = 5)


## stability selection with refit of the smooth intercept
stabsel_parameters(q = 3, PFER = 1, p = 6, sampling.type = "SS")
sel1 <- stabsel(modAll, q = 3, PFER = 1, folds = folds, grid = 1:200, sampling.type = "SS")
sel1

## stability selection without refit of the smooth intercept
sel2 <- stabsel(modAll, refitSmoothOffset = FALSE, q = 3, PFER = 1,
                folds = folds, grid = 1:200, sampling.type = "SS")
sel2
```

`subset_hmatrix` *Subsets hmatrix according to an index*

Description

Subsets hmatrix according to an index

Usage

```
subset_hmatrix(x, index, compress = TRUE)
```

Arguments

x	hmatix object that should be subsetted
index	integer vector with (possibly duplicated) indices for each curve to select
compress	logical, defaults to TRUE. Only used to force a meaningful behaviour of applyFolds with hmatrix objects when using nested resampling.

Details

This methods is primary useful when subsetting repeatedly.

Value

a hmatrix object

Examples

```
t1 <- rep((1:5)/2, each = 3)
id1 <- rep(1:3, 5)
x1 <- matrix(1:15, ncol = 5)
s1 <- (1:5)/2
hmat <- hmatrix(time = t1, id = id1, x = x1, argvals = s1, timeLab = "t1",
                argvalsLab = "s1", xLab = "test")

index1 <- c(1, 1, 3)
index2 <- c(2, 3, 3)
resMat <- subset_hmatrix(hmat, index = index1)
try(resMat2 <- subset_hmatrix(resMat, index = index2))
resMat <- subset_hmatrix(hmat, index = index1, compress = FALSE)
try(resMat2 <- subset_hmatrix(resMat, index = index2))
```

---

summary.FDboost *Print and summary of a boosted functional regression model*

Description

Takes a fitted FDboost-object and produces a print to the console or a summary.

Usage

```
## S3 method for class 'FDboost'
summary(object, ...)

## S3 method for class 'FDboost'
print(x, ...)
```

Arguments

object	a fitted FDboost-object
...	currently not used
x	a fitted FDboost-object

Value

a list with information on the model / a list with summary information

See Also

FDboost for the model fit.

---

truncateTime *Function to truncate time in functional data*

Description

Function to truncate time in functional data

Usage

```
truncateTime(funVar, time, newtime, data)
```

Arguments

funVar	names of functional variables that should be truncated
time	name of time variable
newtime	new time vector that should be used. Must be part of the old time-line.
data	list containing all the data

Value

A list with the data containing all variables of the original dataset with the variables of funVar truncated according to newtime.

Note

All variables that are not part if funVar, or time are simply copied into the new data list

Examples

```
if(require(fda)){
  dat <- fda::growth
  dat$hgtm <- t(dat$hgtm[,1:10])
  dat$hgtf <- t(dat$hgtf[,1:10])

  ## only use time-points 1:16 of variable age
  datTr <- truncateTime(funVar=c("hgtm","hgtf"), time="age", newtime=1:16, data=dat)


  oldpar <- par(mfrow=c(1,2))
  with(dat, funplot(age, hgtm, main="Original data"))
  with(datTr, funplot(age, hgtm, main="Yearly data"))
  par(mfrow=c(1,1))
  par(oldpar)

}
```

`update.FDboost` *Function to update FDboost objects*

Description

Function to update FDboost objects

Usage

```
## S3 method for class 'FDboost'
update(
  object,
  weights = NULL,
  oobweights = NULL,
  risk = NULL,
  trace = NULL,
  ...,
  evaluate = TRUE
)
```

Arguments

`object`	fitted FDboost-object
`weights, oobweights, risk, trace`	see `?FDboost`
`...`	Additional arguments to the call, or arguments with changed values.
`evaluate`	If true evaluate the new call else return the call.

Value

Returns the call of (`evaluate = FALSE`) or the updated (`evaluate = TRUE`) FDboost model

Author(s)

David Ruegamer

Examples

```
######## Example from \code{?FDboost}
data("viscosity", package = "FDboost")
## set time-interval that should be modeled
interval <- "101"

## model time until "interval" and take log() of viscosity
end <- which(viscosity$timeAll == as.numeric(interval))
viscosity$vis <- log(viscosity$visAll[,1:end])
viscosity$time <- viscosity$timeAll[1:end]
# with(viscosity, funplot(time, vis, pch = 16, cex = 0.2))

mod1 <- FDboost(vis ~ 1 + bolsc(T_C, df = 2) + bolsc(T_A, df = 2),
                timeformula = ~ bbs(time, df = 4),
                numInt = "equal", family = QuantReg(),
                offset = NULL, offset_control = o_control(k_min = 9),
                data = viscosity, control=boost_control(mstop = 10, nu = 0.4))

