Package: party (via r-universe)

August 17, 2024

Title A Laboratory for Recursive Partytioning

Date 2024-08-17

Version 1.3-17

Description A computational toolbox for recursive partitioning. The core of the package is ctree(), an implementation of conditional inference trees which embed tree-structured regression models into a well defined theory of conditional inference procedures. This non-parametric class of regression trees is applicable to all kinds of regression problems, including nominal, ordinal, numeric, censored as well as multivariate response variables and arbitrary measurement scales of the covariates. Based on conditional inference trees, cforest() provides an implementation of Breiman's random forests. The function mob() implements an algorithm for recursive partitioning based on parametric models (e.g. linear models, GLMs or survival regression) employing parameter instability tests for split selection. Extensible functionality for visualizing tree-structured regression models is available. The methods are described in Hothorn et al. (2006) <doi:10.1198/106186006X133933>, Zeileis et al. (2008) <doi:10.1198/106186008X319331> and Strobl et al. (2007) <doi:10.1186/1471-2105-8-25>.

Depends R (>= 3.0.0), methods, grid, stats, mvtnorm (>= 1.0-2), modeltools (>= 0.2-21), strucchange

LinkingTo mvtnorm

Imports survival (>= 2.37-7), coin (>= 1.1-0), zoo, sandwich (>= 1.1-1)

Suggests TH.data (>= 1.0-3), mlbench, colorspace, MASS, vcd, ipred, varImp, randomForest

LazyData yes

License GPL-2

URL http://party.R-forge.R-project.org

Repository https://r-forge.r-universe.dev

RemoteUrl https://github.com/r-forge/party

RemoteRef HEAD

RemoteSha a81e5a70ef70edff26d3b04da2fb551c0cce78f7

Contents

`BinaryTree Class` *Class "BinaryTree"*

Description

A class for representing binary trees.

Objects from the Class

Objects can be created by calls of the form `new("BinaryTree", ...)`. The most important slot is `tree`, a (recursive) list with elements

nodeID an integer giving the number of the node, starting with `1` in the root node.

weights the case weights (of the learning sample) corresponding to this node.

criterion a list with test statistics and p-values for each partial hypothesis.

terminal a logical specifying if this is a terminal node.

psplit primary split: a list with elements `variableID` (the number of the input variable splitted), `ordered` (a logical whether the input variable is ordered), `splitpoint` (the cutpoint or set of levels to the left), `splitstatistics` saves the process of standardized two-sample statistics the split point estimation is based on. The logical `toleft` determines if observations go left or right down the tree. For nominal splits, the slot `table` is a vector being greater zero if the corresponding level is available in the corresponding node.

ssplits a list of surrogate splits, each with the same elements as `psplit`.

prediction the prediction of the node: the mean for numeric responses and the conditional class probabilities for nominal or ordered respones. For censored responses, this is the mean of the logrank scores and useless as such.

left a list representing the left daughter node.

right a list representing the right daugther node.

Please note that this data structure may be subject to change in future releases of the package.

Slots

`data`**:** an object of class `"ModelEnv"`.

`responses`**:** an object of class `"VariableFrame"` storing the values of the response variable(s).

`cond_distr_response`**:** a function computing the conditional distribution of the response.

`predict_response`**:** a function for computing predictions.

`tree`**:** a recursive list representing the tree. See above.

`where`**:** an integer vector of length n (number of observations in the learning sample) giving the number of the terminal node the corresponding observations is element of.

`prediction_weights`**:** a function for extracting weights from terminal nodes.

`get_where`**:** a function for determining the number of terminal nodes observations fall into.

`update`**:** a function for updating weights.

Extends

Class `"BinaryTreePartition"`, directly.

Methods

`response(object, ...)`**:** extract the response variables the tree was fitted to.

`treeresponse(object, newdata = NULL, ...)`**:** compute statistics for the conditional distribution of the response as modelled by the tree. For regression problems, this is just the mean. For nominal or ordered responses, estimated conditional class probabilities are returned. Kaplan-Meier curves are computed for censored responses. Note that a list with one element for each observation is returned.

`Predict(object, newdata = NULL, ...)`**:** compute predictions.

`weights(object, newdata = NULL, ...)`: extract the weight vector from terminal nodes each element of the learning sample is element of (`newdata = NULL`) and for new observations, respectively.

