

Package ‘mboost’

January 2, 2012

Title Model-Based Boosting

Version 2.1-1

Date 2011-11-28

Author Torsten Hothorn, Peter Buehlmann, Thomas Kneib, Matthias Schmid and Benjamin Hofner

Maintainer Torsten Hothorn <Torsten.Hothorn@R-project.org>

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 (>= 2.10.0), methods, stats

Imports Matrix, survival, splines, lattice

Suggests multicore, party (>= 0.9-9993), ipred, MASS, fields, BayesX,gbm

LazyLoad yes

LazyData yes

License GPL-2

Repository CRAN

Date/Publication 2011-11-29 07:30:55

R topics documented:

mboost-package	2
baselearners	4
birds	14
blackboost	16
bodyfat	17
boost_control	19

boost_family-class	20
cvrisk	21
Family	24
FP	27
gamboost	29
glmboost	31
IPCweights	34
mboost	34
methods	37
stabsel	43
survFit	45
Westbc	46
wpbc	47

Index	50
--------------	-----------

mboost-package	<i>mboost: Model-Based Boosting</i>
----------------	-------------------------------------

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.

Details

Package:	mboost
Type:	Package
Version:	2.1-1
Date:	2011-11-28
License:	GPL-2
LazyLoad:	yes
LazyData:	yes

This package is intended for modern regression modelling 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 algorithm (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.

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 if package `multicore` is available.

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

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.

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")
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

```

bols(..., by = NULL, index = NULL, intercept = TRUE, df = NULL,
      lambda = 0, contrasts.arg = "contr.treatment")
bbs(..., by = NULL, index = NULL, knots = 20, boundary.knots = NULL,
     degree = 3, differences = 2, df = 4, lambda = NULL, center = FALSE,

```

```

    cyclic = FALSE)
bspatial(..., df = 6)
brad(..., by = NULL, index = NULL, knots = 100, df = 4, lambda = NULL,
      covFun = stationary.cov,
      args = list(Covariance="Matern", smoothness = 1.5, theta=NULL))
brandom(..., df = 4)
btree(..., tree_controls = ctree_control(stump = TRUE,
                                         mincriterion = 0,
                                         savesplitstats = FALSE))
bmono(..., constraint = c("increasing", "decreasing", "convex", "concave"),
      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")
bmrfr(..., by = NULL, index = NULL, bnd = NULL, df = 4, lambda = NULL,
      center = FALSE)
buser(X, K = NULL, by = NULL, index = NULL, df = 4, lambda = NULL)
b11 %+% b12
b11 %X% b12
b11 %0% b12

```

Arguments

...	one or more predictor variables or one data frame of predictor variables.
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=TRUE, the effect is centered around its unpenalized part and therefore any positive number is admissible for df.
lambda	smoothing penalty, computed from df when df is specified.
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 = TRUE 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).
cyclic	if cyclic = TRUE the fitted values coincide at the boundaries (useful for cyclic covariates such as day time etc.).
covFun	the covariance function (i.e. radial basis) needed to compute the basis functions. Per default the stationary.cov function (package <code>fields</code>) is used.
args	a named list of arguments to be passed to <code>cov</code> function. Thus strongly dependent on the specified <code>cov</code> function.
contrasts.arg	a named list of characters suitable for input to the contrasts replacement function, see model.matrix , or a single character which is then used as contrasts for all factors in this base-learner (with the exception of factors in <code>by</code>).
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".
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.
bnd	Object of class <code>bnd</code> , in which the boundaries of a map are defined and from which neighborhood relations can be construed. 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 (<code>by</code> 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.
b11	a linear base-learner or a list of linear base-learners.
b12	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 `bol`s).

In `bol`s(`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 `bol`s(`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 `bol`s (see Hofner et al., 2011). When `df` is given, a ridge estimator with `df` degrees of freedom (trace of hat matrix) 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. 2011) 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).

`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 too 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`. Note that `brandom(...)` is essentially a wrapper to `bol`s(..., `df = 4`), i.e., a wrapper that utilizes ridge-penalized categorical effects. For possible arguments and defaults see `bol`s.