# update nu
mod2 <- update(mod1, control=boost_control(nu = 1)) # mstop will stay the same
# update mstop
mod3 <- update(mod2, control=boost_control(mstop = 100)) # nu=1 does not get changed
mod4 <- update(mod1, formula = vis ~ 1 + bolsc(T_C, df = 2)) # drop one term
```

---

validateFDboost *Cross-Validation and Bootstrapping over Curves*

Description

DEPRECATED! The function `validateFDboost()` is deprecated, use `applyFolds` and `bootstrapCI` instead.

Usage

```
validateFDboost(
  object,
  response = NULL,
  folds = cv(rep(1, length(unique(object$id))), type = "bootstrap"),
  grid = 1:mstop(object),
  fun = NULL,
  getCoefCV = TRUE,
```

```
  riskopt = c("mean", "median"),
  mrdDelete = 0,
  refitSmoothOffset = TRUE,
  showProgress = TRUE,
  ...
)
```

Arguments

object	fitted FDboost-object
response	optional, specify a response vector for the computation of the prediction errors. Defaults to NULL which means that the response of the fitted model is used.
folds	a weight matrix with number of rows equal to the number of observed trajectories.
grid	the grid over which the optimal number of boosting iterations (mstop) is searched.
fun	if fun is NULL, the out-of-bag risk is returned. fun, as a function of object, may extract any other characteristic of the cross-validated models. These are returned as is.
getCoefCV	logical, defaults to TRUE. Should the coefficients and predictions be computed for all the models on the sampled data?
riskopt	how is the optimal stopping iteration determined. Defaults to the mean, but median is possible as well.
mrdDelete	Delete values that are mrdDelete percent smaller than the mean of the response. Defaults to 0 which means that only response values being 0 are not used in the calculation of the MRD (= mean relative deviation).
refitSmoothOffset	logical, should the offset be refitted in each learning sample? Defaults to TRUE. In cvrisk the offset of the original model fit in object is used in all folds.
showProgress	logical, defaults to TRUE.
...	further arguments passed to mclapply

Details

The number of boosting iterations is an important hyper-parameter of boosting and can be chosen using the function validateFDboost as they compute honest, i.e., out-of-bag, estimates of the empirical risk for different numbers of boosting iterations.

The function validateFDboost is especially suited to models with functional response. Using the option refitSmoothOffset the offset is refitted on each fold. Note, that the function validateFDboost expects folds that give weights per curve without considering integration weights. The integration weights of object are used to compute the empirical risk as integral. The argument response can be useful in simulation studies where the true value of the response is known but for the model fit the response is used with noise.

Value

The function `validateFDboost` returns a `validateFDboost`-object, which is a named list containing:

`response`	the response
`yind`	the observation points of the response
`id`	the id variable of the response
`folds`	folds that were used
`grid`	grid of possible numbers of boosting iterations
`coefCV`	if `getCoefCV` is `TRUE` the estimated coefficient functions in the folds
`predCV`	if `getCoefCV` is `TRUE` the out-of-bag predicted values of the response
`oobpreds`	if the type of folds is curves the out-of-bag predictions for each trajectory
`oobrisk`	the out-of-bag risk
`oobriskMean`	the out-of-bag risk at the minimal mean risk
`oobmse`	the out-of-bag mean squared error (MSE)
`oobrelMSE`	the out-of-bag relative mean squared error (relMSE)
`oobmrd`	the out-of-bag mean relative deviation (MRD)
`oobrisk0`	the out-of-bag risk without consideration of integration weights
`oobmse0`	the out-of-bag mean squared error (MSE) without consideration of integration weights
`oobmrd0`	the out-of-bag mean relative deviation (MRD) without consideration of integration weights
`format`	one of "FDboostLong" or "FDboost" depending on the class of the object
`fun_ret`	list of what fun returns if fun was specified

Examples

```
if(require(fda)){
 ## load the data
 data("CanadianWeather", package = "fda")

 ## use data on a daily basis
 canada <- with(CanadianWeather,
                list(temp = t(dailyAv[ , , "Temperature.C"]),
                     l10precip = t(dailyAv[ , , "log10precip"]),
               l10precip_mean = log(colMeans(dailyAv[ , , "Precipitation.mm"]), base = 10),
                     lat = coordinates[ , "N.latitude"],
                     lon = coordinates[ , "W.longitude"],
                     region = factor(region),
                     place = factor(place),
                  day = 1:365,  ## corresponds to t: evaluation points of the fun. response
              day_s = 1:365))  ## corresponds to s: evaluation points of the fun. covariate