`where(object, newdata = NULL, ...)`: extract the number of the terminal nodes each element of the learning sample is element of (`newdata = NULL`) and for new observations, respectively.

`nodes(object, where, ...)`: extract the nodes with given number (`where`).

`plot(x, ...)`: a plot method for `BinaryTree` objects, see `plot.BinaryTree`.

`print(x, ...)`: a print method for `BinaryTree` objects.

Examples

```
set.seed(290875)

airq <- subset(airquality, !is.na(Ozone))
airct <- ctree(Ozone ~ ., data = airq,
               controls = ctree_control(maxsurrogate = 3))

### distribution of responses in the terminal nodes
plot(airq$Ozone ~ as.factor(where(airct)))

### get all terminal nodes from the tree
nodes(airct, unique(where(airct)))

### extract weights and compute predictions
pmean <- sapply(weights(airct), function(w) weighted.mean(airq$Ozone, w))

### the same as
drop(Predict(airct))

### or
unlist(treeresponse(airct))

### don't use the mean but the median as prediction in each terminal node
pmedian <- sapply(weights(airct), function(w)
               median(airq$Ozone[rep(1:nrow(airq), w)]))

plot(airq$Ozone, pmean, col = "red")
points(airq$Ozone, pmedian, col = "blue")
```

---

cforest *Random Forest*

Description

An implementation of the random forest and bagging ensemble algorithms utilizing conditional inference trees as base learners.

Usage

```
cforest(formula, data = list(), subset = NULL, weights = NULL,
        controls = cforest_unbiased(),
        xtrafo = ptrafo, ytrafo = ptrafo, scores = NULL)
proximity(object, newdata = NULL)
```

Arguments

formula	a symbolic description of the model to be fit. Note that symbols like : and - will not work and the tree will make use of all variables listed on the rhs of formula.
data	an data frame containing the variables in the model.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional vector of weights to be used in the fitting process. Non-negative integer valued weights are allowed as well as non-negative real weights. Observations are sampled (with or without replacement) according to probabilities weights / sum(weights). The fraction of observations to be sampled (without replacement) is computed based on the sum of the weights if all weights are integer-valued and based on the number of weights greater zero else. Alternatively, weights can be a double matrix defining case weights for all ncol(weights) trees in the forest directly. This requires more storage but gives the user more control.
controls	an object of class ForestControl-class, which can be obtained using cforest_control (and its convenience interfaces cforest_unbiased and cforest_classical).
xtrafo	a function to be applied to all input variables. By default, the ptrafo function is applied.
ytrafo	a function to be applied to all response variables. By default, the ptrafo function is applied.
scores	an optional named list of scores to be attached to ordered factors.
object	an object as returned by cforest.
newdata	an optional data frame containing test data.

Details

This implementation of the random forest (and bagging) algorithm differs from the reference implementation in randomForest with respect to the base learners used and the aggregation scheme applied.

Conditional inference trees, see ctree, are fitted to each of the ntree (defined via cforest_control) bootstrap samples of the learning sample. Most of the hyper parameters in cforest_control regulate the construction of the conditional inference trees. Therefore you MUST NOT change anything you don't understand completely.

Hyper parameters you might want to change in cforest_control are:

1. The number of randomly preselected variables mtry, which is fixed to the value 5 by default here for technical reasons, while in randomForest the default values for classification and regression vary with the number of input variables.

2. The number of trees ntree. Use more trees if you have more variables.

3. The depth of the trees, regulated by mincriterion. Usually unstopped and unpruned trees are used in random forests. To grow large trees, set mincriterion to a small value.

The aggregation scheme works by averaging observation weights extracted from each of the ntree trees and NOT by averaging predictions directly as in randomForest. See Hothorn et al. (2004) for a description.

Predictions can be computed using predict. For observations with zero weights, predictions are computed from the fitted tree when newdata = NULL. While predict returns predictions of the same type as the response in the data set by default (i.e., predicted class labels for factors), treeresponse returns the statistics of the conditional distribution of the response (i.e., predicted class probabilities for factors). The same is done by predict(..., type = "prob"). Note that for multivariate responses predict does not convert predictions to the type of the response, i.e., type = "prob" is used.

