Package 'mboost'

March 12, 2016

Title Model-Based Boosting

Version 2.6-0

Date 2016-03-11

Description Functional gradient descent algorithm
(boosting) for optimizing general risk functions utilizing
component-wise (penalised) least squares estimates or regression
trees as base-learners for fitting generalized linear, additive
and interaction models to potentially high-dimensional data.

Depends R (>= 2.14.0), methods, stats, parallel, stabs (>= 0.5-0)

Imports Matrix, survival, splines, lattice, nnls, quadprog, utils,
graphics, grDevices, party (>= 1.0-23)

Suggests TH.data, MASS, fields, BayesX, gbm, mlbench, RColorBrewer,
rpart (>= 4.0-3), randomForest, nnet, testthat (>= 0.10.0)

LazyData yes

License GPL-2

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

URL https://github.com/boost-R/mboost,
http://mboost.r-forge.r-project.org

NeedsCompilation yes

Author Torsten Hothorn [aut],
Peter Buehlmann [aut],
Thomas Kneib [aut],
Matthias Schmid [aut],
Benjamin Hofner [aut, cre],
Fabian Sobotka [ctb],
Fabian Scheipl [ctb]

Maintainer Benjamin Hofner <benjamin.hofner@fau.de>

Repository CRAN

Date/Publication 2016-03-12 06:02:17

R topics documented:

`mboost-package` *mboost: Model-Based Boosting*

Description

Functional gradient descent algorithm (boosting) for optimizing general risk functions utilizing component-wise (penalized) least squares estimates or regression trees as base-learners for fitting generalized linear, additive and interaction models to potentially high-dimensional data.

Details

Package:	mboost
Type:	Package
Version:	2.6-0
Date:	2016-03-11
License:	GPL-2
LazyLoad:	yes
LazyData:	yes

This package is intended for modern regression modeling and stands in-between classical generalized linear and additive models, as for example implemented by `lm`, `glm`, or `gam`, and machine-learning approaches for complex interactions models, most prominently represented by `gbm` and `randomForest`.

All functionality in this package is based on the generic implementation of the optimization al-

gorithm (function mboost_fit) that allows for fitting linear, additive, and interaction models (and mixtures of those) in low and high dimensions. The response may be numeric, binary, ordered, censored or count data.

Both theory and applications are discussed by Buehlmann and Hothorn (2007). UseRs without a basic knowledge of boosting methods are asked to read this introduction before analyzing data using this package. The examples presented in this paper are available as package vignette mboost_illustrations.

Note that the model fitting procedures in this package DO NOT automatically determine an appropriate model complexity. This task is the responsibility of the data analyst.

A description of novel features that were introduced in version 2.0 is given in Hothorn et. al (2010).

Hofner et al. (2014) present a comprehensive hands-on tutorial for using the package mboost, which is also available as vignette(package = "mboost", "mboost_tutorial").

Ben Taieba and Hyndman (2013) used this package for fitting their model in the Kaggle Global Energy Forecasting Competition 2012. The corresponding research paper is a good starting point when you plan to analyze your data using mboost.

NEWS in 2.6-series

Series 2.6 includes a lot of bug fixes and improvements. Most notably, the development of the package is now hosted entirely on github in the project boost-R/mboost. Furthermore, the package is now maintained by Benjamin Hofner.

For more changes see
```
news(Version >= "2.6-0", package  = "mboost")
```

NEWS in 2.5-series

Crossvaliation does not stop on errors in single folds anymore an was sped up by setting mc.preschedule = FALSE if parallel computations via mclapply are used. The plot.mboost function is now documented. Values outside the boundary knots are now better handeled (forbidden during fitting, while linear extrapolation is used for prediction). Further perfomance improvements and a lot of bug fixes have been added.

For more changes see
```
news(Version >= "2.5-0", package  = "mboost")
```

NEWS in 2.4-series

Bootstrap confidence intervals have been implemented in the novel confint function. The stability selection procedure has now been moved to a stand-alone package called **stabs**, which now also implements an iterface to use stability selection with other fitting functions. A generic function for "mboost" models is implemented in **mboost**.

For more changes see
```
news(Version >= "2.4-0", package  = "mboost")
```

NEWS in 2.3-series

The stability selection procedure has been completely rewritten and improved. The code base is now extensively tested. New options allow for a less conservative error control.

Constrained effects can now be fitted using quadratic programming methods using the option `type = "quad.prog"` (default) for highly improved speed. Additionally, new constraints have been added.

Other important changes include:

- A new replacement function `mstop(mod) <- i` as an alternative to `mod[i]` was added (as suggested by Achim Zeileis).
- We added new families `Hurdle` and `Multinomial`.
- We added a new argument `stopintern` for internal stopping (based on out-of-bag data) during fitting to `boost_control`.

For more changes see
`news(Version >= "2.3-0", package = "mboost")`

NEWS in 2.2-series

Starting from version 2.2, the default for the degrees of freedom has changed. Now the degrees of freedom are (per default) defined as

$$\mathrm{df}(\lambda) = \mathrm{trace}(2S - S^\top S),$$

with smoother matrix $S = X(X^\top X + \lambda K)^{-1} X$ (see Hofner et al., 2011). Earlier versions used the trace of the smoother matrix $\mathrm{df}(\lambda) = \mathrm{trace}(S)$ as degrees of freedom. One can change the deployed definition using `options(mboost_dftraceS = TRUE)` (see also B. Hofner et al., 2011 and `bols`).

Other important changes include:

- We switched from packages `multicore` and `snow` to `parallel`
- We changed the behavior of `bols(x, intercept = FALSE)` when `x` is a factor: now the intercept is simply dropped from the design matrix and the coding can be specified as usually for factors. Additionally, a new contrast is introduced: `"contr.dummy"` (see `bols` for details).
- We changed the computation of B-spline basis at the boundaries; B-splines now also use equidistant knots in the boundaries (per default).

For more changes see
`news(Version >= "2.2-0" & Version < "2.3-0", package = "mboost")`

NEWS in 2.1-series

In the 2.1 series, we added multiple new base-learners including `bmono` (monotonic effects), `brad` (radial basis functions) and `bmrf` (Markov random fields), and extended `bbs` to incorporate cyclic splines (via argument `cyclic = TRUE`). We also changed the default `df` for `bspatial` to 6.

Starting from this version, we now also automatically center the variables in `glmboost` (argument `center = TRUE`).

For more changes see
`news(Version >= "2.1-0" & Version < "2.2-0", package = "mboost")`

NEWS in 2.0-series

Version 2.0 comes with new features, is faster and more accurate in some aspects. In addition, some changes to the user interface were necessary: Subsetting `mboost` objects changes the object. At each time, a model is associated with a number of boosting iterations which can be changed (increased or decreased) using the subset operator.

The `center` argument in `bols` was renamed to `intercept`. Argument `z` renamed to `by`.

The base-learners `bns` and `bss` are deprecated and replaced by `bbs` (which results in qualitatively the same models but is computationally much more attractive).

New features include new families (for example for ordinal regression) and the `which` argument to the `coef` and `predict` methods for selecting interesting base-learners. Predict methods are much faster now.

The memory consumption could be reduced considerably, thanks to sparse matrix technology in package `Matrix`. Resampling procedures run automatically in parallel on OSes where parallelization via package `parallel` is available.

The most important advancement is a generic implementation of the optimizer in function `mboost_fit`.

For more changes see

```
news(Version >= "2.0-0" & Version < "2.1-0", package  = "mboost")
```

Author(s)

Torsten Hothorn <Torsten.Hothorn@R-project.org>,
Peter Buehlmann, Thomas Kneib, Matthias Schmid and Benjamin Hofner

References

Peter Buehlmann and Torsten Hothorn (2007), Boosting algorithms: regularization, prediction and model fitting. *Statistical Science*, **22**(4), 477–505.

Torsten Hothorn, Peter Buehlmann, Thomas Kneib, Matthias Schmid and Benjamin Hofner (2010), Model-based Boosting 2.0. *Journal of Machine Learning Research*, **11**, 2109–2113.

Benjamin Hofner, Torsten Hothorn, Thomas Kneib, and Matthias Schmid (2011), A framework for unbiased model selection based on boosting. *Journal of Computational and Graphical Statistics*, **20**, 956–971.

Benjamin Hofner, Andreas Mayr, Nikolay Robinzonov and Matthias Schmid (2014). Model-based Boosting in R: A Hands-on Tutorial Using the R Package mboost. *Computational Statistics*, **29**, 3–35.
http://dx.doi.org/10.1007/s00180-012-0382-5

Available as vignette via: `vignette(package = "mboost", "mboost_tutorial")`

Souhaib Ben Taieba and Rob J. Hyndman (2014), A gradient boosting approach to the Kaggle load forecasting competition. *International Journal of Forecasting*, **30**, 382–394.
http://dx.doi.org/10.1016/j.ijforecast.2013.07.005

See Also

The main fitting functions include:
`gamboost` for boosted (generalized) additive models, `glmboost` for boosted linear models and

blackboost for boosted trees.
See there for more details and further links.

Examples

```
data("bodyfat", package = "TH.data")
set.seed(290875)

### model conditional expectation of DEXfat given
model <- mboost(DEXfat ~
    bols(age) +                 ### a linear function of age
    btree(hipcirc, waistcirc) + ### a non-linear interaction of
                                ### hip and waist circumference
    bbs(kneebreadth),           ### a smooth function of kneebreadth
    data = bodyfat, control = boost_control(mstop = 100))

### bootstrap for assessing `optimal' number of boosting iterations
cvm <- cvrisk(model, papply = lapply)

### restrict model to mstop(cvm)
model[mstop(cvm), return = FALSE]
mstop(model)

### plot age and kneebreadth
layout(matrix(1:2, nc = 2))
plot(model, which = c("age", "kneebreadth"))

### plot interaction of hip and waist circumference
attach(bodyfat)
nd <- expand.grid(hipcirc = h <- seq(from = min(hipcirc),
                                to = max(hipcirc),
                                length = 100),
                  waistcirc = w <- seq(from = min(waistcirc),
                                to = max(waistcirc),
                                length = 100))
plot(model, which = 2, newdata = nd)
detach(bodyfat)

### customized plot
layout(1)
pr <- predict(model, which = "hip", newdata = nd)
persp(x = h, y = w, z = matrix(pr, nrow = 100, ncol = 100))
```

---

baselearners *Base-learners for Gradient Boosting*

Description

Base-learners for fitting base-models in the generic implementation of component-wise gradient boosting in function mboost.

Usage

```
## linear base-learner
bols(..., by = NULL, index = NULL, intercept = TRUE, df = NULL,
     lambda = 0, contrasts.arg = "contr.treatment")

## smooth P-spline base-learner
bbs(..., by = NULL, index = NULL, knots = 20, boundary.knots = NULL,
    degree = 3, differences = 2, df = 4, lambda = NULL, center = FALSE,
    cyclic = FALSE, constraint = c("none", "increasing", "decreasing"),
    deriv = 0)

## bivariate P-spline base-learner
bspatial(..., df = 6)

## radial basis functions base-learner
brad(..., by = NULL, index = NULL, knots = 100, df = 4, lambda = NULL,
     covFun = fields::stationary.cov,
     args = list(Covariance="Matern", smoothness = 1.5, theta=NULL))

## random effects base-learner
brandom(..., by = NULL, index = NULL, df = 4, lambda = NULL,
        contrasts.arg = "contr.dummy")

## tree based base-learner
btree(..., tree_controls = party::ctree_control(stump = TRUE,
                                                mincriterion = 0,
                                                savesplitstats = FALSE))

## constrained effects base-learner
bmono(...,
      constraint = c("increasing", "decreasing", "convex", "concave",
                     "none", "positive", "negative"),
      type = c("quad.prog", "iterative"),
      by = NULL, index = NULL, knots = 20, boundary.knots = NULL,
      degree = 3, differences = 2, df = 4, lambda = NULL,
      lambda2 = 1e6, niter=10, intercept = TRUE,
      contrasts.arg = "contr.treatment",
      boundary.constraints = FALSE,
      cons.arg = list(lambda = 1e+06, n = NULL, diff_order = NULL))

## Markov random field base-learner
bmrf(..., by = NULL, index = NULL, bnd = NULL, df = 4, lambda = NULL,
    center = FALSE)
```

```
## user-specified base-learner
buser(X, K = NULL, by = NULL, index = NULL, df = 4, lambda = NULL)

## combining single base-learners to form new,
## more complex base-learners
bl1 %+% bl2
bl1 %X% bl2
bl1 %O% bl2
```

Arguments

`...`	one or more predictor variables or one matrix or data frame of predictor variables. For smooth base-learners, the number of predictor variables and the number of columns in the data frame / matrix must be less than or equal to 2. If a matrix (with at least 2 columns) is given to `bols` or `brandom`, it is directly used as the design matrix. Especially, no intercept term is added regardless of argument `intercept`. If the argument has only one column, it is simplified to a vector and an intercept is added or not according to the argmuent `intercept`.
`by`	an optional variable defining varying coefficients, either a factor or numeric variable. If `by` is a factor, the coding is determined by the global `options("contrasts")` or as specified "locally" for the factor (see `contrasts`). Per default treatment coding is used. Note that the main effect needs to be specified in a separate base-learner.
`index`	a vector of integers for expanding the variables in `...`. For example, `bols(x, index = index)` is equal to `bols(x[index])`, where `index` is an integer of length greater or equal to `length(x)`.
`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. Certain restrictions have to be kept for the specification of `df` since most of the base-learners rely on penalization approaches with a non-trivial null space. For example, for P-splines fitted with `bbs`, `df` has to be larger than the order of differences employed in the construction of the penalty term. However, when option `center != FALSE`, the effect is centered around its unpenalized part and therefore any positive number is admissible for `df`. For details on the computation of degrees of freedom see section 'Global Options'.
`lambda`	smoothing penalty, computed from `df` when `df` is specified. For details on the computation of degrees of freedom see section 'Global Options'.
`knots`	either the number of knots or a vector of the positions of the interior knots (for more details see below). For multiple predictor variables, `knots` may be a named list where the names in the list are the variable names.
`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.This is only advised for `bbs(..., cyclic = TRUE)`, where the boundary knots specify the points at which the cyclic function should be joined. In analogy to `knots` a names list 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).

intercept — if intercept = TRUE an intercept is added to the design matrix of a linear base-learner. If intercept = FALSE, continuous covariates should be (mean-) centered.