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) unstable 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` 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).

`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 `bnf`. 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 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).

Three global options affect the base-learners: `option("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.

option("mboost_indexmin") is an integer for the sample size required to optimize model fitting by taking ties into account. option("mboost_dftraceS"), which is also TRUE by default, indicates that the trace of the smoother matrix is used as degrees of freedom. If FALSE, an alternative is used (see Hofner et al., 2011).

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 (2011).

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 $X = \text{kronecker}(X2, X1)$, then `b11 %X% b12` with design matrices `X1` and `X2`, respectively, can be used to efficiently compute Ridge-estimates following Currie, Durban, Eilers (2006). In 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.

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).

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.
- 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*, in press. doi:10.1198/jcgs.2011.09220
- Preliminary version: Department of Statistics, Technical Report Nr. 72, LMU Muenchen. <http://epub.ub.uni-muenchen.de/11243/>
- B. Hofner, J. Mueller, and T. Hothorn (2011), Monotonicity-Constrained Species Distribution Models, *Ecology*, 92:1895-1901.
- Thomas Kneib, Torsten Hothorn and Gerhard Tutz (2009), Variable selection and model choice in geoadaptive regression models, *Biometrics*, **65**(2), 626–634.

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)
attributes(spline1)

knots.x2 <- quantile(x2, c(0.25, 0.5, 0.75))
spline2 <- bbs(x2, knots = knots.x2, df = 5)
attributes(spline2)

attributes(ols3 <- bols(x3))
attributes(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 = 400))

```

```

coef(model, which = 1) # one base-learner for x1 and x2
coef(model, which = 2:3) # two separate base-learners for x1 and x2

### 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)
attributes(spline3)

## specify number of knots separately
form2 <- y ~ bspatial(x1, x2, knots = list(x1 = 12, x2 = 12))

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

### random intercept
id <- factor(rep(1:10, each = 5))
ranef <- brandom(id)
attributes(ranef)

## 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)
ranef <- brandom(id, by = z)
attributes(ranef)

### 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 = 1000))
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),
  control = boost_control(mstop = 1000))
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 = 2500))
cf <- unlist(coef(mod))
cf[1] <- cf[1] + mod$offset
cf
coef(lm(y ~ x, data = tmpdata))

### quicker and better with (mean-) centering
tmpdata$x_center <- tmpdata$x - mean(tmpdata$x)
mod_center <- gamboost(y ~ bols(int, intercept = FALSE) +
  bols(x_center, intercept = FALSE),
  data = tmpdata,
  control = boost_control(mstop = 500))
cf_center <- unlist(coef(mod_center, which=1:2))
## due to the shift in x direction we need to subtract
## beta_1 * mean(x) to get the correct intercept
cf_center[1] <- cf_center[1] + mod_center$offset -
  cf_center[2] * mean(tmpdata$x)
cf_center
coef(lm(y ~ x, data = tmpdata))

### 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)

### 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)%0%
              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")

## 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)))

```

Description

Environmental variables and bird counts for identifying suitable bird habitats

Usage

```
data("birds")
```

Format

A data frame with 258 observations on the following 10 variables.

GST Growing stock per grid

DBH Mean diameter of the largest three trees

AOT Age of oldest tree

AFS Age of forest stand

DWC Amount of dead wood of conifers

LOG Amount of logs per grid

x_gk grid location, x coordinate

y_gk grid location, y coordinate

SG4 observed number of birds from structural gild 4: Requirement of regeneration (Phylloscopus trochilus, Aegithalos caudatus)

SG5 observed number of birds from structural gild 5: Requirement of regeneration combined with planted conifers (Phylloscopus collybita, Turdus merula, Sylvia atricapilla).

Details

Counts of breeding bird communities collected at 258 observation plots in a northern Bavarian forest district are the response variable of interest. Along with the number of birds in two structural gilds, 6 covariates are given here and one is interested in quantifying their impact on habitat suitability.

Source

Joerg Mueller (2005). Forest structures as key factor for beetle and bird communities in beech forests. PhD thesis, Munich University of Technology. <http://mediatum.ub.tum.de>

References

Thomas Kneib and Joerg Mueller and Torsten Hothorn (2008), Spatial smoothing techniques for the assessment of habitat suitability, *Environmental and Ecological Statistics*, **15**(3), 343–364.

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(),
           tree_controls = 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.
tree_controls	an object of class "TreeControl", which can be obtained using <code>ctree_control</code> . 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 <code>mboost_fit</code> , including weights, offset, family and control. For default values see <code>mboost_fit</code> .

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.

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")
```