## center temperature curves per day
canada$tempRaw <- canada$temp
```

```
canada$temp <- scale(canada$temp, scale = FALSE)
rownames(canada$temp) <- NULL ## delete row-names

## fit the model
mod <- FDboost(l10precip ~ 1 + bolsc(region, df = 4) +
                bsignal(temp, s = day_s, cyclic = TRUE, boundary.knots = c(0.5, 365.5)),
               timeformula = ~ bbs(day, cyclic = TRUE, boundary.knots = c(0.5, 365.5)),
               data = canada)
mod <- mod[75]

  #### create folds for 3-fold bootstrap: one weight for each curve
  set.seed(124)
  folds_bs <- cv(weights = rep(1, mod$ydim[1]), type = "bootstrap", B = 3)

  ## compute out-of-bag risk on the 3 folds for 1 to 75 boosting iterations
  cvr <- applyFolds(mod, folds = folds_bs, grid = 1:75)

  ## compute out-of-bag risk and coefficient estimates on folds
  cvr2 <- validateFDboost(mod, folds = folds_bs, grid = 1:75)

  ## weights per observation point
  folds_bs_long <- folds_bs[rep(seq_len(nrow(folds_bs)), times = mod$ydim[2]), ]
  attr(folds_bs_long, "type") <- "3-fold bootstrap"
  ## compute out-of-bag risk on the 3 folds for 1 to 75 boosting iterations
  cvr3 <- cvrisk(mod, folds = folds_bs_long, grid = 1:75)

  ## plot the out-of-bag risk
  oldpar <- par(mfrow = c(1,3))
  plot(cvr); legend("topright", lty=2, paste(mstop(cvr)))
  plot(cvr2)
  plot(cvr3); legend("topright", lty=2, paste(mstop(cvr3)))

  ## plot the estimated coefficients per fold
  ## more meaningful for higher number of folds, e.g., B = 100
  par(mfrow = c(2,2))
  plotPredCoef(cvr2, terms = FALSE, which = 1)
  plotPredCoef(cvr2, terms = FALSE, which = 3)

  ## compute out-of-bag risk and predictions for leaving-one-curve-out cross-validation
  cvr_jackknife <- validateFDboost(mod, folds = cvLong(unique(mod$id),
                                   type = "curves"), grid = 1:75)
  plot(cvr_jackknife)
  ## plot oob predictions per fold for 3rd effect
  plotPredCoef(cvr_jackknife, which = 3)
  ## plot coefficients per fold for 2nd effect
  plotPredCoef(cvr_jackknife, which = 2, terms = FALSE)

  par(oldpar)

}
```

viscosity *Viscosity of resin over time*

Description

In an experimental setting the viscosity of resin was measured over time to asses the curing process depending on 5 binary factors (low-high).

Usage

```
data("viscosity")
```

Format

A data list with 64 observations on the following 7 variables.

`visAll` viscosity measures over all available time points

`timeAll` time points of viscosity measures

`T_C` temperature of tools

`T_A` temperature of resin

`T_B` temperature of curing agent

`rspeed` rotational speed

`mflow` mass flow

Details

The aim is to determine factors that affect the curing process in the mold. The desired viscosity-curve has low values in the beginning followed by a sharp increase. Due to technical reasons the measuring method of the rheometer has to be changed in a certain range of viscosity. The first observations are measured by rotation of a blade giving observations every two seconds, the later observations are measured through oscillation of a blade giving observations every ten seconds. In the later observations the resin is quite hard so the measurements should be interpreted as a qualitative measure of hardening.

Source

Wolfgang Raffelt, Technical University of Munich, Institute for Carbon Composites

Examples

```
data("viscosity", package = "FDboost")
## set time-interval that should be modeled
interval <- "101"

## model time until "interval" and take log() of viscosity
end <- which(viscosity$timeAll==as.numeric(interval))
viscosity$vis <- log(viscosity$visAll[,1:end])
```

```
viscosity$time <- viscosity$timeAll[1:end]

## fit median regression model with 100 boosting iterations,
## step-length 0.4 and smooth time-specific offset
## the factors are in effect coding -1, 1 for the levels
mod <- FDboost(vis ~ 1 + bols(T_C, contrasts.arg = "contr.sum", intercept=FALSE)
               + bols(T_A, contrasts.arg = "contr.sum", intercept=FALSE),
               timeformula=~bbs(time, lambda=100),
               numInt="equal", family=QuantReg(),
               offset=NULL, offset_control = o_control(k_min = 9),
               data=viscosity, control=boost_control(mstop = 100, nu = 0.4))
summary(mod)
```

`wide2long` *Transform id and time of wide format into long format*

Description

Transform id and time from wide format into long format, i.e., time and id are repeated accordingly so that two vectors of the same length are returned.

Usage