center — if center != FALSE the corresponding effect is re-parameterized such that the unpenalized part of the fit is subtracted and only the deviation effect is fitted. The unpenalized, parametric part has then to be included in separate base-learners using bols (see the examples below). There are two possible ways to re-parameterization; center = "differenceMatrix" is based on the difference matrix (the default for bbs with one covariate only) and center = "spectralDecomp" uses a spectral decomposition of the penalty matrix (see Fahrmeir et al., 2004, Section 2.3 for details). The latter option is the default (and currently only option) for bbs with multiple covariates or bmrf.

cyclic — if cyclic = TRUE the fitted values coincide at the boundaries (useful for cyclic covariates such as day time etc.). For details see Hofner et al. (2014).

covFun — the covariance function (i.e. radial basis) needed to compute the basis functions. Per default stationary.cov function (from package fields) is used.

args — a named list of arguments to be passed to cov.function. Thus strongly dependent on the specified cov.function.

contrasts.arg — a named list of characters suitable for input to the contrasts replacement function, or the contrast matrix itself, see model.matrix, or a single character string (or contrast matrix) which is then used as contrasts for all factors in this base-learner (with the exception of factors in by). See also example below for setting contrasts. Note that a special contrasts.arg exists in package mboost, namely "contr.dummy". This contrast is used per default in brandom and can also be used in bols. 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 seperate effect estimate (see example below).

tree_controls — an object of class "TreeControl", which can be obtained using ctree_control. Defines hyper-parameters for the trees which are used as base-learners, stumps are fitted by default.

constraint — type of constraint to be used. The constraint can be either monotonic "increasing" (default), "decreasing" or "convex" or "concave". Additionally, "none" can be used to specify unconstrained P-splines. This is especially of interest in conjunction with boundary.constraints = TRUE.

type — determines how the constrained least squares problem should be solved. If type = "quad.prog", a numeric quadratic programming method (Goldfarb and Idnani, 1982, 1983) is used (see solve.QP in package **quadprog**). If type = "iterative", the iterative procedure described in Hofner et al. (2011b) is used. The quadratic programming approach is usually much faster than the iterative approach. For details see Hofner et al. (2014).

lambda2 — penalty parameter for the (monotonicity) constraint.

niter — maximum number of iterations used to compute constraint estimates. Increase this number if a warning is displayed.

`boundary.constraints`
: a logical indicating whether additional constraints on the boundaries of the spline should be applied (default: FALSE). This is still experimental.

`cons.arg`
: a named list with additional arguments for boundary constraints. The element `lambda` specifies the penalty parameter that is used for the additional boundary constraint. The element `n` specifies the number of knots to be subject to the constraint and can be either a scalar (use same number of constrained knots on each side) or a vector. Per default 10% of the knots on each side are used. The element `diff_order` can be used to specify the order of the boundary penalty: 1 (constant; default for monotonically constrained effects) or 2 (linear; default for all other effects).

`bnd`
: Object of class `bnd`, in which the boundaries of a map are defined and from which neighborhood relations can be constructed. See `read.bnd`. If a boundary object is not available, the neighborhood matrix can also be given directly.

`X`
: design matrix as it should be used in the penalized least squares estimation. Effect modifiers do not need to be included here (`by` can be used for convenience).

`K`
: penalty matrix as it should be used in the penalized least squares estimation. If `NULL` (default), unpenalized estimation is used.

`deriv`
: an integer; the derivative of the spline of the given order at the data is computed, defaults to zero. Note that this argument is only used to set up the design matrix and cannot be used in the fitting process.

`bl1`
: a linear base-learner or a list of linear base-learners.

`bl2`
: a linear base-learner or a list of linear base-learners.

Details

`bols` refers to linear base-learners (potentially estimated with a ridge penalty), while `bbs` provide penalized regression splines. `bspatial` fits bivariate surfaces and `brandom` defines random effects base-learners. In combination with option `by`, these base-learners can be turned into varying coefficient terms. The linear base-learners are fitted using Ridge Regression where the penalty parameter `lambda` is either computed from `df` (default for `bbs`, `bspatial`, and `brandom`) or specified directly (`lambda = 0` means no penalization as default for `bols`).

In `bols(x)`, `x` may be a numeric vector or factor. Alternatively, `x` can be a data frame containing numeric or factor variables. In this case, or when multiple predictor variables are specified, e.g., using `bols(x1, x2)`, the model is equivalent to `lm(y ~ ., data = x)` or `lm(y ~ x1 + x2)`, respectively. By default, an intercept term is added to the corresponding design matrix (which can be omitted using `intercept = FALSE`). It is *strongly* advised to (mean-) center continuous covariates, if no intercept is used in `bols` (see Hofner et al., 2011a). If `x` is a matrix, it is directly used as the design matrix and no further preprocessing (such as addition of an intercept) is conducted. When `df` (or `lambda`) is given, a ridge estimator with `df` degrees of freedom (see section 'Global Options') is used as base-learner. Note that all variables are treated as a group, i.e., they enter the model together if the corresponding base-learner is selected. For ordinal variables, a ridge penalty for the differences of the adjacent categories (Gertheiss and Tutz 2009, Hofner et al. 2011a) is applied.

With `bbs`, the P-spline approach of Eilers and Marx (1996) is used. P-splines use a squared k-th-order difference penalty which can be interpreted as an approximation of the integrated squared k-th derivative of the spline. In `bbs` the argument `knots` specifies either the number of (equidistant)

interior knots to be used for the regression spline fit or a vector including the positions of the *interior* knots. Additionally, `boundary.knots` can be specified. However, this is only advised if one uses cyclic constraints, where the `boundary.knots` specify the points where the function is joined (e.g., `boundary.knots = c(0, 2 * pi)` for angles as in a sine function or `boundary.knots = c(0, 24)` for hours during the day). For details on cylcic splines in the context of boosting see Hofner et al. (2014).

`bspatial` implements bivariate tensor product P-splines for the estimation of either spatial effects or interaction surfaces. Note that `bspatial(x, y)` is equivalent to `bbs(x, y, df = 6)`. For possible arguments and defaults see there. The penalty term is constructed based on bivariate extensions of the univariate penalties in `x` and `y` directions, see Kneib, Hothorn and Tutz (2009) for details. Note that the dimensions of the penalty matrix increase (quickly) with the number of knots with strong impact on computational time. Thus, both should not be chosen to large. Different knots for `x` and `y` can be specified by a named list.

`brandom(x)` specifies a random effects base-learner based on a factor variable `x` that defines the grouping structure of the data set. For each level of `x`, a separate random intercept is fitted, where the random effects variance is governed by the specification of the degrees of freedom `df` or `lambda` (see section 'Global Options'). Note that `brandom(...)` is essentially a wrapper to `bols(..., df = 4, contrasts.arg = "contr.dummy")`, i.e., a wrapper that utilizes ridge-penalized categorical effects. For possible arguments and defaults see `bols`.

For all linear base-learners the amount of smoothing is determined by the trace of the hat matrix, as indicated by `df`.

If `by` is specified as an additional argument, a varying coefficients term is estimated, where `by` is the interaction variable and the effect modifier is given by either `x` or `x` and `y` (specified via `...`). If `bbs` is used, this corresponds to the classical situation of varying coefficients, where the effect of `by` varies over the co-domain of `x`. In case of `bspatial` as base-learner, the effect of `by` varies with respect to both `x` and `y`, i.e. an interaction surface between `x` and `y` is specified as effect modifier. For `brandom` specification of `by` leads to the estimation of random slopes for covariate `by` with grouping structure defined by factor `x` instead of a simple random intercept. In `bbs`, `bspatial` and `brandom` the computation of the smoothing parameter `lambda` for given `df`, or vice versa, might become (numerically) instable if the values of the interaction variable `by` become too large. In this case, we recommend to rescale the interaction covariate e.g. by dividing by `max(abs(by))`. If `bbs` or `bspatial` is specified with an factor variable `by` with more than two factors, the degrees of freedom are shared for the complete base-learner (i.e., spread over all factor levels). Note that the null space (see next paragraph) increases, as a separate null space for each factor level is present. Thus, the minimum degrees of freedom increase with increasing number of levels of `by` (if `center = FALSE`).

For `bbs` and `bspatial`, option `center != FALSE` requests that the fitted effect is centered around its parametric, unpenalized part (the so called null space). For example, with second order difference penalty, a linear effect of `x` remains unpenalized by `bbs` and therefore the degrees of freedom for the base-learner have to be larger than two. To avoid this restriction, option `center = TRUE` subtracts the unpenalized linear effect from the fit, allowing to specify any positive number as `df`. Note that in this case the linear effect `x` should generally be specified as an additional base-learner `bols(x)`. For `bspatial` and, for example, second order differences, a linear effect of `x` (`bols(x)`), a linear effect of `y` (`bols(y)`), and their interaction (`bols(x*y)`) are subtracted from the effect and have to be added separately to the model equation. More details on centering can be found in Kneib, Hothorn and Tutz (2009) and Fahrmeir, Kneib and Lang (2004). We strongly recommend to consult the latter reference before using this option.

`brad(x)` specifies penalized radial basis functions as used in Kriging. If `knots` is used to specify the

number of knots, the function cover.design is used to specify the location of the knots such that they minimize a geometric space-filling criterion. Furthermore, knots can be specified directly via a matrix. The cov.function allows to specify the radial basis functions. Per default, the flexible Matern correlation function is used. This is specified using cov.function = stationary.cov with Covariance = "Matern" specified via args. If an effective range theta is applicable for the correlation function (e.g., the Matern family) the user can specify this value. Per default (if theta = NULL) the effective range is chosen as $\theta = max(||x_i - x_j||)/c$ such that the correlation function

$$\rho(c; \theta = 1) = \varepsilon,$$

where $\varepsilon = 0.001$.

bmrf builds a base of a Markov random field consisting of several regions with a neighborhood structure. The input variable is the observed region. The penalty matrix is either construed from a boundary object or must be given directly via the option bnd. In that case the dimnames of the matrix have to be the region names, on the diagonal the number of neighbors have to be given for each region, and for each neighborhood relation the value in the matrix has to be -1, else 0. With a boundary object at hand, the fitted or predicted values can be directly plotted into the map using drawmap.

buser(X, K) specifies a base-learner with user-specified design matrix X and penalty matrix K, where X and K are used to minimize a (penalized) least squares criterion with quadratic penalty. This can be used to easily specify base-learners that are not implemented (yet). See examples below for details how buser can be used to mimic existing base-learners. Note that for predictions you need to set up the design matrix for the new data manually.

For a categorical covariate with non-observed categories bols(x) and brandom(x) both assign a zero effect to these categories. However, the non-observed categories must be listed in levels(x). Thus, predictions are possible for new observations if they correspond to this category.

By default, all linear base-learners include an intercept term (which can be removed using intercept = FALSE for bols or center = TRUE for bbs). In this case, an explicit global intercept term should be added to gamboost via bols (see example below). With bols(x, intercept = FALSE) with categorical covariate x a separate effect for each group (mean effect) is estimated (see examples for resulting design matrices).

Smooth estimates with constraints can be computed using the base-learner bmono() which specifies P-spline base-learners with an additional asymmetric penalty enforcing monotonicity or convexity/concavity (see and Eilers, 2005). For more details in the boosting context and monotonic effects of ordinal factors see Hofner, Mueller and Hothorn (2011b). The quadratic-programming based algorithm is described in Hofner et al. (2014). Alternative monotonicity constraints are implemented via T-splines in bbs() (Beliakov, 2000).

Two or more linear base-learners can be joined using %+%. A tensor product of two or more linear base-learners is returned by %X%. When the design matrix can be written as the Kronecker product of two matrices X = kronecker(X2, X1), then bl1 %O% bl2 with design matrices X1 and X2, respectively, can be used to efficiently compute Ridge-estimates following Currie, Durban, Eilers (2006). In all cases the overall degrees of freedom of the combined base-learner increase (additive or multiplicative, respectively). These three features are experimental and for expert use only.

btree fits a stump to one or more variables. Note that blackboost is more efficient for boosting stumps. For further references see Hothorn, Hornik, Zeileis (2006) and Hothorn et al. (2010).

Note that the base-learners bns and bss are deprecated (and no longer available). Please use bbs instead, which results in qualitatively the same models but is computationally much more attractive.

Value

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

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).

Global Options

Three global options affect the base-learners:

`options("mboost_useMatrix")` defaulting to `TRUE` indicates that the base-learner may use sparse matrix techniques for its computations. This reduces the memory consumption but might (for smaller sample sizes) require more computing time.

`options("mboost_indexmin")` is an integer that specifies the minimum sample size needed to optimize model fitting by automatically taking ties into account (default = 10000).

`options("mboost_dftraceS")` `FALSE` by default, indicating how the degrees of freedom should be computed. Per default

$$\mathrm{df}(\lambda) = \mathrm{trace}(2S - S^\top S),$$

with smoother matrix $S = X(X^\top X + \lambda K)^{-1} X$ is used (see Hofner et al., 2011a). If `TRUE`, the trace of the smoother matrix $\mathrm{df}(\lambda) = \mathrm{trace}(S)$ is used as degrees of freedom.

Note that these formulae specify the relation of `df` and `lambda` as the smoother matrix S depends only on λ (and the (fixed) design matrix X, the (fixed) penalty matrix K).

References

Iain D. Currie, Maria Durban, and Paul H. C. Eilers (2006), Generalized linear array models with applications to multidimensional smoothing. *Journal of the Royal Statistical Society, Series B–Statistical Methodology*, **68**(2), 259–280.

Paul H. C. Eilers (2005), Unimodal smoothing. *Journal of Chemometrics*, **19**, 317–328.

Paul H. C. Eilers and Brian D. Marx (1996), Flexible smoothing with B-splines and penalties. *Statistical Science*, **11**(2), 89-121.

Ludwig Fahrmeir, Thomas Kneib and Stefan Lang (2004), Penalized structured additive regression for space-time data: a Bayesian perspective. *Statistica Sinica*, **14**, 731-761.

Jan Gertheiss and Gerhard Tutz (2009), Penalized regression with ordinal predictors, *International Statistical Review*, **77**(3), 345–365.

D. Goldfarb and A. Idnani (1982), Dual and Primal-Dual Methods for Solving Strictly Convex Quadratic Programs. In J. P. Hennart (ed.), Numerical Analysis, Springer-Verlag, Berlin, pp. 226-239.