bodyfat

Prediction of Body Fat by Skinfold Thickness, Circumferences, and Bone Breadths

Description

For 71 healthy female subjects, body fat measurements and several anthropometric measurements are available for predictive modelling of body fat.

Usage

```
data("bodyfat")
```

Format

A data frame with 71 observations on the following 10 variables.

age age in years.

DEXfat body fat measured by DXA, response variable.

waistcirc waist circumference.

hipcirc hip circumference.

elbowbreadth breadth of the elbow.

kneebreadth breadth of the knee.

anthro3a sum of logarithm of three anthropometric measurements.

anthro3b sum of logarithm of three anthropometric measurements.

anthro3c sum of logarithm of three anthropometric measurements.

anthro4 sum of logarithm of three anthropometric measurements.

Details

Garcia et al. (2005) report on the development of predictive regression equations for body fat content by means of common anthropometric measurements which were obtained for 71 healthy German women. In addition, the women's body composition was measured by Dual Energy X-Ray Absorptiometry (DXA). This reference method is very accurate in measuring body fat but finds little applicability in practical environments, mainly because of high costs and the methodological efforts needed. Therefore, a simple regression equation for predicting DXA measurements of body fat is of special interest for the practitioner. Backward-elimination was applied to select important variables from the available anthropometrical measurements, and Garcia (2005) report a final linear model utilizing hip circumference, knee breadth and a compound covariate which is defined as the sum of log chin skinfold, log triceps skinfold and log subscapular skinfold.

Source

Ada L. Garcia, Karen Wagner, Torsten Hothorn, Corinna Koebnick, Hans-Joachim F. Zunft and Ulrike Trippo (2005), Improved prediction of body fat by measuring skinfold thickness, circumferences, and bone breadths. *Obesity Research*, **13**(3), 626–634.

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

Examples

```
data("bodyfat", package = "mboost")

### final model proposed by Garcia et al. (2005)
fmod <- lm(DEXfat ~ hipcirc + anthro3a + kneebreadth, data = bodyfat)
coef(fmod)

### plot additive model for same variables
amod <- gamboost(DEXfat ~ hipcirc + anthro3a + kneebreadth,
                 data = bodyfat, baselearner = "bbs")
```

```
layout(matrix(1:3, ncol = 3))
plot(amod[mstop(AIC(amod, "corrected"))], ask = FALSE)
```

boost_control	<i>Control Hyper-parameters for Boosting Algorithms</i>
---------------	---

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"),
              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 <code>family = Poisson()</code> and a smaller value is better.
risk	a character indicating how the empirical risk should be computed for each boosting iteration. <code>inbag</code> leads to risks computed for the learning sample (i.e., all non-zero weights), <code>oobag</code> to risks based on the out-of-bag (all observations with zero weights) and <code>none</code> to no risk computations at all.
center	deprecated. A logical indicating if the numerical covariates should be mean centered before fitting. Only implemented for <code>glmboost</code> . In <code>blackboost</code> centering is not needed. In <code>gamboost</code> centering is only needed if <code>bols</code> 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 <code>glmboost</code> 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()`

 cvrisk

Cross-Validation

Description

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

Usage

```
cvrisk(object, folds = cv(model.weights(object)),
       grid = 1:mstop(object),
       papply = if (require("multicore")) mclapply else lapply,
       fun = NULL, ...)
cv(weights, type = c("bootstrap", "kfold", "subsampling"),
   B = ifelse(type == "kfold", 10, 25), prob = 0.5, strata = NULL)
```