```
wide2long(time, id)
```

Arguments

`time`	the observation points
`id`	the id for the curve

Value

a list with `time` and `id`

`[.hmatrix` *Extract or replace parts of a hmatrix-object*

Description

Operator acting on hmatrix preserving the attributes when rows are extracted.

Usage

```
## S3 method for class 'hmatrix'
x[i, j, ..., drop = FALSE]
```

Arguments

x	object from which to extract element(s) or in which to replace element(s).
i, j	indices specifying elements to extract or replace. Indices are numeric vectors or empty (missing) or NULL. Numeric values are coerced to integer as by as.integer (and hence truncated towards zero).
...	not used
drop	If TRUE the result is coerced to the lowest possible dimension (or just a matrix). This only works for extracting elements, not for the replacement, defaults to FALSE.

Details

If used on columns or rows/columns a matrix is returned. If used on rows only, i.e. x[i,] an object of class hmatrix is returned. The id is changed so that it runs from 1, ..., nNew, where nNew is the number of different id values in the new hmatrix-object. From the functional covariate x rows are selected accordingly.

Value

a "hmatrix" object

See Also

?"["

%Xc% *Constrained row tensor product*

Description

Combining single base-learners to form new, more complex base-learners, with an identifiability constraint to center the interaction around the intercept and around the two main effects. Suitable for functional response.

Usage

```
bl1 %Xc% bl2
```

Arguments

bl1	base-learner 1, e.g. bols(x1)
bl2	base-learner 2, e.g. bols(x2)

Details

Similar to %X% in package mboost, see %X%, a row tensor product of linear base-learners is returned by %Xc%. %Xc% applies a sum-to-zero constraint to the design matrix suitable for functional response if an interaction of two scalar covariates is specified in the case that the model contains a global intercept and both main effects, as the interaction is centered around the intercept and centered around the two main effects. See Web Appendix A of Brockhaus et al. (2015) for details on how to enforce the constraint for the functional intercept. Use, e.g., in a model call to FDboost, following the scheme, y ~ 1 + bolsc(x1) + bolsc(x2) + bols(x1) %Xc% bols(x2), where 1 induces a global intercept and x1, x2 are factor variables, see Ruegamer et al. (2018).

Value

An object of class blg (base-learner generator) with a dpp function as for other baselearners.

Author(s)

Sarah Brockhaus, David Ruegamer

References

Brockhaus, S., Scheipl, F., Hothorn, T. and Greven, S. (2015): The functional linear array model. Statistical Modelling, 15(3), 279-300.

Ruegamer D., Brockhaus, S., Gentsch K., Scherer, K., Greven, S. (2018). Boosting factor-specific functional historical models for the detection of synchronization in bioelectrical signals. Journal of the Royal Statistical Society: Series C (Applied Statistics), 67, 621-642.

Examples

```
######## Example for function-on-scalar-regression with interaction effect of two scalar covariates
data("viscosity", package = "FDboost")
## set time-interval that should be modeled
interval <- "101"

## model time until "interval" and take log() of viscosity
end <- which(viscosity$timeAll == as.numeric(interval))
viscosity$vis <- log(viscosity$visAll[,1:end])
viscosity$time <- viscosity$timeAll[1:end]
# with(viscosity, funplot(time, vis, pch = 16, cex = 0.2))

## fit model with interaction that is centered around the intercept
## and the two main effects
mod1 <- FDboost(vis ~ 1 + bolsc(T_C, df=1) + bolsc(T_A, df=1) +
                bols(T_C, df=1) %Xc% bols(T_A, df=1),
                timeformula = ~bbs(time, df=6),
                numInt = "equal", family = QuantReg(),
                offset = NULL, offset_control = o_control(k_min = 9),
                data = viscosity, control=boost_control(mstop = 100, nu = 0.4))

## check centering around intercept
colMeans(predict(mod1, which = 4))
```

```
## check centering around main effects
colMeans(predict(mod1, which = 4)[viscosity$T_A == "low", ])
colMeans(predict(mod1, which = 4)[viscosity$T_A == "high", ])
colMeans(predict(mod1, which = 4)[viscosity$T_C == "low", ])
colMeans(predict(mod1, which = 4)[viscosity$T_C == "low", ])

## find optimal mstop using cvrsik() or validateFDboost()
## ...

## look at interaction effect in one plot
# funplot(mod1$yind, predict(mod1, which=4))
```

Index