D. Goldfarb and A. Idnani (1983), A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical Programming*, **27**, 1–33.

Benjamin Hofner, Torsten Hothorn, Thomas Kneib, and Matthias Schmid (2011a), A framework for unbiased model selection based on boosting. *Journal of Computational and Graphical Statistics*, **20**, 956–971.

Benjamin Hofner, Joerg Mueller, and Torsten Hothorn (2011b), Monotonicity-Constrained Species Distribution Models, *Ecology*, **92**, 1895–1901.

Benjamin Hofner, Thomas Kneib and Torsten Hothorn (2014), A Unified Framework of Constrained Regression. *Statistics & Computing*. Online first. DOI:10.1007/s11222-014-9520-y. Preliminary version: http://arxiv.org/abs/1403.7118

Thomas Kneib, Torsten Hothorn and Gerhard Tutz (2009), Variable selection and model choice in geoadditive regression models, *Biometrics*, **65**(2), 626–634.

Torsten Hothorn, Kurt Hornik, Achim Zeileis (2006), Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, **15**, 651–674.

Torsten Hothorn, Peter Buehlmann, Thomas Kneib, Matthias Schmid and Benjamin Hofner (2010), Model-based Boosting 2.0, *Journal of Machine Learning Research*, **11**, 2109–2113.

G. M. Beliakov (2000), Shape Preserving Approximation using Least Squares Splines, *Approximation Theory and its Applications*, **16**(4), 80–98.

See Also

mboost

Examples

```
set.seed(290875)

n <- 100
x1 <- rnorm(n)
x2 <- rnorm(n) + 0.25 * x1
x3 <- as.factor(sample(0:1, 100, replace = TRUE))
x4 <- gl(4, 25)
y <- 3 * sin(x1) + x2^2 + rnorm(n)
weights <- drop(rmultinom(1, n, rep.int(1, n) / n))

### set up base-learners
spline1 <- bbs(x1, knots = 20, df = 4)
extract(spline1, "design")[1:10, 1:10]
extract(spline1, "penalty")
knots.x2 <- quantile(x2, c(0.25, 0.5, 0.75))
spline2 <- bbs(x2, knots = knots.x2, df = 5)
ols3 <- bols(x3)
extract(ols3)
ols4 <- bols(x4)

### compute base-models
drop(ols3$dpp(weights)$fit(y)$model) ## same as:
coef(lm(y ~ x3, weights = weights))

drop(ols4$dpp(weights)$fit(y)$model) ## same as:
coef(lm(y ~ x4, weights = weights))

### fit model, component-wise
mod1 <- mboost_fit(list(spline1, spline2, ols3, ols4), y, weights)

### more convenient formula interface
```

```
  mod2 <- mboost(y ~ bbs(x1, knots = 20, df = 4) +
                     bbs(x2, knots = knots.x2, df = 5) +
                     bols(x3) + bols(x4), weights = weights)
  all.equal(coef(mod1), coef(mod2))


  ### grouped linear effects
  # center x1 and x2 first
  x1 <- scale(x1, center = TRUE, scale = FALSE)
  x2 <- scale(x2, center = TRUE, scale = FALSE)
  model <- gamboost(y ~ bols(x1, x2, intercept = FALSE) +
                        bols(x1, intercept = FALSE) +
                        bols(x2, intercept = FALSE),
                        control = boost_control(mstop = 50))
  coef(model, which = 1)   # one base-learner for x1 and x2
  coef(model, which = 2:3) # two separate base-learners for x1 and x2
                           # zero because they were (not yet) selected.

  ### example for bspatial
  x1 <- runif(250,-pi,pi)
  x2 <- runif(250,-pi,pi)

  y <- sin(x1) * sin(x2) + rnorm(250, sd = 0.4)

  spline3 <- bspatial(x1, x2, knots = 12)
  Xmat <- extract(spline3, "design")
  ## 12 inner knots + 4 boundary knots = 16 knots per direction
  ## THUS: 16 * 16 = 256 columns
  dim(Xmat)
  extract(spline3, "penalty")[1:10, 1:10]

  ## specify number of knots separately
  form1 <- y ~ bspatial(x1, x2, knots = list(x1 = 12, x2 = 14))

  ## decompose spatial effect into parametric part and
  ## deviation with one df
  form2 <- y ~ bols(x1) + bols(x2) + bols(x1, by = x2, intercept = FALSE) +
               bspatial(x1, x2, knots = 12, center = TRUE, df = 1)

############################################################
## Do not run and check these examples automatically as
## they take some time

  mod1 <- gamboost(form1)
  plot(mod1)

  mod2 <- gamboost(form2)
  ## automated plot function:
  plot(mod2)
  ## plot sum of linear and smooth effects:
  library(lattice)
  df <- expand.grid(x1 = unique(x1), x2 = unique(x2))
  df$pred <- predict(mod2, newdata = df)
```

```
  levelplot(pred ~ x1 * x2, data = df)

## End(Not run and test)


  ## specify radial basis function base-learner for spatial effect
  ## and use data-adaptive effective range (theta = NULL, see 'args')
  form3 <- y ~ brad(x1, x2)
  ## Now use different settings, e.g. 50 knots and theta fixed to 0.4
  ## (not really a good setting)
  form4 <- y ~ brad(x1, x2, knots = 50, args = list(theta = 0.4))

############################################################
## Do not run and check these examples automatically as
## they take some time
  mod3 <- gamboost(form3)
  plot(mod3)
  dim(extract(mod3, what = "design", which = "brad")[[1]])
  knots <- attr(extract(mod3, what = "design", which = "brad")[[1]], "knots")

  mod4 <- gamboost(form4)
  dim(extract(mod4, what = "design", which = "brad")[[1]])
  plot(mod4)

## End(Not run and test)


  ### random intercept
  id <- factor(rep(1:10, each = 5))
  raneff <- brandom(id)
  extract(raneff, "design")
  extract(raneff, "penalty")

  ## random intercept with non-observed category
  set.seed(1907)
  y <- rnorm(50, mean = rep(rnorm(10), each = 5), sd = 0.1)
  plot(y ~ id)
  # category 10 not observed
  obs <- c(rep(1, 45), rep(0, 5))
  model <- gamboost(y ~ brandom(id), weights = obs)
  coef(model)
  fitted(model)[46:50] # just the grand mean as usual for
                       # random effects models


  ### random slope
  z <- runif(50)
  raneff <- brandom(id, by = z)
  extract(raneff, "design")
  extract(raneff, "penalty")

  ### specify simple interaction model (with main effect)
  n <- 210
```

```
x <- rnorm(n)
X <- model.matrix(~ x)
z <- gl(3, n/3)
Z <- model.matrix(~z)
beta <- list(c(0,1), c(-3,4), c(2, -4))
y <- rnorm(length(x), mean = (X * Z[,1]) %*% beta[[1]] +
                             (X * Z[,2]) %*% beta[[2]] +
                             (X * Z[,3]) %*% beta[[3]])
plot(y ~ x, col = z)
## specify main effect and interaction
mod_glm <- gamboost(y ~ bols(x) + bols(x, by = z),
                control = boost_control(mstop = 100))
nd <- data.frame(x, z)
nd <- nd[order(x),]
nd$pred_glm <- predict(mod_glm, newdata = nd)
for (i in seq(along = levels(z)))
    with(nd[nd$z == i,], lines(x, pred_glm, col = z))
mod_gam <- gamboost(y ~ bbs(x) + bbs(x, by = z, df = 8),
                    control = boost_control(mstop = 100))
nd$pred_gam <- predict(mod_gam, newdata = nd)
for (i in seq(along = levels(z)))
    with(nd[nd$z == i,], lines(x, pred_gam, col = z, lty = "dashed"))
### convenience function for plotting
par(mfrow = c(1,3))
plot(mod_gam)


### remove intercept from base-learner
### and add explicit intercept to the model
tmpdata <- data.frame(x = 1:100, y = rnorm(1:100), int = rep(1, 100))
mod <- gamboost(y ~ bols(int, intercept = FALSE) +
                    bols(x, intercept = FALSE),
                data = tmpdata,
                control = boost_control(mstop = 1000))
cf <- unlist(coef(mod))
## add offset
cf[1] <- cf[1] + mod$offset
signif(cf, 3)
signif(coef(lm(y ~ x, data = tmpdata)), 3)

### much quicker and better with (mean-) centering
tmpdata$x_center <- tmpdata$x - mean(tmpdata$x)
mod_center <- gamboost(y ~ bols(int, intercept = FALSE) +
                           bols(x_center, intercept = FALSE),
                       data = tmpdata,
                       control = boost_control(mstop = 100))
cf_center <- unlist(coef(mod_center, which=1:2))
## due to the shift in x direction we need to subtract
## beta_1 * mean(x) to get the correct intercept
cf_center[1] <- cf_center[1] + mod_center$offset -
                cf_center[2] * mean(tmpdata$x)
signif(cf_center, 3)
signif(coef(lm(y ~ x, data = tmpdata)), 3)
```

```
############################################################
## Do not run and check these examples automatically as
## they take some time

  ### large data set with ties
  nunique <- 100
  xindex <- sample(1:nunique, 1000000, replace = TRUE)
  x <- runif(nunique)
  y <- rnorm(length(xindex))
  w <- rep.int(1, length(xindex))

  ### brute force computations
  op <- options()
  options(mboost_indexmin = Inf, mboost_useMatrix = FALSE)
  ## data pre-processing
  b1 <- bbs(x[xindex])$dpp(w)
  ## model fitting
  c1 <- b1$fit(y)$model
  options(op)

  ### automatic search for ties, faster
  b2 <- bbs(x[xindex])$dpp(w)
  c2 <- b2$fit(y)$model

  ### manual specification of ties, even faster
  b3 <- bbs(x, index = xindex)$dpp(w)
  c3 <- b3$fit(y)$model

  all.equal(c1, c2)
  all.equal(c1, c3)

## End(Not run and test)


  ### cyclic P-splines
  set.seed(781)
  x <- runif(200, 0,(2*pi))
  y <- rnorm(200, mean=sin(x), sd=0.2)
  newX <- seq(0,2*pi, length=100)
  ### model without cyclic constraints
  mod <- gamboost(y ~ bbs(x, knots = 20))
  ### model with cyclic constraints
  mod_cyclic <- gamboost(y ~ bbs(x, cyclic=TRUE, knots = 20,
                                 boundary.knots=c(0, 2*pi)))
  par(mfrow = c(1,2))
  plot(x,y, main="bbs (non-cyclic)", cex=0.5)
  lines(newX, sin(newX), lty="dotted")
  lines(newX + 2 * pi, sin(newX), lty="dashed")
  lines(newX, predict(mod, data.frame(x = newX)),
        col="red", lwd = 1.5)
  lines(newX + 2 * pi, predict(mod, data.frame(x = newX)),
        col="blue", lwd=1.5)
```

```
  plot(x,y, main="bbs (cyclic)", cex=0.5)
  lines(newX, sin(newX), lty="dotted")
  lines(newX + 2 * pi, sin(newX), lty="dashed")
  lines(newX, predict(mod_cyclic, data.frame(x = newX)),
        col="red", lwd = 1.5)
  lines(newX + 2 * pi, predict(mod_cyclic, data.frame(x = newX)),
        col="blue", lwd = 1.5)

  ### use buser() to mimic p-spline base-learner:
  set.seed(1907)
  x <- rnorm(100)
  y <- rnorm(100, mean = x^2, sd = 0.1)
  mod1 <- gamboost(y ~ bbs(x))
  ## now extract design and penalty matrix
  X <- extract(bbs(x), "design")
  K <- extract(bbs(x), "penalty")
  ## use X and K in buser()
  mod2 <- gamboost(y ~ buser(X, K))
  max(abs(predict(mod1) - predict(mod2)))  # same results

  ### use buser() to mimic penalized ordinal base-learner:
  z <- as.ordered(sample(1:3, 100, replace=TRUE))
  y <- rnorm(100, mean = as.numeric(z), sd = 0.1)
  X <- extract(bols(z))
  K <- extract(bols(z), "penalty")
  index <- extract(bols(z), "index")
  mod1 <- gamboost(y ~  buser(X, K, df = 1, index = index))
  mod2 <- gamboost(y ~  bols(z, df = 1))
  max(abs(predict(mod1) - predict(mod2)))  # same results

  ### kronecker product for matrix-valued responses
  data("volcano", package = "datasets")
  layout(matrix(1:2, ncol = 2))

  ## estimate mean of image treating image as matrix
  image(volcano, main = "data")
  x1 <- 1:nrow(volcano)
  x2 <- 1:ncol(volcano)

  vol <- as.vector(volcano)
  mod <- mboost(vol ~ bbs(x1, df = 3, knots = 10)%O%
                      bbs(x2, df = 3, knots = 10),
                      control = boost_control(nu = 0.25))
  mod[250]

  volf <- matrix(fitted(mod), nrow = nrow(volcano))
  image(volf, main = "fitted")

############################################################
## Do not run and check these examples automatically as
## they take some time

  ## the old-fashioned way, a waste of space and time
```

```
  x <- expand.grid(x1, x2)
  modx <- mboost(vol ~ bbs(Var2, df = 3, knots = 10) %X%
                       bbs(Var1, df = 3, knots = 10), data = x,
                       control = boost_control(nu = 0.25))
  modx[250]

  max(abs(fitted(mod) - fitted(modx)))

## End(Not run and test)


  ### setting contrasts via contrasts.arg
  x <- as.factor(sample(1:4, 100, replace = TRUE))

  ## compute base-learners with different reference categories
  BL1 <- bols(x, contrasts.arg = contr.treatment(4, base = 1)) # default
  BL2 <- bols(x, contrasts.arg = contr.treatment(4, base = 2))
  ## compute 'sum to zero contrasts' using character string
  BL3 <- bols(x, contrasts.arg = "contr.sum")

  ## extract model matrices to check if it works
  extract(BL1)
  extract(BL2)
  extract(BL3)

  ### setting contrasts using named lists in contrasts.arg
  x2 <- as.factor(sample(1:4, 100, replace = TRUE))

  BL4 <- bols(x, x2,
              contrasts.arg = list(x = contr.treatment(4, base = 2),
                                   x2 = "contr.helmert"))
  extract(BL4)

  ### using special contrast: "contr.dummy":
  BL5 <- bols(x, contrasts.arg = "contr.dummy")
  extract(BL5)
```

`blackboost` *Gradient Boosting with Regression Trees*

Description

Gradient boosting for optimizing arbitrary loss functions where regression trees are utilized as base-learners.