Arguments

object	an object of class <code>mboost</code> .
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 <code>cv</code> and defaults to 25 bootstrap samples.
grid	a vector of stopping parameters the empirical risk is to be evaluated for.
papply	(parallel) apply function. In the absence of package <code>multicore</code> sequential computations via <code>lapply</code> are performed. Alternatively, parallel computing via <code>mclapply</code> (package <code>multicore</code>), or <code>clusterApplyLB</code> (package <code>snow</code>) can be used. In the latter case, usually more setup is needed (see example for some details).
fun	if <code>fun</code> is <code>NULL</code> , the out-of-sample risk is returned. <code>fun</code> , as a function of <code>object</code> , may extract any other characteristic of the cross-validated models. These are returned as is.
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 <code>weights</code> for stratification.
...	additional arguments passed to <code>mclapply</code> eventually.

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 `fold`s matrix.

If package `multicore` is available, `cvrisk` can be easily used in parallel on cores/processors available by specifying `papply = mcapply`. 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.

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 = "mboost")

### 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)
cmtree <- cvrisk(blackbox, papply = lapply)
plot(cmtree)

### cvrisk in parallel modes:

## Not run: ## multicore only runs properly on unix systems
library("multicore")
cvrisk(model)

## End(Not run)

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

## End(Not run)

```

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))
```

Arguments

ngradient	a function with arguments y, f and w implementing the <i>negative</i> 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 <i>scalar</i> 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 if weights are allowed.
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 <code>link = "norm"</code>), parameters of which may be specified via the <code>...</code> 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 $\text{risk}(y, f, w)$ with respect to f . By default, the risk function is the weighted sum of the loss function $\text{loss}(y, f)$ but can be chosen arbitrarily. The $\text{ngradient}(y, f)$ function is the negative gradient of $\text{loss}(y, f)$ 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_2 Boosting 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-size $\nu = 0.1$ 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. When using this family, an ordered set of threshold parameters is re-estimated in each boosting iteration. `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).

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.

See Also

[mboost](#) for the usage of FamilyS. See [boost_family-class](#) for objects resulting from a call to Family.

Examples

```
Laplace()

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

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 . See [mfp](#) for the reference implementation.

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 = "mboost")
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
fpm <- as.formula(paste("DEXfat ~ ",
  paste("FP(", indep, ", scaling = FALSE)", collapse = "+")))
fpm

### fit linear model
bf_fp <- glmboost(fpm, 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, Matthias Schmid and Benjamin Hofner (2010), Model-based Boosting 2.0. *Journal of Machine Learning Research*, **11**, 2109 – 2113.

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, control = boost_control(), ...)
## Default S3 method:
glmboost(x, ...)
## S3 method for class 'glmboost'
plot(x, main = deparse(x$call), col = NULL,
     off2int = 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.
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 .
na.action	a function which indicates what should happen when the data contain NAs.
center	logical indicating of the predictor variables are centered before fitting.
control	a list of parameters controlling the algorithm.
x	design matrix or an object of class <code>glmboost</code> for plotting. Sparse matrices of class <code>Matrix</code> can be used as well.

<code>y</code>	vector of responses.
<code>main</code>	a title for the plot.
<code>col</code>	(a vector of) colors for plotting the lines representing the coefficient paths.
<code>off2int</code>	logical indicating whether the offset should be added to the intercept (if there is any) or if the offset is neglected for plotting (default).
<code>...</code>	additional arguments passed to <code>mboost_fit</code> , including <code>weights</code> , <code>offset</code> , <code>family</code> and <code>control</code> . For default values see <code>mboost_fit</code> .

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

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 = 5000),
                   center = FALSE)

cars.gb

### coefficients should coincide
```

```

coef(cars.gb) + c(cars.gb$offset, 0)
coef(lm(dist ~ speed, data = cars))

### plot fit
layout(matrix(1:2, ncol = 2))
plot(dist ~ speed, data = cars)
lines(cars$speed, predict(cars.gb), col = "red")

### 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)

par(mfrow=c(1,2))
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 = 5000),
                     family = Laplace())

cars.gbl

coef(cars.gbl) + c(cars.gbl$offset, 0)
lines(cars$speed, predict(cars.gbl), col = "green")

### Huber loss with adaptive choice of delta
cars.gbh <- glmboost(dist ~ speed, data = cars,
                     control = boost_control(mstop = 5000),
                     family = Huber())

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

### plot coefficient path of glmboost
par(mai = par("mai") * c(1, 1, 1, 2.5))
plot(cars.gb)

### sparse high-dimensional example
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(),
       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.
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 <code>blg</code> , 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 <code>boost_control</code> .
oobweights	an additional vector of out-of-bag weights (used internally by <code>cvrisk</code>).
...	additional arguments passed to <code>mboost_fit</code> , including <code>weights</code> , <code>offset</code> , <code>family</code> and <code>control</code> .

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.

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.

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 = "mboost")

### 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, ask = FALSE, main = "formula")

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

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, ...]

## 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 'mboost'
extract(object, what = c("design", "penalty", "lambda", "df",
                        "coefficients", "residuals", "bnames", "offset",
                        "nuisance", "weights", "index", "control"),
        which = NULL, ...)

## S3 method for class 'glmboost'
extract(object, what = c("design", "coefficients", "residuals",
                        "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'
nuisance(object)