Usage

```
blackboost(formula, data = list(),
           weights = NULL, na.action = na.pass,
           tree_controls = party::ctree_control(
```

```
                teststat = "max",
                testtype = "Teststatistic",
                mincriterion = 0,
                maxdepth = 2, savesplitstats = FALSE),
            ...)
```

Arguments

formula	a symbolic description of the model to be fit.
data	a data frame containing the variables in the model.
weights	an optional vector of weights to be used in the fitting process.
na.action	a function which indicates what should happen when the data contain NAs.
tree_controls	an object of class "TreeControl", which can be obtained using ctree_control. Defines hyper-parameters for the trees which are used as base-learners. It is wise to make sure to understand the consequences of altering any of its arguments.
...	additional arguments passed to mboost_fit, including weights, offset, family and control. For default values see mboost_fit.

Details

This function implements the 'classical' gradient boosting utilizing regression trees as base-learners. Essentially, the same algorithm is implemented in package gbm. The main difference is that arbitrary loss functions to be optimized can be specified via the family argument to blackboost whereas gbm uses hard-coded loss functions. Moreover, the base-learners (conditional inference trees, see ctree) are a little bit more flexible.

The regression fit is a black box prediction machine and thus hardly interpretable.

Partial dependency plots are not yet available; see example section for plotting of additive tree models.

Value

An object of class mboost with print and predict methods being available.

References

Peter Buehlmann and Torsten Hothorn (2007), Boosting algorithms: regularization, prediction and model fitting. *Statistical Science*, **22**(4), 477–505.

Torsten Hothorn, Kurt Hornik and Achim Zeileis (2006). Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, **15**(3), 651–674.

Yoav Freund and Robert E. Schapire (1996), Experiments with a new boosting algorithm. In *Machine Learning: Proc. Thirteenth International Conference*, 148–156.

Jerome H. Friedman (2001), Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, **29**, 1189–1232.

Greg Ridgeway (1999), The state of boosting. *Computing Science and Statistics*, **31**, 172–181.

See Also

mboost for the generic boosting function and glmboost for boosted linear models and gamboost for boosted additive models. See cvrisk for cross-validated stopping iteration. Furthermore see boost_control, Family and methods

Examples

```
### a simple two-dimensional example: cars data
cars.gb <- blackboost(dist ~ speed, data = cars,
                      control = boost_control(mstop = 50))
cars.gb

### plot fit
plot(dist ~ speed, data = cars)
lines(cars$speed, predict(cars.gb), col = "red")

### set up and plot additive tree model
if (require("party")) {
    ctrl <- ctree_control(maxdepth = 3)
    viris <- subset(iris, Species != "setosa")
    viris$Species <- viris$Species[, drop = TRUE]
    imod <- mboost(Species ~ btree(Sepal.Length, tree_controls = ctrl) +
                             btree(Sepal.Width, tree_controls = ctrl) +
                             btree(Petal.Length, tree_controls = ctrl) +
                             btree(Petal.Width, tree_controls = ctrl),
                   data = viris, family = Binomial())[500]
    layout(matrix(1:4, ncol = 2))
    plot(imod)
}
```

---

boost_control *Control Hyper-parameters for Boosting Algorithms*

Description

Definition of the initial number of boosting iterations, step size and other hyper-parameters for boosting algorithms.

Usage

```
boost_control(mstop = 100, nu = 0.1,
              risk = c("inbag", "oobag", "none"), stopintern = FALSE,
              center = TRUE, trace = FALSE)
```

Arguments

mstop	an integer giving the number of initial boosting iterations.
nu	a double (between 0 and 1) defining the step size or shrinkage parameter. The default is probably too large for many applications with `family = Poisson()` and a smaller value is better.
risk	a character indicating how the empirical risk should be computed for each boosting iteration. `inbag` leads to risks computed for the learning sample (i.e., all non-zero weights), `oobag` to risks based on the out-of-bag (all observations with zero weights) and `none` to no risk computations at all.
stopintern	a logical that defines if the boosting algorithm stops internally when the out-of-bag risk in one iteration is larger than the out-of-bag risk in the iteration before. Can also be a positive number giving the risk difference that needs to be exceeded.
center	deprecated. A logical indicating if the numerical covariates should be mean centered before fitting. Only implemented for `glmboost`. In `blackboost` centering is not needed. In `gamboost` centering is only needed if `bols` base-learners are specified without intercept. In this case centering of the covariates is essential and should be done manually (at the moment). Will be removed in favour of a corresponding argument in `glmboost` in the future (and gives a warning).
trace	a logical triggering printout of status information during the fitting process.

Details

Objects returned by this function specify hyper-parameters of the boosting algorithms implemented in `glmboost`, `gamboost` and `blackboost` (via the `control` argument).

Value

An object of class `boost_control`, a list.

See Also

`mboost`

---

boost_family-class *Class "boost_family": Gradient Boosting Family*

Description

Objects of class `boost_family` define negative gradients of loss functions to be optimized.

Objects from the Class

Objects can be created by calls of the form `Family(...)`

Slots

`ngradient`: a function with arguments `y` and `f` implementing the *negative* gradient of the `loss` function.

`risk`: a risk function with arguments `y`, `f` and `w`, the weighted mean of the loss function by default.

`offset`: a function with argument `y` and `w` (weights) for computing a *scalar* offset.

`weights`: a logical indicating if weights are allowed.

`check_y`: a function for checking the class / mode of a response variable.

`nuisance`: a function for extracting nuisance parameters.

`response`: inverse link function of a GLM or any other transformation on the scale of the response.

`rclass`: function to derive class predictions from conditional class probabilities (for models with factor response variable).

`name`: a character giving the name of the loss function for pretty printing.

`charloss`: a character, the deparsed loss function.

See Also

`Family`

Examples

```
Laplace()
```

`confint.mboost` *Pointwise Bootstrap Confidence Intervals*

Description

Compute and display pointwise confidence intervals

Usage

```
## S3 method for class 'mboost'
confint(object, parm = NULL, level = 0.95, B = 1000,
        B.mstop = 25, newdata = NULL, which = parm,
        papply = ifelse(B.mstop == 0, mclapply, lapply),
        cvrisk_options = list(), ...)
## S3 method for class 'mboost.ci'
plot(x, which, level = x$level, ylim = NULL, type = "l", col = "black",
     ci.col = rgb(170, 170, 170, alpha = 85, maxColorValue = 255),
     raw = FALSE, print_levelplot = TRUE,...)
## S3 method for class 'mboost.ci'
lines(x, which, level = x$level,
```

```
     col = rgb(170, 170, 170, alpha = 85, maxColorValue = 255),
     raw = FALSE, ...)


## S3 method for class 'glmboost'
confint(object, parm = NULL, level = 0.95,
        B = 1000, B.mstop = 25, which = parm, ...)
## S3 method for class 'glmboost.ci'
print(x, which = NULL, level = x$level, pe = FALSE, ...)
```

Arguments

object	a fitted model object of class glmboost, gamboost or mboost for which the confidence intervals should be computed.
parm, which	a subset of base-learners to take into account for computing confidence intervals. See mboost_methods for details. parm is just a synonyme for which to be in line with the generic confint function. Preferably use which.
level	the confidence level required.
B	number of outer bootstrap replicates used to compute the empirical bootstrap confidence intervals.
B.mstop	number of inner bootstrap replicates used to determine the optimal mstop on each of the B bootstrap samples.
newdata	optionally, a data frame on which to compute the predictions for the confidence intervals.
papply	(parallel) apply function for the outer bootstrap, defaults to mclapply if no inner bootstrap is used to determine the optimal stopping iteration. For details see argument papply in cvrisk. Be careful with your computing resources if you use parallel computing for both, the inner and the outer bootstrap.
cvrisk_options	(optionally) specify a named list with arguments to the inner bootstrap. For example use cvrisk_options = list(mc.cores = 2) to specify that the mclapply function within cvrisk uses 2 cores to compute the optimal mstop.
x	a confidence interval object.
ylim	limits of the y scale. Per default computed from the data to plot.
type	type of graphic for the point estimate, i.e., for the predicted function. Per default a line is plotted.
col	color of the point estimate, i.e., for the predicted function.
ci.col	color of the confidence interval.
raw	logical, should the raw function estimates or the derived confidence estimates be plotted?
print_levelplot	logical, should the **lattice** levelplot be printed or simply returned for further modifications. This argument is only considered if bivariate effect estimates are plotted. If print_levelplot is set to FALSE, a list with objects mean, lowerPI and upperPI is returned containing the three levelplot objects.

pe	logical, should the point estimtate (PE) be also returned?
...	additional arguments to the outer bootstrap such as `mc.cores`.

Details

Use a nested boostrap approach to compute pointwise confidence intervals for the predicted partial functions or regression parameters.

Value

An object of class `glmboost.ci` or `mboost.ci` with special `print` and/or `plot` functions.

Author(s)

Benjamin Hofner <benjamin.hofner@fau.de>

References

Benjamin Hofner, Thomas Kneib and Torsten Hothorn (2014), A Unified Framework of Constrained Regression. *Statistics & Computing*. Online first. DOI:10.1007/s11222-014-9520-y.

Preliminary version: http://arxiv.org/abs/1403.7118

See Also

`cvrisk` for crossvalidation approaches and `mboost_methods` for other methods.

Examples

```
############################################################
## Do not run these examples automatically as they take
## some time (~ 30 seconds depending on the system)

### a simple linear example
set.seed(1907)
data <- data.frame(x1 = rnorm(100), x2 = rnorm(100),
                   z = factor(sample(1:3, 100, replace = TRUE)))
data$y <- rnorm(100, mean = data$x1 - data$x2 - 1 * (data$z == 2) +
                            1 * (data$z == 3), sd = 0.1)
linmod <- glmboost(y ~ x1 + x2 + z, data = data,
                   control = boost_control(mstop = 200))

## compute confidence interval from 10 samples. Usually one should use
## at least 1000 samples.
CI <- confint(linmod, B = 10, level = 0.9)
CI

## to compute a confidence interval for another level simply change the
## level in the print function:
print(CI, level = 0.8)
## or print a subset (with point estimates):
```

```
print(CI, level = 0.8, pe = TRUE, which = "z")

### a simple smooth example
set.seed(1907)
data <- data.frame(x1 = rnorm(100), x2 = rnorm(100))
data$y <- rnorm(100, mean = data$x1^2 - sin(data$x2), sd = 0.1)
gam <- gamboost(y ~ x1 + x2, data = data,
                control = boost_control(mstop = 200))

## compute confidence interval from 10 samples. Usually one should use
## at least 1000 samples.
CI_gam <- confint(gam, B = 10, level = 0.9)

par(mfrow = c(1, 2))
plot(CI_gam, which = 1)
plot(CI_gam, which = 2)
## to compute a confidence interval for another level simply change the
## level in the plot or lines function:
lines(CI_gam, which = 2, level = 0.8)
```

`cvrisk` *Cross-Validation*

Description

Cross-validated estimation of the empirical risk for hyper-parameter selection.

Usage

```
## S3 method for class 'mboost'
cvrisk(object, folds = cv(model.weights(object)),
       grid = 1:mstop(object),
       papply = mclapply,
       fun = NULL, corrected = TRUE, mc.preschedule = FALSE, ...)
cv(weights, type = c("bootstrap", "kfold", "subsampling"),
   B = ifelse(type == "kfold", 10, 25), prob = 0.5, strata = NULL)
```

Arguments

`object`	an object of class `mboost`.
`folds`	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. Can be computed using function `cv` and defaults to 25 bootstrap samples.
`grid`	a vector of stopping parameters the empirical risk is to be evaluated for.
`papply`	(parallel) apply function, defaults to `mclapply`. Alternatively, `parLapply` can be used. In the latter case, usually more setup is needed (see example for some details). To run `cvrisk` sequentially (i.e. not in parallel), one can use `lapply`.

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.
corrected	if TRUE, the corrected cross-validation scheme of Verweij and van Houwelingen (1993) is used in case of Cox models. Otherwise, the naive standard cross-validation scheme is used.
mc.preschedule	preschedule tasks if are parallelized using mclapply (default: FALSE)? For details see mclapply.
weights	a numeric vector of weights for the model to be cross-validated.
type	character argument for specifying the cross-validation method. Currently (stratified) bootstrap, k-fold cross-validation and subsampling 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.
...	additional arguments passed to mclapply.

Details

The number of boosting iterations is a hyper-parameter of the boosting algorithms implemented in this package. Honest, i.e., cross-validated, estimates of the empirical risk for different stopping parameters mstop are computed by this function which can be utilized to choose an appropriate number of boosting iterations to be applied.

Different forms of cross-validation can be applied, for example 10-fold cross-validation or bootstrapping. The weights (zero weights correspond to test cases) are defined via the folds matrix.

cvrisk runs in parallel on OSes where forking is possible (i.e., not on Windows) and multiple cores/processors are available. The scheduling can be changed by the corresponding arguments of mclapply (via the dot arguments).

The function cv can be used to build an appropriate weight matrix to be used with cvrisk. If strata is defined sampling is performed in each stratum separately thus preserving the distribution of the strata variable in each fold.

Value

An object of class cvrisk (when fun wasn't 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 a mstop method.

References

Torsten Hothorn, Friedrich Leisch, Achim Zeileis and Kurt Hornik (2006), The design and analysis of benchmark experiments. *Journal of Computational and Graphical Statistics*, **14**(3), 675–699.

Andreas Mayr, Benjamin Hofner, and Matthias Schmid (2012). The importance of knowing when to stop - a sequential stopping rule for component-wise gradient boosting. *Methods of Information*

in Medicine, **51**, 178–186.
DOI: http://dx.doi.org/10.3414/ME11-02-0030

Verweij and van Houwelingen (1993). Cross-validation in survival analysis. *Statistics in Medicine*, **12**:2305–2314.

See Also

`AIC.mboost` for `AIC` based selection of the stopping iteration. Use `mstop` to extract the optimal stopping iteration from `cvrisk` object.

Examples

```
data("bodyfat", package = "TH.data")

### fit linear model to data
model <- glmboost(DEXfat ~ ., data = bodyfat, center = TRUE)

### AIC-based selection of number of boosting iterations
maic <- AIC(model)
maic

### inspect coefficient path and AIC-based stopping criterion
par(mai = par("mai") * c(1, 1, 1, 1.8))
plot(model)
abline(v = mstop(maic), col = "lightgray")

### 10-fold cross-validation
cv10f <- cv(model.weights(model), type = "kfold")
cvm <- cvrisk(model, folds = cv10f, papply = lapply)
print(cvm)
mstop(cvm)
plot(cvm)

### 25 bootstrap iterations (manually)
set.seed(290875)
n <- nrow(bodyfat)
bs25 <- rmultinom(25, n, rep(1, n)/n)
cvm <- cvrisk(model, folds = bs25, papply = lapply)
print(cvm)
mstop(cvm)
plot(cvm)

### same by default
set.seed(290875)
cvrisk(model, papply = lapply)

### 25 bootstrap iterations (using cv)
set.seed(290875)
bs25_2 <- cv(model.weights(model), type="bootstrap")
all(bs25 == bs25_2)
```

```
  ### trees
  blackbox <- blackboost(DEXfat ~ ., data = bodyfat)
  cvtree <- cvrisk(blackbox, papply = lapply)
  plot(cvtree)


  ### cvrisk in parallel modes:

  ## Not run: ## parallel:::mclapply only runs properly on unix systems
    cvrisk(model)

## End(Not run)

  ## Not run: ## infrastructure needs to be set up in advance
    cl <- makeCluster(25) # e.g. to run cvrisk on 25 nodes via PVM
    myApply <- function(X, FUN, ...) {
      myFun <- function(...) {
          library("mboost") # load mboost on nodes
          FUN(...)
      }
      ## further set up steps as required
      parLapply(cl = cl, X, myFun, ...)
    }
    cvrisk(model, papply = myApply)
    stopCluster(cl)

## End(Not run)
```

Family *Gradient Boosting Families*

Description

boost_family objects provide a convenient way to specify loss functions and corresponding risk functions to be optimized by one of the boosting algorithms implemented in this package.

Usage

```
Family(ngradient, loss = NULL, risk = NULL,
       offset = function(y, w)
           optimize(risk, interval = range(y),
                    y = y, w = w)$minimum,
       check_y = function(y) y,
       weights = c("any", "none", "zeroone", "case"),
       nuisance = function() return(NA),
       name = "user-specified", fW = NULL,
       response = function(f) NA,
       rclass = function(f) NA)
```

```
AdaExp()
AUC()
Binomial(link = c("logit", "probit"), ...)
GaussClass()
GaussReg()
Gaussian()
Huber(d = NULL)
Laplace()
Poisson()
GammaReg(nuirange = c(0, 100))
CoxPH()
QuantReg(tau = 0.5, qoffset = 0.5)
ExpectReg(tau = 0.5)
NBinomial(nuirange = c(0, 100))
PropOdds(nuirange = c(-0.5, -1), offrange = c(-5, 5))
Weibull(nuirange = c(0, 100))
Loglog(nuirange = c(0, 100))
Lognormal(nuirange = c(0, 100))
Gehan()
Hurdle(nuirange = c(0, 100))
Multinomial()
```

Arguments

ngradient	a function with arguments y, f and w implementing the *negative* gradient of the loss function (which is to be minimized).
loss	an optional loss function with arguments y and f.
risk	an optional risk function with arguments y, f and w to be minimized (!), the weighted mean of the loss function by default.
offset	a function with argument y and w (weights) for computing a *scalar* offset.
fW	transformation of the fit for the diagonal weights matrix for an approximation of the boosting hat matrix for loss functions other than squared error.
response	inverse link function of a GLM or any other transformation on the scale of the response.
rclass	function to derive class predictions from conditional class probabilities (for models with factor response variable).
check_y	a function for checking and transforming the class / mode of a response variable.
nuisance	a function for extracting nuisance parameters from the family.
weights	a character indicating what type of weights are allowed. These can be either arbitrary (non-negative) weights code"any", only zero and one weights "zeroone", (non-negative) interger weights "case", or no weights are allowed "none".
name	a character giving the name of the loss function for pretty printing.
link	link function for binomial family. Alternatively, one may supply the name of a distribution (for example link = "norm"), parameters of which may be specified via the ... argument.

d	delta parameter for Huber loss function. If omitted, it is chosen adaptively.
tau	the quantile or expectile to be estimated, a number strictly between 0 and 1.
qoffset	quantile of response distribution to be used as offset.
nuirange	a vector containing the end-points of the interval to be searched for the minimum risk w.r.t. the nuisance parameter. In case of PropOdds, the starting values for the nuisance parameters.
offrange	interval to search for offset in.
...	additional arguments to link functions.

Details

The boosting algorithm implemented in mboost minimizes the (weighted) empirical risk function `risk(y, f, w)` with respect to `f`. By default, the risk function is the weighted sum of the loss function `loss(y, f)` but can be chosen arbitrarily. The `ngradient(y, f)` function is the negative gradient of `loss(y, f)` with respect to `f`.

Pre-fabricated functions for the most commonly used loss functions are available as well. Buehlmann and Hothorn (2007) give a detailed overview of the available loss functions. The `offset` function returns the population minimizers evaluated at the response, i.e., $1/2 \log(p/(1-p))$ for `Binomial()` or `AdaExp()` and $(\sum w_i)^{-1} \sum w_i y_i$ for `Gaussian()` and the median for `Huber()` and `Laplace()`. A short summary of the available families is given in the following paragraphs:

`AdaExp()`, `Binomial()` and `AUC()` implement families for binary classification. `AdaExp()` uses the exponential loss, which essentially leads to the AdaBoost algorithm of Freund and Schapire (1996). `Binomial()` implements the negative binomial log-likelihood of a logistic regression model as loss function. Thus, using `Binomial` family closely corresponds to fitting a logistic model. Alternative link functions can be specified via the name of the corresponding distribution, for example `link = "cauchy"` lead to pcauchy used as link function. This feature is still experimental and not well tested.

However, the coefficients resulting from boosting with family `Binomial(link = "logit")` are $1/2$ of the coefficients of a logit model obtained via glm. This is due to the internal recoding of the response to -1 and $+1$ (see below). However, Buehlmann and Hothorn (2007) argue that the family `Binomial` is the preferred choice for binary classification. For binary classification problems the response `y` has to be a `factor`. Internally `y` is re-coded to -1 and $+1$ (Buehlmann and Hothorn 2007). `AUC()` uses $1 - AUC(y, f)$ as the loss function. The area under the ROC curve (AUC) is defined as $AUC = (n_{-1} n_1)^{-1} \sum_{i:y_i=1} \sum_{j:y_j=-1} I(f_i > f_j)$. Since this is not differentiable in `f`, we approximate the jump function $I((f_i - f_j) > 0)$ by the distribution function of the triangular distribution on $[-1, 1]$ with mean 0, similar to the logistic distribution approximation used in Ma and Huang (2005).

`Gaussian()` is the default family in mboost. It implements L_2Boosting for continuous response. Note that families `GaussReg()` and `GaussClass()` (for regression and classification) are deprecated now. `Huber()` implements a robust version for boosting with continuous response, where the Huber-loss is used. `Laplace()` implements another strategy for continuous outcomes and uses the L_1-loss instead of the L_2-loss as used by `Gaussian()`.

`Poisson()` implements a family for fitting count data with boosting methods. The implemented loss function is the negative Poisson log-likelihood. Note that the natural link function $\log(\mu) = \eta$ is assumed. The default step-site `nu = 0.1` is probably too large for this family (leading to infinite residuals) and smaller values are more appropriate.

`GammaReg()` implements a family for fitting nonnegative response variables. The implemented loss function is the negative Gamma log-likelihood with logarithmic link function (instead of the natural link).

`CoxPH()` implements the negative partial log-likelihood for Cox models. Hence, survival models can be boosted using this family.

`QuantReg()` implements boosting for quantile regression, which is introduced in Fenske et al. (2009). `ExpectReg` works in analogy, only for expectiles, which were introduced to regression by Newey and Powell (1987).

Families with an additional scale parameter can be used for fitting models as well: `PropOdds()` leads to proportional odds models for ordinal outcome variables (Schmid et al., 2011). When using this family, an ordered set of threshold parameters is re-estimated in each boosting iteration. An example is given below which also shows how to obtain the thresholds. `NBinomial()` leads to regression models with a negative binomial conditional distribution of the response. `Weibull()`, `Loglog()`, and `Lognormal()` implement the negative log-likelihood functions of accelerated failure time models with Weibull, log-logistic, and lognormal distributed outcomes, respectively. Hence, parametric survival models can be boosted using these families. For details see Schmid and Hothorn (2008) and Schmid et al. (2010).

`Gehan()` implements rank-based estimation of survival data in an accelerated failure time model. The loss function is defined as the sum of the pairwise absolute differences of residuals. The response needs to be defined as `Surv(y, delta)`, where `y` is the observed survial time (subject to censoring) and `delta` is the non-censoring indicator (see `Surv` for details). For details on `Gehan()` see Johnson and Long (2011).

Hurdle models for zero-inflated count data can be fitted by using a combination of the `Binomial()` and `Hurdle()` families. While the `Binomial()` family allows for fitting the zero-generating process of the Hurdle model, `Hurdle()` fits a negative binomial regression model to the non-zero counts. Note that the specification of the Hurdle model allows for using `Binomial()` and `Hurdle()` independently of each other.

Linear or additive multinomial logit models can be fitted using `Multinomial()`; although is family requires some extra effort for model specification (see example). More specifically, the predictor must be in the form of a linear array model (see `%O%`). Note that this family does not work with tree-based base-learners at the moment. The class corresponding to the last level of the factor coding of the response is used as reference class.

Value

An object of class `boost_family`.

Warning

The coefficients resulting from boosting with family `Binomial` are $1/2$ of the coefficients of a logit model obtained via `glm`. This is due to the internal recoding of the response to -1 and $+1$ (see above).

For `AUC()`, variables should be centered and scaled and observations with weight > 0 must not contain missing values. The estimated coefficients for `AUC()` have no probabilistic interpretation.

Author(s)

`ExpectReg()` was donated by Fabian Sobotka. `AUC()` was donated by Fabian Scheipl.

References

Peter Buehlmann and Torsten Hothorn (2007), Boosting algorithms: regularization, prediction and model fitting. *Statistical Science*, **22**(4), 477–505.

Nora Fenske, Thomas Kneib, and Torsten Hothorn (2011), Identifying risk factors for severe childhood malnutrition by boosting additive quantile regression. *Journal of the American Statistical Association*, **106**:494-510.

Yoav Freund and Robert E. Schapire (1996), Experiments with a new boosting algorithm. In *Machine Learning: Proc. Thirteenth International Conference*, 148–156.

Shuangge Ma and Jian Huang (2005), Regularized ROC method for disease classification and biomarker selection with microarray data. *Bioinformatics*, **21**(24), 4356–4362.

Whitney K. Newey and James L. Powell (1987), Asymmetric least squares estimation and testing. *Econometrika*, **55**, 819–847.

Matthias Schmid and Torsten Hothorn (2008), Flexible boosting of accelerated failure time models. *BMC Bioinformatics*, **9**(269).

Matthias Schmid, Sergej Potapov, Annette Pfahlberg, and Torsten Hothorn (2010). Estimation and regularization techniques for regression models with multidimensional prediction functions. *Statistics and Computing*, **20**, 139–150.

Schmid, M., T. Hothorn, K. O. Maloney, D. E. Weller and S. Potapov (2011): Geoadditive regression modeling of stream biological condition. *Environmental and Ecological Statistics*, **18**(4), 709–733.

Benjamin Hofner, Andreas Mayr, Nikolay Robinzonov and Matthias Schmid (2014). Model-based Boosting in R: A Hands-on Tutorial Using the R Package mboost. *Computational Statistics*, **29**, 3–35.
http://dx.doi.org/10.1007/s00180-012-0382-5

Available as vignette via: `vignette(package = "mboost", "mboost_tutorial")`

Brent A. Johnson and Qi Long (2011) Survival ensembles by the sum of pairwise differences with application to lung cancer microarray studies. *Annals of Applied Statistics*, **5**, 1081–1101.

See Also

`mboost` for the usage of `Family`s. See `boost_family-class` for objects resulting from a call to `Family`.

Examples

```
### Define a new family
MyGaussian <- function(){
       Family(ngradient = function(y, f, w = 1) y - f,
       loss = function(y, f) (y - f)^2,
       name = "My Gauss Variant")
}
```

```
############################################################
## Do not run and check these examples automatically as
## they take some time

### Proportional odds model
data(iris)
iris$Species <- factor(iris$Species, ordered = TRUE)
if (require("MASS")) {
    (mod.polr <- polr(Species  ~ Sepal.Length, data = iris))
}
mod.PropOdds <- glmboost(Species  ~ Sepal.Length, data = iris,
                         family = PropOdds(nuirange = c(-0.5, 3)))
mstop(mod.PropOdds) <- 1000
## thresholds are treated as nuisance parameters, to extract these use
nuisance(mod.PropOdds)
## effect estimate
coef(mod.PropOdds)["Sepal.Length"]
## make thresholds comparable to a model without intercept
nuisance(mod.PropOdds) - coef(mod.PropOdds)["(Intercept)"] -
    attr(coef(mod.PropOdds), "offset")

## End(Not run and test)


### Multinomial logit model via a linear array model
## One needs to convert the data to a list
myiris <- as.list(iris)
## ... and define a dummy vector with one factor level less
## than the outcome, which is used as reference category.
myiris$class <- factor(levels(iris$Species)[-nlevels(iris$Species)])
## Now fit the linear array model
mlm <- mboost(Species ~ bols(Sepal.Length, df = 2) %O%
                        bols(class, df = 2, contrasts.arg = "contr.dummy"),
              data = myiris,
              family = Multinomial())
coef(mlm) ## one should use more boosting iterations.
head(round(pred <- predict(mlm, type = "response"), 2))

## Prediction with new data:
newdata <- as.list(iris[1,])
## One always needs to keep the dummy vector class as above!
newdata$class <- factor(levels(iris$Species)[-nlevels(iris$Species)])
pred2 <- predict(mlm, type = "response", newdata = newdata)
## check results
pred[1, ]
pred2

############################################################
## Do not run and check these examples automatically as
## they take some time

## Compare results with nnet::multinom
if (require("nnet")) {
```

```
    mlmn <- multinom(Species ~ Sepal.Length, data = iris)
    max(abs(fitted(mlm[1000], type = "response") -
            fitted(mlmn, type = "prob")))

}

## End(Not run and test)
```

FP *Fractional Polynomials*

Description

Fractional polynomials transformation for continuous covariates.