```

Arguments

object	objects of class <code>glmboost</code> , <code>gamboost</code> , <code>blackboost</code> or <code>gbAIC</code> .
x	objects of class <code>glmboost</code> or <code>gamboost</code> .
model	objects of class <code>mboost</code>
newdata	optionally, a data frame in which to look for variables with which to predict. In case the model was fitted using the <code>matrix</code> interface to <code>glmboost</code> , <code>newdata</code> must be a <code>matrix</code> as well (an error is given otherwise).
which	a subset of base-learners to take into account for computing predictions or coefficients. If <code>which</code> is given (as an integer vector or characters corresponding to base-learners) a list or <code>matrix</code> is returned.
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 <code>type = "response"</code> gives the predicted probabilities. The <code>"class"</code> option returns predicted classes.
<code>aggregate</code>	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. <code>"cumsum"</code> returns a matrix with the predictions for all iterations simultaneously (in columns). <code>"none"</code> returns a list with matrices where the j th columns of the respective matrix contains the predictions of the base-learner of the j th boosting iteration (and zero if the base-learner is not selected in this iteration).
<code>off2int</code>	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).
<code>i</code>	integer. Index specifying the model to extract. If <code>i</code> is smaller than the initial <code>mstop</code> , a subset is used. If <code>i</code> is larger than the initial <code>mstop</code> , additional boosting steps are performed until step <code>i</code> is reached. See details for more information.
<code>return</code>	a logical indicating whether the changed object is returned.
<code>method</code>	a character specifying if the corrected AIC criterion or a classical ($-2 \log\text{Lik} + k * \text{df}$) should be computed.
<code>df</code>	a character specifying how degrees of freedom should be computed: <code>trace</code> defines degrees of freedom by the trace of the boosting hat matrix and <code>actset</code> uses the number of non-zero coefficients for each boosting iteration.
<code>k</code>	numeric, the <i>penalty</i> per parameter to be used; the default <code>k = 2</code> is the classical AIC. Only used when <code>method = "classical"</code> .
<code>what</code>	a character specifying the quantities to extract. Depending on object this can be a subset of <code>"design"</code> (default; design matrix), <code>"penalty"</code> (penalty matrix), <code>"lambda"</code> (smoothing parameter), <code>"df"</code> (degrees of freedom), <code>"coefficients"</code> , <code>"residuals"</code> , <code>"bnames"</code> (names of the base-learners), <code>"offset"</code> , <code>"nuisance"</code> , <code>"weights"</code> , <code>"index"</code> (index of ties used to expand the design matrix) and <code>"control"</code> . In future versions additional extractors might be specified.
<code>asmatrix</code>	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 <i>sparse</i> matrix). This option is only applicable if <code>extract</code> returns matrices, i.e., <code>what = "design"</code> or <code>what = "penalty"</code> .
<code>expand</code>	a logical indicating whether the design matrix should be expanded (default: FALSE). This is useful if ties were taken into account either manually (via argument <code>index</code> in a base-learner) or automatically for data sets with many observations. <code>expand = TRUE</code> is equivalent to <code>extract(B)[extract(B, what = "index"),]</code> for a base-learner <code>B</code> .
<code>...</code>	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 simultaneously (`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 simultaneously) 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 `residuals` function can be used to extract the residuals (i.e., the negative gradient of the current iteration). `resid` is an alias for `residuals`.