Usage

```
FP(x, p = c(-2, -1, -0.5, 0.5, 1, 2, 3), scaling = TRUE)
```

Arguments

x	a numeric vector.
p	all powers of x to be included.
scaling	a logical indicating if the measurements are scaled prior to model fitting.

Details

A fractional polynomial refers to a model $\sum_{j=1}^{k}(\beta_j x^{p_j} + \gamma_j x^{p_j} \log(x)) + \beta_{k+1} \log(x) + \gamma_{k+1} \log(x)^2$, where the degree of the fractional polynomial is the number of non-zero regression coefficients β and γ.

Value

A matrix including all powers p of x, all powers p of `log(x)`, and `log(x)`.

References

Willi Sauerbrei and Patrick Royston (1999), Building multivariable prognostic and diagnostic models: transformation of the predictors by using fractional polynomials. *Journal of the Royal Statistical Society A*, **162**, 71–94.

See Also

`gamboost` to fit smooth models, `bbs` for P-spline base-learners

Examples

```
data("bodyfat", package = "TH.data")
tbodyfat <- bodyfat

### map covariates into [1, 2]
indep <- names(tbodyfat)[-2]
tbodyfat[indep] <- lapply(bodyfat[indep], function(x) {
    x <- x - min(x)
    x / max(x) + 1
})

### generate formula
fpfm <- as.formula(paste("DEXfat ~ ",
    paste("FP(", indep, ", scaling = FALSE)", collapse = "+")))
fpfm

### fit linear model
bf_fp <- glmboost(fpfm, data = tbodyfat,
                  control = boost_control(mstop = 3000))

### when to stop
mstop(aic <- AIC(bf_fp))
plot(aic)

### coefficients
cf <- coef(bf_fp[mstop(aic)])
length(cf)
cf[abs(cf) > 0]
```

---

gamboost *Gradient Boosting with Smooth Components*

Description

Gradient boosting for optimizing arbitrary loss functions, where component-wise smoothing procedures are utilized as base-learners.

Usage

```
gamboost(formula, data = list(),
         baselearner = c("bbs", "bols", "btree", "bss", "bns"),
         dfbase = 4, ...)
```

Arguments

formula	a symbolic description of the model to be fit.
data	a data frame containing the variables in the model.

baselearner	a character specifying the component-wise base learner to be used: bbs means P-splines with a B-spline basis (see Schmid and Hothorn 2008), bols linear models and btree boosts stumps. bss and bns are deprecated. Component-wise smoothing splines have been considered in Buehlmann and Yu (2003) and Schmid and Hothorn (2008) investigate P-splines with a B-spline basis. Kneib, Hothorn and Tutz (2009) also utilize P-splines with a B-spline basis, supplement them with their bivariate tensor product version to estimate interaction surfaces and spatial effects and also consider random effects base learners.
dfbase	an integer vector giving the degrees of freedom for the smoothing spline, either globally for all variables (when its length is one) or separately for each single covariate.
...	additional arguments passed to mboost_fit, including weights, offset, family and control. For default values see mboost_fit.

Details

A (generalized) additive model is fitted using a boosting algorithm based on component-wise univariate base-learners. The base-learners can either be specified via the formula object or via the baselearner argument (see bbs for an example). If the base-learners specified in formula differ from baselearner, the latter argument will be ignored. Furthermore, two additional base-learners can be specified in formula: bspatial for bivariate tensor product penalized splines and brandom for random effects.

Value

An object of class mboost with print, AIC, plot and predict methods being available.

References

Peter Buehlmann and Bin Yu (2003), Boosting with the L2 loss: regression and classification. *Journal of the American Statistical Association*, **98**, 324–339.

Peter Buehlmann and Torsten Hothorn (2007), Boosting algorithms: regularization, prediction and model fitting. *Statistical Science*, **22**(4), 477–505.

Thomas Kneib, Torsten Hothorn and Gerhard Tutz (2009), Variable selection and model choice in geoadditive regression models, *Biometrics*, **65**(2), 626–634.

Matthias Schmid and Torsten Hothorn (2008), Boosting additive models using component-wise P-splines as base-learners. *Computational Statistics \& Data Analysis*, **53**(2), 298–311.

Torsten Hothorn, Peter Buehlmann, Thomas Kneib, Mattthias Schmid and Benjamin Hofner (2010), Model-based Boosting 2.0. *Journal of Machine Learning Research*, **11**, 2109 – 2113.

Benjamin Hofner, Andreas Mayr, Nikolay Robinzonov and Matthias Schmid (2014). Model-based Boosting in R: A Hands-on Tutorial Using the R Package mboost. *Computational Statistics*, **29**, 3–35.
http://dx.doi.org/10.1007/s00180-012-0382-5

Available as vignette via: vignette(package = "mboost", "mboost_tutorial")

See Also

mboost for the generic boosting function and glmboost for boosted linear models and blackboost for boosted trees. See e.g. bbs for possible base-learners. See cvrisk for cross-validated stopping iteration. Furthermore see boost_control, Family and methods.

Examples

```
### a simple two-dimensional example: cars data
cars.gb <- gamboost(dist ~ speed, data = cars, dfbase = 4,
                    control = boost_control(mstop = 50))
cars.gb
AIC(cars.gb, method = "corrected")

### plot fit for mstop = 1, ..., 50
plot(dist ~ speed, data = cars)
tmp <- sapply(1:mstop(AIC(cars.gb)), function(i)
    lines(cars$speed, predict(cars.gb[i]), col = "red"))
lines(cars$speed, predict(smooth.spline(cars$speed, cars$dist),
                          cars$speed)$y, col = "green")

### artificial example: sinus transformation
x <- sort(runif(100)) * 10
y <- sin(x) + rnorm(length(x), sd = 0.25)
plot(x, y)
### linear model
lines(x, fitted(lm(y ~ sin(x) - 1)), col = "red")
### GAM
lines(x, fitted(gamboost(y ~ x,
                control = boost_control(mstop = 500))),
      col = "green")
```

---

glmboost *Gradient Boosting with Component-wise Linear Models*

Description

Gradient boosting for optimizing arbitrary loss functions where component-wise linear models are utilized as base-learners.

Usage

```
## S3 method for class 'formula'
glmboost(formula, data = list(), weights = NULL,
          na.action = na.pass, contrasts.arg = NULL,
          center = TRUE, control = boost_control(), ...)
## S3 method for class 'matrix'
glmboost(x, y, center = TRUE, weights = NULL,
```

```
          na.action = na.pass, control = boost_control(), ...)
## Default S3 method:
glmboost(x, ...)
```

Arguments

formula	a symbolic description of the model to be fit.
data	a data frame containing the variables in the model.
weights	an optional vector of weights to be used in the fitting process.
na.action	a function which indicates what should happen when the data contain NAs.
contrasts.arg	a list, whose entries are contrasts suitable for input to the contrasts replacement function and whose names are the names of columns of data containing factors. See model.matrix.default.
center	logical indicating of the predictor variables are centered before fitting.
control	a list of parameters controlling the algorithm.
x	design matrix. Sparse matrices of class Matrix can be used as well.
y	vector of responses.
...	additional arguments passed to mboost_fit, including weights, offset, family and control. For default values see mboost_fit.

Details

A (generalized) linear model is fitted using a boosting algorithm based on component-wise univariate linear models. The fit, i.e., the regression coefficients, can be interpreted in the usual way. The methodology is described in Buehlmann and Yu (2003), Buehlmann (2006), and Buehlmann and Hothorn (2007).

Value

An object of class glmboost with print, coef, AIC and predict methods being available. For inputs with longer variable names, you might want to change par("mai") before calling the plot method of glmboost objects visualizing the coefficients path.

References

Peter Buehlmann and Bin Yu (2003), Boosting with the L2 loss: regression and classification. *Journal of the American Statistical Association*, **98**, 324–339.

Peter Buehlmann (2006), Boosting for high-dimensional linear models. *The Annals of Statistics*, **34**(2), 559–583.

Peter Buehlmann and Torsten Hothorn (2007), Boosting algorithms: regularization, prediction and model fitting. *Statistical Science*, **22**(4), 477–505.

Torsten Hothorn, Peter Buehlmann, Thomas Kneib, Mattthias Schmid and Benjamin Hofner (2010), Model-based Boosting 2.0. *Journal of Machine Learning Research*, **11**, 2109–2113.

Benjamin Hofner, Andreas Mayr, Nikolay Robinzonov and Matthias Schmid (2014). Model-based Boosting in R: A Hands-on Tutorial Using the R Package mboost. *Computational Statistics*, **29**,

3–35.
http://dx.doi.org/10.1007/s00180-012-0382-5

Available as vignette via: `vignette(package = "mboost", "mboost_tutorial")`

See Also

`mboost` for the generic boosting function and `gamboost` for boosted additive models and `blackboost` for boosted trees. See `cvrisk` for cross-validated stopping iteration. Furthermore see `boost_control`, `Family` and `methods`

Examples

```
### a simple two-dimensional example: cars data
cars.gb <- glmboost(dist ~ speed, data = cars,
                    control = boost_control(mstop = 2000),
                    center = FALSE)
cars.gb

### coefficients should coincide
cf <- coef(cars.gb, off2int = TRUE)     ## add offset to intercept
coef(cars.gb) + c(cars.gb$offset, 0)    ## add offset to intercept (by hand)
signif(cf, 3)
signif(coef(lm(dist ~ speed, data = cars)), 3)
## almost converged. With higher mstop the results get even better

### now we center the design matrix for
### much quicker "convergence"
cars.gb_centered <- glmboost(dist ~ speed, data = cars,
                             control = boost_control(mstop = 2000),
                             center = TRUE)

## plot coefficient paths of glmboost
par(mfrow=c(1,2), mai = par("mai") * c(1, 1, 1, 2.5))
plot(cars.gb, main = "without centering")
plot(cars.gb_centered, main = "with centering")

### alternative loss function: absolute loss
cars.gbl <- glmboost(dist ~ speed, data = cars,
                     control = boost_control(mstop = 1000),
                     family = Laplace())
cars.gbl
coef(cars.gbl, off2int = TRUE)

### plot fit
par(mfrow = c(1,1))
plot(dist ~ speed, data = cars)
lines(cars$speed, predict(cars.gb), col = "red")     ## quadratic loss
lines(cars$speed, predict(cars.gbl), col = "green")  ## absolute loss

### Huber loss with adaptive choice of delta
cars.gbh <- glmboost(dist ~ speed, data = cars,
```

```
                    control = boost_control(mstop = 1000),
                    family = Huber())

lines(cars$speed, predict(cars.gbh), col = "blue")   ## Huber loss
legend("topleft", col = c("red", "green", "blue"), lty = 1,
       legend = c("Gaussian", "Laplace", "Huber"), bty = "n")

### sparse high-dimensional example that makes use of the matrix
### interface of glmboost and uses the matrix representation from
### package Matrix
library("Matrix")
n <- 100
p <- 10000
ptrue <- 10
X <- Matrix(0, nrow = n, ncol = p)
X[sample(1:(n * p), floor(n * p / 20))] <- runif(floor(n * p / 20))
beta <- numeric(p)
beta[sample(1:p, ptrue)] <- 10
y <- drop(X %*% beta + rnorm(n, sd = 0.1))
mod <- glmboost(y = y, x = X, center = TRUE) ### mstop needs tuning
coef(mod, which = which(beta > 0))
```

`IPCweights` *Inverse Probability of Censoring Weights*

Description

Compute weights for censored regression via the inverted probability of censoring principle.

Usage

```
IPCweights(x, maxweight = 5)
```

Arguments

`x`	an object of class `Surv`.
`maxweight`	the maximal value of the returned weights.

Details

Inverse probability of censoring weights are one possibility to fit models formulated in the *full data world* in the presence of censoring, i.e., the *observed data world*, see van der Laan and Robins (2003) for the underlying theory and Hothorn et al. (2006) for an application to survival analysis.

Value

A vector of numeric weights.

References

Mark J. van der Laan and James M. Robins (2003), *Unified Methods for Censored Longitudinal Data and Causality*, Springer, New York.

Torsten Hothorn, Peter Buehlmann, Sandrine Dudoit, Annette Molinaro and Mark J. van der Laan (2006), Survival ensembles. *Biostatistics* **7**(3), 355–373.

Peter Buehlmann and Torsten Hothorn (2007), Boosting algorithms: regularization, prediction and model fitting. *Statistical Science*, **22**(4), 477–505.

`mboost` *Model-based Gradient Boosting*

Description

Gradient boosting for optimizing arbitrary loss functions, where component-wise models are utilized as base-learners.

Usage

```
mboost(formula, data = list(), na.action = na.omit,
       baselearner = c("bbs", "bols", "btree", "bss", "bns"), ...)
mboost_fit(blg, response, weights = rep(1, NROW(response)), offset = NULL,
           family = Gaussian(), control = boost_control(), oobweights =
           as.numeric(weights == 0))
```

Arguments

`formula`	a symbolic description of the model to be fit.
`data`	a data frame containing the variables in the model.
`na.action`	a function which indicates what should happen when the data contain `NA`s.
`baselearner`	a character specifying the component-wise base learner to be used: `bbs` means P-splines with a B-spline basis (see Schmid and Hothorn 2008), `bols` linear models and `btree` boosts stumps. `bss` and `bns` are deprecated. Component-wise smoothing splines have been considered in Buehlmann and Yu (2003) and Schmid and Hothorn (2008) investigate P-splines with a B-spline basis. Kneib, Hothorn and Tutz (2009) also utilize P-splines with a B-spline basis, supplement them with their bivariate tensor product version to estimate interaction surfaces and spatial effects and also consider random effects base learners.
`blg`	a list of objects of class `blg`, as returned by all base-learners.
`response`	the response variable.
`weights`	a numeric vector of weights (optional).
`offset`	a numeric vector to be used as offset (optional).
`family`	a `Family` object.
`control`	a list of parameters controlling the algorithm. For more details see `boost_control`.

oobweights	an additional vector of out-of-bag weights, which is used for the out-of-bag risk (i.e., if boost_control(risk = "oobag")). This argument is also used internally by cvrisk.
...	additional arguments passed to mboost_fit, including weights, offset, family and control.