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.

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 parameter `lambda` is set to zero. If a matrix is returned by `extract` one can 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 handled 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`".

For (generalized) linear and additive models, the `AIC` function can be used to compute both the classical AIC (only available for `family = Binomial()` and `family = 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)).

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

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.

See Also

`gamboost`, `glmboost` and `blackboost` for model fitting. See `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) +
                  bols(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 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")

```

stabsel

Stability Selection

Description

Selection of influential variables or model components with error control.

Usage

```

stabsel(object, FWER = 0.05, cutoff, q,
        folds = cv(model.weights(object), type = "subsampling"),
        papply = if (require("multicore")) mclapply else lapply, ...)

```

Arguments

object	an mboost object.
FWER	family-wise error rate to be controlled by the selection procedure.
cutoff	cutoff between 0.5 and 1.
q	average number of selected variables.
folds	a weight matrix with number of rows equal to the number of observations, see cvrisk .
papply	(parallel) apply function. In the absence of package multicore sequential computations via lapply are performed. Alternatively, parallel computing via mclapply (package multicore), or clusterApplyLB (package snow) can be used. In the latter case, usually more setup is needed (see example for some details).
...	additional arguments to cvrisk .

Details

This function implements the "stability selection" procedure by Meinshausen and Bühlmann (2010). Either `cutoff` or `q` must be specified. The probability of selecting at least one non-influential variable (or model component) is less than FWER.

Value

An object of class `stabsel` with 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.
FWER	family-wise error rate.

References

N. Meinshausen and P. Bühlmann (2010), Stability selection. *Journal of the Royal Statistical Society, Series B*, **72**(4).

Examples

```
### (too) low-dimensional example
sbody <- stabsel(glmboost(DEXfat ~ ., data = bodyfat), q = 3,
                papply = lapply)

sbody
opar <- par(mai = par("mai") * c(1, 1, 1, 2.7))
plot(sbody)
par(opar)
```

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 <code>mboost</code> 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 <code>survFit</code> 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 <code>time</code> .
time	the time points at which the survivor functions are evaluated.
n.event	the number of events observed at each time point given in <code>time</code> .

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)
```

Westbc

Breast Cancer Gene Expression

Description

Gene expressions for 7129 genes in 49 breast cancer samples and the status of lymph node involvement.

Usage

```
data("Westbc")
```

Format

An list with two elements to be converted to class ExpressionSet (see package Biobase).

Details

A full description of the data can be found in West et al. (2001) and an application of boosted linear models is given by Buehlmann (2006).

Source

Mike West, Carrie Blanchette, Holly Dressman, Erich Huang, Seiichi Ishida, Rainer Spang, Harry Zuzan, John A. Olson Jr., Jeffrey R. Marks and Joseph R. Nevins (2001), Predicting the clinical status of human breast cancer by using gene expression profiles, *Proceedings of the National Academy of Sciences*, **98**, 11462-11467. <http://data.cgt.duke.edu/west.php>

References

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.

Examples

```
## Not run:
library("Biobase")
data("Westbc", package = "mboost")
westbc <- new("ExpressionSet",
             phenoData = new("AnnotatedDataFrame", data = Westbc$pheno),
             assayData = assayDataNew(exprs = Westbc$assay))

## End(Not run)
```

wpbc

Wisconsin Prognostic Breast Cancer Data

Description

Each record represents follow-up data for one breast cancer case. These are consecutive patients seen by Dr. Wolberg since 1984, and include only those cases exhibiting invasive breast cancer and no evidence of distant metastases at the time of diagnosis.

Usage

```
data("wpbc")
```

Format

A data frame with 198 observations on the following 34 variables.

`status` a factor with levels N (nonrecur) and R (recur)

`time` recurrence time (for `status == "R"`) or disease-free time (for `status == "N"`).

`mean_radius` radius (mean of distances from center to points on the perimeter) (mean).

`mean_texture` texture (standard deviation of gray-scale values) (mean).

`mean_perimeter` perimeter (mean).

`mean_area` area (mean).

`mean_smoothness` smoothness (local variation in radius lengths) (mean).

`mean_compactness` compactness (mean).

`mean_concavity` concavity (severity of concave portions of the contour) (mean).

`mean_concavepoints` concave points (number of concave portions of the contour) (mean).

mean_symmetry symmetry (mean).
 mean_fractaldim fractal dimension (mean).
 SE_radius radius (mean of distances from center to points on the perimeter) (SE).
 SE_texture texture (standard deviation of gray-scale values) (SE).
 SE_perimeter perimeter (SE).
 SE_area area (SE).
 SE_smoothness smoothness (local variation in radius lengths) (SE).
 SE_compactness compactness (SE).
 SE_concavity concavity (severity of concave portions of the contour) (SE).
 SE_concavepoints concave points (number of concave portions of the contour) (SE).
 SE_symmetry symmetry (SE).
 SE_fractaldim fractal dimension (SE).
 worst_radius radius (mean of distances from center to points on the perimeter) (worst).
 worst_texture texture (standard deviation of gray-scale values) (worst).
 worst_perimeter perimeter (worst).
 worst_area area (worst).
 worst_smoothness smoothness (local variation in radius lengths) (worst).
 worst_compactness compactness (worst).
 worst_concavity concavity (severity of concave portions of the contour) (worst).
 worst_concavepoints concave points (number of concave portions of the contour) (worst).
 worst_symmetry symmetry (worst).
 worst_fractaldim fractal dimension (worst).
 tsize diameter of the excised tumor in centimeters.
 pnodes number of positive axillary lymph nodes observed at time of surgery.

Details

The first 30 features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.

There are two possible learning problems: predicting status or predicting the time to recur.

1) Predicting field 2, outcome: R = recurrent, N = non-recurrent - Dataset should first be filtered to reflect a particular endpoint; e.g., recurrences before 24 months = positive, non-recurrence beyond 24 months = negative. - 86.3 previous version of this data.

2) Predicting Time To Recur (field 3 in recurrent records) - Estimated mean error 13.9 months using Recurrence Surface Approximation.

The data are originally available from the UCI machine learning repository, see <http://www.ics.uci.edu/~mllearn/databases/breast-cancer-wisconsin/>.

Source

W. Nick Street, Olvi L. Mangasarian and William H. Wolberg (1995). An inductive learning approach to prognostic prediction. In A. Prieditis and S. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 522–530, San Francisco, Morgan Kaufmann.

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

Examples

```
data("wpbc", package = "mboost")

### fit logistic regression model with 100 boosting iterations
coef(glmboost(status ~ ., data = wpbc[,colnames(wpbc) != "time"],
              family = Binomial()))
```