Details

The function implements component-wise functional gradient boosting in a generic way. Basically, the algorithm is initialized with a function for computing the negative gradient of the loss function (via its family argument) and one or more base-learners (given as blg). Usually blg and response are computed in the functions gamboost, glmboost, blackboost or mboost.

The algorithm minimized the in-sample empirical risk defined as the weighted sum (by weights) of the loss function (corresponding to the negative gradient) evaluated at the data.

The structure of the model is determined by the structure of the base-learners. If more than one base-learner is given, the model is additive in these components.

Base-learners can be specified via a formula interface (function mboost) or as a list of objects of class bl, see bols.

oobweights is a vector used internally by cvrisk. When carrying out cross-validation to determine the optimal stopping iteration of a boosting model, the default value of oobweights (out-of-bag weights) assures that the cross-validated risk is computed using the same observation weights as those used for fitting the boosting model. It is strongly recommended to leave this argument unspecified.

Note that the more convenient modelling interfaces gamboost, glmboost and blackboost all call mboost directly.

Value

An object of class mboost with print, AIC, plot and predict methods being available.

References

Peter Buehlmann and Bin Yu (2003), Boosting with the L2 loss: regression and classification. *Journal of the American Statistical Association*, **98**, 324–339.

Peter Buehlmann and Torsten Hothorn (2007), Boosting algorithms: regularization, prediction and model fitting. *Statistical Science*, **22**(4), 477–505.

Torsten Hothorn, Peter Buehlmann, Thomas Kneib, Mattthias Schmid and Benjamin Hofner (2010), Model-based Boosting 2.0. *Journal of Machine Learning Research*, **11**, 2109–2113.

Yoav Freund and Robert E. Schapire (1996), Experiments with a new boosting algorithm. In *Machine Learning: Proc. Thirteenth International Conference*, 148–156.

Jerome H. Friedman (2001), Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, **29**, 1189–1232.

Benjamin Hofner, Andreas Mayr, Nikolay Robinzonov and Matthias Schmid (2014). Model-based Boosting in R: A Hands-on Tutorial Using the R Package mboost. *Computational Statistics*, **29**, 3–35.
http://dx.doi.org/10.1007/s00180-012-0382-5

Available as vignette via: `vignette(package = "mboost", "mboost_tutorial")`

See Also

`glmboost` for boosted linear models and `blackboost` for boosted trees. See e.g. `bbs` for possible base-learners. See `cvrisk` for cross-validated stopping iteration. Furthermore see `boost_control`, `Family` and `methods`.

Examples

```
data("bodyfat", package = "TH.data")

### formula interface: additive Gaussian model with
### a non-linear step-function in `age', a linear function in `waistcirc'
### and a smooth non-linear smooth function in `hipcirc'
mod <- mboost(DEXfat ~ btree(age) + bols(waistcirc) + bbs(hipcirc),
              data = bodyfat)
layout(matrix(1:6, nc = 3, byrow = TRUE))
plot(mod, main = "formula")

### the same
with(bodyfat,
     mod <- mboost_fit(list(btree(age), bols(waistcirc), bbs(hipcirc)),
                       response = DEXfat))
plot(mod, main = "base-learner")
```

`methods` *Methods for Gradient Boosting Objects*

Description

Methods for models fitted by boosting algorithms.

Usage

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

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

## S3 method for class 'mboost'
coef(object, which = NULL,
    aggregate = c("sum", "cumsum", "none"), ...)
## S3 method for class 'glmboost'
coef(object, which = NULL,
     aggregate = c("sum", "cumsum", "none"), off2int = FALSE, ...)
```

```
## S3 method for class 'mboost'
x[i, return = TRUE, ...]
mstop(x) <- value

## S3 method for class 'mboost'
AIC(object, method = c("corrected", "classical", "gMDL"),
    df = c("trace", "actset"), ..., k = 2)

## S3 method for class 'mboost'
mstop(object, ...)
## S3 method for class 'gbAIC'
mstop(object, ...)
## S3 method for class 'cvrisk'
mstop(object, ...)

## S3 method for class 'mboost'
predict(object, newdata = NULL,
        type = c("link", "response", "class"), which = NULL,
        aggregate = c("sum", "cumsum", "none"), ...)
## S3 method for class 'glmboost'
predict(object, newdata = NULL,
        type = c("link", "response", "class"), which = NULL,
        aggregate = c("sum", "cumsum", "none"), ...)

## S3 method for class 'mboost'
fitted(object, ...)

## S3 method for class 'mboost'
residuals(object, ...)
## S3 method for class 'mboost'
resid(object, ...)

## S3 method for class 'glmboost'
variable.names(object, which = NULL, usedonly = FALSE, ...)
## S3 method for class 'mboost'
variable.names(object, which = NULL, usedonly = FALSE, ...)

## S3 method for class 'mboost'
extract(object, what = c("design", "penalty", "lambda", "df",
                         "coefficients", "residuals",
                         "variable.names", "bnames", "offset",
                         "nuisance", "weights", "index", "control"),
        which = NULL, ...)
## S3 method for class 'glmboost'
extract(object, what = c("design", "coefficients", "residuals",
                         "variable.names", "bnames", "offset",
                         "nuisance", "weights", "control"),
```

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

## S3 method for class 'mboost'
logLik(object, ...)
## S3 method for class 'gamboost'
hatvalues(model, ...)
## S3 method for class 'glmboost'
hatvalues(model, ...)

## S3 method for class 'mboost'
selected(object, ...)

## S3 method for class 'mboost'
risk(object, ...)

## S3 method for class 'mboost'
nuisance(object)
```

Arguments

object	objects of class glmboost, gamboost, blackboost or gbAIC.
x	objects of class glmboost or gamboost.
model	objects of class mboost
newdata	optionally, a data frame in which to look for variables with which to predict. In case the model was fitted using the matrix interface to glmboost, newdata must be a matrix as well (an error is given otherwise).
which	a subset of base-learners to take into account for computing predictions or coefficients. If which is given (as an integer vector or characters corresponding to base-learners) a list or matrix is returned.
usedonly	logical. Indicating whether all variable names should be returned or only those selected in the boosting algorithm.
type	the type of prediction required. The default is on the scale of the predictors; the alternative "response" is on the scale of the response variable. Thus for a binomial model the default predictions are of log-odds (probabilities on logit scale) and type = "response" gives the predicted probabilities. The "class" option returns predicted classes.
aggregate	a character specifying how to aggregate predictions or coefficients of single base-learners. The default returns the prediction or coefficient for the final number of boosting iterations. "cumsum" returns a list with matrices (one per base-learner) with the cumulative coefficients for all iterations simultaneously (in columns). "none" returns a list of matrices where the jth columns of the respective matrix contains the predictions of the base-learner of the jth boosting iteration (and zero if the base-learner is not selected in this iteration).

off2int	logical. Indicating whether the offset should be added to the intercept (if there is any) or if the offset is returned as attribute of the coefficient (default).
i	integer. Index specifying the model to extract. If i is smaller than the initial mstop, a subset is used. If i is larger than the initial mstop, additional boosting steps are performed until step i is reached. See details for more information.
value	integer. See i.
return	a logical indicating whether the changed object is returned.
method	a character specifying if the corrected AIC criterion or a classical (-2 logLik + k * df) should be computed.
df	a character specifying how degrees of freedom should be computed: trace defines degrees of freedom by the trace of the boosting hat matrix and actset uses the number of non-zero coefficients for each boosting iteration.
k	numeric, the *penalty* per parameter to be used; the default k = 2 is the classical AIC. Only used when method = "classical".
what	a character specifying the quantities to extract. Depending on object this can be a subset of "design" (default; design matrix), "penalty" (penalty matrix), "lambda" (smoothing parameter), "df" (degrees of freedom), "coefficients", "residuals", "variable.names", "bnames" (names of the base-learners), "offset", "nuisance", "weights", "index" (index of ties used to expand the design matrix) and "control". In future versions additional extractors might be specified.
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 where 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.
...	additional arguments passed to callies.

Details

These functions can be used to extract details from fitted models. print shows a dense representation of the model fit and summary gives a more detailed representation.

The function coef extracts the regression coefficients of a linear model fitted using the glmboost function or an additive model fitted using the gamboost. Per default, only coefficients of selected base-learners are returned. However, any desired coefficient can be extracted using the which argument (see examples for details). Per default, the coefficient of the final iteration is returned (aggregate = "sum") but it is also possible to return the coefficients from all iterations simultaniously (aggregate = "cumsum"). If aggregate = "none" is specified, the coefficients of the *selected* base-learners are returned (see examples below). For models fitted via glmboost with option center = TRUE the intercept is rarely selected. However, it is implicitly estimated through the centering of the design matrix. In this case the intercept is always returned except which is specified such that the intercept is not selected. See examples below.

The predict function can be used to predict the status of the response variable for new observations whereas fitted extracts the regression fit for the observations in the learning sample. For predict newdata can be specified, otherwise the fitted values are returned. If which is specified, marginal effects of the corresponding base-learner(s) are returned. The argument type can be used to make predictions on the scale of the link (i.e., the linear predictor $X\beta$), the response (i.e. $h(X\beta)$, where h is the response function) or the class (in case of classification). Furthermore, the predictions can be aggregated analogously to coef by setting aggregate to either sum (default; predictions of the final iteration are given), cumsum (predictions of all iterations are returned simultaniously) or none (change of prediction in each iteration). If applicable the offset is added to the predictions. If marginal predictions are requested the offset is attached to the object via attr(..., "offset") as adding the offset to one of the marginal predictions doesn't make much sense.

The [.mboost function can be used to enhance or restrict a given boosting model to the specified boosting iteration i. Note that in both cases the original x will be changed to reduce the memory footprint. If the boosting model is enhanced by specifying an index that is larger than the initial mstop, only the missing i - mstop steps are fitted. If the model is restricted, the spare steps are not dropped, i.e., if we increase i again, these boosting steps are immediately available. Alternatively, the same operation can be done by mstop(x) <- i.

The residuals function can be used to extract the residuals (i.e., the negative gradient of the current iteration). resid is is an alias for residuals.

Variable names (including those of interaction effects specified via by in a base-learner) can be extracted using the generic function variable.names, which has special methods for boosting objects.

The generic extract function can be used to extract various characteristics of a fitted model or a base-learner. Note that the sometimes a penalty function is returned (e.g. by extract(bols(x), what = "penalty")) even if the estimation is unpenalized. However, in this case the penalty paramter lambda is set to zero. If a matrix is returned by extract one can to set asmatrix = TRUE if the returned matrix should be coerced to class matrix. If asmatrix = FALSE one might get a sparse matrix as implemented in package Matrix. If one requests the design matrix (what = "design") expand = TRUE expands the resulting matrix by taking the duplicates handeled via index into account.

The ids of base-learners selected during the fitting process can be extracted using selected(). The nuisance() method extracts nuisance parameters from the fit that are handled internally by the corresponding family object, see "boost_family". The risk() function can be used to extract the computed risk (either the "inbag" risk or the "oobag" risk, depending on the control argument; see boost_control).

For (generalized) linear and additive models, the AIC function can be used to compute both the classical AIC (only available for familiy = Binomial() and familiy = Poisson()) and corrected AIC (Hurvich et al., 1998, only available when family = Gaussian() was used). Details on the used approximations for the hat matrix can be found in Buehlmann and Hothorn (2007). The AIC is useful for the determination of the optimal number of boosting iterations to be applied (which can be extracted via mstop). The degrees of freedom are either computed via the trace of the boosting hat matrix (which is rather slow even for moderate sample sizes) or the number of variables (non-zero coefficients) that entered the model so far (faster but only meaningful for linear models fitted via gamboost (see Hastie, 2007)). For a discussion of the use of AIC based stopping see also Mayr, Hofner and Schmid (2012).

In addition, the general Minimum Description Length criterion (Buehlmann and Yu, 2006) can be computed using function AIC.

Note that `logLik` and `AIC` only make sense when the corresponding `Family` implements the appropriate loss function.

Warning

The coefficients resulting from boosting with family `Binomial` are $1/2$ of the coefficients of a logit model obtained via `glm`. This is due to the internal recoding of the response to -1 and $+1$ (see `Binomial`).

Note

The `[.mboost` function changes the original object, i.e. `gbmodel[10]` changes `gbmodel` directly!

References

Benjamin Hofner, Andreas Mayr, Nikolay Robinzonov and Matthias Schmid (2014). Model-based Boosting in R: A Hands-on Tutorial Using the R Package mboost. *Computational Statistics*, **29**, 3–35.
http://dx.doi.org/10.1007/s00180-012-0382-5

Clifford M. Hurvich, Jeffrey S. Simonoff and Chih-Ling Tsai (1998), Smoothing parameter selection in nonparametric regression using an improved Akaike information criterion. *Journal of the Royal Statistical Society, Series B*, **20**(2), 271–293.

Peter Buehlmann and Torsten Hothorn (2007), Boosting algorithms: regularization, prediction and model fitting. *Statistical Science*, **22**(4), 477–505.

Trevor Hastie (2007), Discussion of "Boosting algorithms: Regularization, prediction and model fitting" by Peter Buehlmann and Torsten Hothorn. *Statistical Science*, **22**(4), 505.

Peter Buehlmann and Bin Yu (2006), Sparse boosting. *Journal of Machine Learning Research*, **7**, 1001–1024.

Andreas Mayr, Benjamin Hofner, and Matthias Schmid (2012). The importance of knowing when to stop - a sequential stopping rule for component-wise gradient boosting. *Methods of Information in Medicine*, **51**, 178–186.
DOI: http://dx.doi.org/10.3414/ME11-02-0030

See Also

`gamboost`, `glmboost` and `blackboost` for model fitting.

`plot.mboost` for plotting methods.

`cvrisk` for cross-validated stopping iteration.