Index

- *Topic **classes**
 - boost_family-class, 20
- *Topic **datagen**
 - FP, 27
- *Topic **datasets**
 - birds, 14
 - bodyfat, 17
 - Westbc, 46
 - wdbc, 47
- *Topic **methods**
 - methods, 37
- *Topic **misc**
 - boost_control, 19
- *Topic **models**
 - baselearners, 4
 - blackboost, 16
 - cvrisk, 21
 - Family, 24
 - gamboost, 29
 - glmboost, 31
 - mboost, 34
 - mboost-package, 2
- *Topic **nonlinear**
 - gamboost, 29
 - mboost, 34
- *Topic **nonparametric**
 - mboost-package, 2
 - stabsel, 43
- *Topic **package**
 - mboost-package, 2
- *Topic **regression**
 - blackboost, 16
 - cvrisk, 21
 - glmboost, 31
- *Topic **smooth**
 - mboost-package, 2
- *Topic **survival**
 - IPCweights, 34
 - [.mboost (methods), 37
 - %+% (baselearners), 4
 - %0% (baselearners), 4
 - %X% (baselearners), 4
 - AdaExp (Family), 24
 - AIC, 30, 32, 36
 - AIC.mboost, 22
 - AIC.mboost (methods), 37
 - AUC (Family), 24
 - baselearners, 4
 - bbs, 28–30, 35, 36
 - bbs (baselearners), 4
 - Binomial, 41
 - Binomial (Family), 24
 - birds, 14
 - blackboost, 3, 9, 16, 19, 30, 32, 35, 36, 41, 45
 - bmono (baselearners), 4
 - bmrfr (baselearners), 4
 - bns (baselearners), 4
 - bodyfat, 17
 - bols, 3, 19, 29, 35
 - bols (baselearners), 4
 - boost_control, 17, 19, 30, 32, 35, 36
 - boost_family, 40
 - boost_family-class, 27
 - boost_family-class, 20
 - brad (baselearners), 4
 - brandom, 30
 - brandom (baselearners), 4
 - bspatial, 30
 - bspatial (baselearners), 4
 - bss (baselearners), 4
 - btree, 29, 35
 - btree (baselearners), 4
 - buser (baselearners), 4
 - coef, 32
 - coef.glmboost (methods), 37
 - coef.mboost (methods), 37

- contrasts, 5, 6
- cover.design, 8
- CoxPH, 45
- CoxPH (Family), 24
- ctree, 16
- ctree_control, 6, 16
- cv (cvrisk), 21
- cvrisk, 17, 21, 30, 32, 36, 41, 44
- drawmap, 8
- ExpectReg (Family), 24
- extract (methods), 37
- Family, 17, 20, 24, 30, 32, 35, 36, 41
- fitted.mboost (methods), 37
- FP, 27
- gam, 2
- gamboost, 3, 17, 19, 28, 29, 32, 35, 36, 39–41, 45
- GammaReg (Family), 24
- GaussClass (Family), 24
- Gaussian (Family), 24
- GaussReg (Family), 24
- gbm, 2, 16
- glm, 2, 25, 26, 41
- glmboost, 3, 17, 19, 30, 31, 35, 36, 38–41, 45
- hatvalues.gamboost (methods), 37
- hatvalues.glmboost (methods), 37
- Huber (Family), 24
- IPCweights, 34
- Laplace (Family), 24
- lapply, 21, 44
- lm, 2
- logLik.mboost (methods), 37
- Loglog (Family), 24
- Lognormal (Family), 24
- mboost, 10, 17, 19, 25–27, 30, 32, 34
- mboost-package, 2
- mboost_fit, 2, 3, 16, 29, 32, 35
- mboost_fit (mboost), 34
- mclapply, 21, 22, 44
- methods, 17, 30, 32, 36, 37
- mfp, 28
- model.matrix, 6
- model.matrix.default, 31
- mstop (methods), 37
- NBinomial (Family), 24
- nuisance (methods), 37
- pcauchy, 25
- plot, 30, 36
- plot.glmboost (glmboost), 31
- plot.survFit (survFit), 45
- Poisson (Family), 24
- predict, 16, 30, 32, 36, 45
- predict.blackboost (methods), 37
- predict.gamboost (methods), 37
- predict.glmboost (methods), 37
- predict.mboost (methods), 37
- print, 16, 30, 32, 36
- print.glmboost (methods), 37
- print.mboost (methods), 37
- PropOdds (Family), 24
- QuantReg (Family), 24
- randomForest, 2
- read.bnd, 6
- resid.mboost (methods), 37
- residuals.mboost (methods), 37
- selected (methods), 37
- show, boost_family-method
(boost_family-class), 20
- stabsel, 43
- stationary.cov, 6
- summary.mboost (methods), 37
- survFit, 45
- Weibull (Family), 24
- Westbc, 46
- wpbc, 47