Examples

```
### a simple two-dimensional example: cars data
cars.gb <- glmboost(dist ~ speed, data = cars,
                    control = boost_control(mstop = 2000),
                    center = FALSE)
cars.gb
```

```
### initial number of boosting iterations
mstop(cars.gb)

### AIC criterion
aic <- AIC(cars.gb, method = "corrected")
aic

### extract coefficients for glmboost
coef(cars.gb)
coef(cars.gb, off2int = TRUE)        # offset added to intercept
coef(lm(dist ~ speed, data = cars))  # directly comparable

cars.gb_centered <- glmboost(dist ~ speed, data = cars,
                             center = TRUE)
selected(cars.gb_centered)           # intercept never selected
coef(cars.gb_centered)               # intercept implicitly estimated
                                     # and thus returned
## intercept is internally corrected for mean-centering
- mean(cars$speed) * coef(cars.gb_centered, which="speed") # = intercept
# not asked for intercept thus not returned
coef(cars.gb_centered, which="speed")
# explicitly asked for intercept
coef(cars.gb_centered, which=c("Intercept", "speed"))

### enhance or restrict model
cars.gb <- gamboost(dist ~ speed, data = cars,
                    control = boost_control(mstop = 100, trace = TRUE))
cars.gb[10]
cars.gb[100, return = FALSE] # no refitting required
cars.gb[150, return = FALSE] # only iterations 101 to 150
                             # are newly fitted

### coefficients for optimal number of boosting iterations
coef(cars.gb[mstop(aic)])
plot(cars$dist, predict(cars.gb[mstop(aic)]),
     ylim = range(cars$dist))
abline(a = 0, b = 1)

### example for extraction of coefficients
set.seed(1907)
n <- 100
x1 <- rnorm(n)
x2 <- rnorm(n)
x3 <- rnorm(n)
x4 <- rnorm(n)
int <- rep(1, n)
y <- 3 * x1^2 - 0.5 * x2 + rnorm(n, sd = 0.1)
data <- data.frame(y = y, int = int, x1 = x1, x2 = x2, x3 = x3, x4 = x4)

model <- gamboost(y ~ bols(int, intercept = FALSE) +
                      bbs(x1, center = TRUE, df = 1) +
                      bols(x1, intercept = FALSE) +
                      bols(x2, intercept = FALSE) +
```

```
                    bols(x3, intercept = FALSE) +
                    bols(x4, intercept = FALSE),
                  data = data, control = boost_control(mstop = 500))

coef(model) # standard output (only selected base-learners)
coef(model,
     which = 1:length(variable.names(model))) # all base-learners
coef(model, which = "x1") # shows all base-learners for x1

cf1 <- coef(model, which = c(1,3,4), aggregate = "cumsum")
tmp <- sapply(cf1, function(x) x)
matplot(tmp, type = "l", main = "Coefficient Paths")

cf1_all <- coef(model, aggregate = "cumsum")
cf1_all <- lapply(cf1_all, function(x) x[, ncol(x)]) # last element
## same as coef(model)

cf2 <- coef(model, aggregate = "none")
cf2 <- lapply(cf2, rowSums) # same as coef(model)

### example continued for extraction of predictions

yhat <- predict(model) # standard prediction; here same as fitted(model)
p1 <- predict(model, which = "x1") # marginal effects of x1
orderX <- order(data$x1)
## rowSums needed as p1 is a matrix
plot(data$x1[orderX], rowSums(p1)[orderX], type = "b")

## better: predictions on a equidistant grid
new_data <- data.frame(x1 = seq(min(data$x1), max(data$x1), length = 100))
p2 <- predict(model, newdata = new_data, which = "x1")
lines(new_data$x1, rowSums(p2), col = "red")

### extraction of model characteristics
extract(model, which = "x1")  # design matrices for x1
extract(model, what = "penalty", which = "x1") # penalty matrices for x1
extract(model, what = "lambda", which = "x1") # df and corresponding lambda for x1
     ## note that bols(x1, intercept = FALSE) is unpenalized

extract(model, what = "bnames")  ## name of complete base-learner
extract(model, what = "variable.names") ## only variable names
variable.names(model)            ## the same

### extract from base-learners
extract(bbs(x1), what = "design")
extract(bbs(x1), what = "penalty")
## weights and lambda can only be extracted after using dpp
weights <- rep(1, length(x1))
extract(bbs(x1)$dpp(weights), what = "lambda")
```

---

plot *Plot effect estimates of boosting models*

Description

Plot coefficient plots for `glmboost` models and partial effect plots for all other `mboost` models.

Usage

```
## S3 method for class 'glmboost'
plot(x, main = deparse(x$call), col = NULL,
     off2int = FALSE, ...)

## S3 method for class 'mboost'
plot(x, which = NULL, newdata = NULL,
     type = "b", rug = TRUE, rugcol = "black",
     ylim = NULL, xlab = NULL, ylab = expression(f[partial]),
     add = FALSE, ...)

## S3 method for class 'mboost'
lines(x, which = NULL, type = "l", rug = FALSE, ...)
```

Arguments

`x`	object of class `glmboost` or an object inheriting from `mboost` for plotting.
`main`	a title for the plot.
`col`	(a vector of) colors for plotting the lines representing the coefficient paths.
`off2int`	logical indicating whether the offset should be added to the intercept (if there is any) or if the offset is neglected for plotting (default).
`which`	a subset of base-learners used for plotting. If `which` is given (as an integer vector or characters corresponding to base-learners) only the corresponding partial effect plots are depicted. Per default all selected base-learners are plotted.
`newdata`	optionally, a data frame in which to look for variables with which to make predictions that are then plotted. This is especially useful if the data that was used to fit the model shows some larger gaps as effect plots are linearly interpolated between observations. For an example using `newdata` see below.
`type`	character string giving the type of plot desired. Per default, points and lines are plotted (`"b"`). Other useful options are points (`"p"`) or lines (`"l"`). See `plot.default` for details.
`rug`	logical. Should a rug be added to the x-axis?
`rugcol`	color for the rug.
`ylim`	the y limits of the plot.
`xlab`	a label for the x axis.
`ylab`	a label for the y axis.
`add`	logical. Should the plot be added to the previous plot?
`...`	Additional arguments to the `plot` functions. E.g. one can specify the x limits `xlim` or the color of the plot using `col`.

Details

The coefficient paths for `glmboost` models show how the coefficient estimates evolve with increasing `mstop`. Each line represents one parameter estimate. Parameter estimates are only depicted when they they are selected at any time in the boosting model. Parameters that are not selected are droped from the figure (see example).

Models specified with `gamboost` or `mboost` are plotted as partial effects. Only the effect of the current bossting iteration is depicted instead of the coefficient paths as for linear models. The function `lines` is just a wrapper to `plot(... , add = TRUE)` where per default the effect is plotted as line and the `rug` is set to `FALSE`.

Spatial effects can be also plotted using the function `plot` for mboost models (using `lattice` graphics). More complex effects reuquire manual plotting: One needs to predict the effects on a disired grid and plot the effect estimates.

Value

A plot of the fitted model.

References

Benjamin Hofner, Andreas Mayr, Nikolay Robinzonov and Matthias Schmid (2014). Model-based Boosting in R: A Hands-on Tutorial Using the R Package mboost. *Computational Statistics*, **29**, 3–35.
http://dx.doi.org/10.1007/s00180-012-0382-5

See Also

`mboost_methods` for further methods.

Examples

```
### a simple example: cars data with one random variable
set.seed(1234)
cars$z <- rnorm(50)

########################################
## Plot linear models
########################################

## fit a linear model
cars.lm <- glmboost(dist ~ speed + z, data = cars)

## plot coefficient paths of glmboost
par(mfrow = c(3, 1), mar = c(4, 4, 4, 8))
plot(cars.lm,
     main = "Coefficient paths (offset not included)")
plot(cars.lm, off2int = TRUE,
     main = "Coefficient paths (offset included in intercept)")

## plot coefficient paths only for the first 15 steps,
```

```
## i.e., bevore z is selected
mstop(cars.lm) <- 15
plot(cars.lm, off2int = TRUE, main = "z is not yet selected")


########################################
## Plot additive models; basics
########################################

## fit an additive model
cars.gam <- gamboost(dist ~ speed + z, data = cars)

## plot effects
par(mfrow = c(1, 2), mar = c(4, 4, 0.1, 0.1))
plot(cars.gam)

## use same y-lims
plot(cars.gam, ylim = c(-50, 50))

## plot only the effect of speed
plot(cars.gam, which = "speed")
## as partial matching is used we could also use
plot(cars.gam, which = "sp")


########################################
## More complex plots
########################################

## Let us use more boosting iterations and compare the effects.

## We change the plot type and plot both effects in one figure:
par(mfrow = c(1, 1), mar = c(4, 4, 4, 0.1))
mstop(cars.gam) <- 100
plot(cars.gam, which = 1, col = "red", type = "l", rug = FALSE,
     main = "Compare effect for various models")

## Now the same model with 1000 iterations
mstop(cars.gam) <- 1000
lines(cars.gam, which = 1, col = "grey", lty = "dotted")

## There are some gaps in the data. Use newdata to get a smoother curve:
newdata <- data.frame(speed = seq(min(cars$speed), max(cars$speed),
                                  length = 200))
lines(cars.gam, which = 1, col = "grey", lty = "dashed",
      newdata = newdata)

## The model with 1000 steps seems to overfit the data.
## Usually one should use e.g. cross-validation to tune the model.

## Finally we refit the model using linear effects as comparison
cars.glm <- gamboost(dist ~ speed + z, baselearner = bols, data = cars)
lines(cars.glm, which = 1, col = "black")
```

```
## We see that all effects are more or less linear.

## Add a legend
legend("topleft", title = "Model",
       legend = c("... with mstop = 100", "... with mstop = 1000",
         "... with linear effects"),
       lty = c("solid", "dashed", "solid"),
       col = c("red", "grey", "black"))
```

---

stabsel *Stability Selection*

Description

Selection of influential variables or model components with error control.

Usage

```
## a method to compute stability selection paths for fitted mboost models
## S3 method for class 'mboost'
stabsel(x, cutoff, q, PFER,
        folds = subsample(model.weights(x), B = B),
        B = ifelse(sampling.type == "MB", 100, 50),
        assumption = c("unimodal", "r-concave", "none"),
        sampling.type = c("SS", "MB"),
        papply = mclapply, verbose = TRUE, FWER, eval = TRUE, ...)

## just a wrapper to stabsel(p, ..., eval = FALSE)
## S3 method for class 'mboost'
stabsel_parameters(p, ...)
```

Arguments

x, p	an fitted model of class "mboost".
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.
folds	a weight matrix with number of rows equal to the number of observations, see cvrisk and subsample. 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.

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 code"none".
sampling.type	use sampling scheme of of Shah & Samworth (2013), i.e., with complementarty pairs (sampling.type = "SS"), or the original sampling scheme of Meinshausen & Buehlmann (2010).
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 complementray pairs in the former case this leads to $2B$ subsamples.
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.
FWER	deprecated. Only for compatibility with older versions, use PFER instead.
eval	logical. Determines whether stability selection is evaluated (eval = TRUE; default) or if only the parameter combination is returned.
...	additional arguments to parallel apply methods such as mclapply and to cvrisk.

Details

For details see stabsel in package **stabs** and Hofner et al. (2014).

Value

An object of class stabsel with a special print method. The object has the following elements:

phat	selection probabilities.
selected	elements with maximal selection probability greater cutoff.
max	maximum of selection probabilities.
cutoff	cutoff used.
q	average number of selected variables used.
PFER	per-family error rate.
sampling.type	the sampling type used for stability selection.
assumption	the assumptions made on the selection probabilities.
call	the call.

References

B. Hofner, L. Boccuto and M. Goeker (2014), Controlling false discoveries in high-dimensional situations: Boosting with stability selection. *Technical Report*, arXiv:1411.1285. http://arxiv.org/abs/1411.1285.

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 and stabsel_parameters

Examples

```
  ## make data set available
  data("bodyfat", package = "TH.data")
  ## set seed
  set.seed(1234)

  ### low-dimensional example
  mod <- glmboost(DEXfat ~ ., data = bodyfat)

  ## compute cutoff ahead of running stabsel to see if it is a sensible
  ## parameter choice.
  ##   p = ncol(bodyfat) - 1 (= Outcome) + 1 ( = Intercept)
  stabsel_parameters(q = 3, PFER = 1, p = ncol(bodyfat) - 1 + 1,
                     sampling.type = "MB")
  ## the same:
  stabsel(mod, q = 3, PFER = 1, sampling.type = "MB", eval = FALSE)

############################################################
## Do not run and check these examples automatically as
## they take some time (~ 10 seconds depending on the system)

  ## now run stability selection
  (sbody <- stabsel(mod, q = 3, PFER = 1, sampling.type = "MB"))
  opar <- par(mai = par("mai") * c(1, 1, 1, 2.7))
  plot(sbody)
  par(opar)

  plot(sbody, type = "maxsel", ymargin = 6)

## End(Not run and test)
```

---

survFit *Survival Curves for a Cox Proportional Hazards Model*

Description

Computes the predicted survivor function for a Cox proportional hazards model.

Usage

```
## S3 method for class 'mboost'
survFit(object, newdata = NULL, ...)
## S3 method for class 'survFit'
plot(x, xlab = "Time", ylab = "Probability", ...)
```

Arguments

object	an object of class mboost which is assumed to have a CoxPH family component.
newdata	an optional data frame in which to look for variables with which to predict the survivor function.
x	an object of class survFit for plotting.
xlab	the label of the x axis.
ylab	the label of the y axis.
...	additional arguments passed to callies.

Details

If newdata = NULL, the survivor function of the Cox proportional hazards model is computed for the mean of the covariates used in the blackboost, gamboost, or glmboost call. The Breslow estimator is used for computing the baseline survivor function. If newdata is a data frame, the predict method of object, along with the Breslow estimator, is used for computing the predicted survivor function for each row in newdata.

Value

An object of class survFit containing the following components:

surv	the estimated survival probabilities at the time points given in time.
time	the time points at which the survivor functions are evaluated.
n.event	the number of events observed at each time point given in time.

See Also

gamboost, glmboost and blackboost for model fitting.

Examples

```
library("survival")
data("ovarian", package = "survival")

fm <- Surv(futime,fustat) ~ age + resid.ds + rx + ecog.ps
fit <- glmboost(fm, data = ovarian, family = CoxPH(),
    control=boost_control(mstop = 500))

S1 <- survFit(fit)
S1
newdata <- ovarian[c(1,3,12),]
S2 <- survFit(fit, newdata = newdata)
S2

plot(S1)
```

Index