Ensembles of conditional inference trees have not yet been extensively tested, so this routine is meant for the expert user only and its current state is rather experimental. However, there are some things available in cforest that can't be done with randomForest, for example fitting forests to censored response variables (see Hothorn et al., 2006a) or to multivariate and ordered responses.

Moreover, when predictors vary in their scale of measurement of number of categories, variable selection and computation of variable importance is biased in favor of variables with many potential cutpoints in randomForest, while in cforest unbiased trees and an adequate resampling scheme are used by default. See Hothorn et al. (2006b) and Strobl et al. (2007) as well as Strobl et al. (2009).

The proximity matrix is an $n \times n$ matrix P with P_{ij} equal to the fraction of trees where observations i and j are element of the same terminal node (when both i and j had non-zero weights in the same bootstrap sample).

Value

An object of class RandomForest-class.

References

Leo Breiman (2001). Random Forests. *Machine Learning*, 45(1), 5–32.

Torsten Hothorn, Berthold Lausen, Axel Benner and Martin Radespiel-Troeger (2004). Bagging Survival Trees. *Statistics in Medicine*, **23**(1), 77–91.

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

Torsten Hothorn, Kurt Hornik and Achim Zeileis (2006b). Unbiased Recursive Partitioning: A Conditional Inference Framework. *Journal of Computational and Graphical Statistics*, **15**(3), 651–674. Preprint available from https://www.zeileis.org/papers/Hothorn+Hornik+Zeileis-2006.pdf

Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis and Torsten Hothorn (2007). Bias in Random Forest Variable Importance Measures: Illustrations, Sources and a Solution. *BMC Bioinformatics*, **8**, 25. doi:10.1186/14712105825

Carolin Strobl, James Malley and Gerhard Tutz (2009). An Introduction to Recursive Partitioning: Rationale, Application, and Characteristics of Classification and Regression Trees, Bagging, and Random forests. *Psychological Methods*, **14**(4), 323–348.

Examples

```
set.seed(290875)

### honest (i.e., out-of-bag) cross-classification of
### true vs. predicted classes
data("mammoexp", package = "TH.data")
table(mammoexp$ME, predict(cforest(ME ~ ., data = mammoexp,
                           control = cforest_unbiased(ntree = 50)),
                           OOB = TRUE))

### fit forest to censored response
if (require("TH.data") && require("survival")) {

    data("GBSG2", package = "TH.data")
    bst <- cforest(Surv(time, cens) ~ ., data = GBSG2,
               control = cforest_unbiased(ntree = 50))

    ### estimate conditional Kaplan-Meier curves
    treeresponse(bst, newdata = GBSG2[1:2,], OOB = TRUE)

    ### if you can't resist to look at individual trees ...
    party:::prettytree(bst@ensemble[[1]], names(bst@data@get("input")))
}

### proximity, see ?randomForest
iris.cf <- cforest(Species ~ ., data = iris,
                   control = cforest_unbiased(mtry = 2))
iris.mds <- cmdscale(1 - proximity(iris.cf), eig = TRUE)
op <- par(pty="s")
pairs(cbind(iris[,1:4], iris.mds$points), cex = 0.6, gap = 0,
      col = c("red", "green", "blue")[as.numeric(iris$Species)],
      main = "Iris Data: Predictors and MDS of Proximity Based on cforest")
par(op)
```

`Conditional Inference Trees`
Conditional Inference Trees

Description

Recursive partitioning for continuous, censored, ordered, nominal and multivariate response variables in a conditional inference framework.

Usage

```
ctree(formula, data, subset = NULL, weights = NULL,
      controls = ctree_control(), xtrafo = ptrafo, ytrafo = ptrafo,
      scores = NULL)
```

Arguments

formula	a symbolic description of the model to be fit. Note that symbols like : and - will not work and the tree will make use of all variables listed on the rhs of formula.
data	a data frame containing the variables in the model.
subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional vector of weights to be used in the fitting process. Only non-negative integer valued weights are allowed.
controls	an object of class TreeControl, which can be obtained using ctree_control.
xtrafo	a function to be applied to all input variables. By default, the ptrafo function is applied.
ytrafo	a function to be applied to all response variables. By default, the ptrafo function is applied.
scores	an optional named list of scores to be attached to ordered factors.

Details

Conditional inference trees estimate a regression relationship by binary recursive partitioning in a conditional inference framework. Roughly, the algorithm works as follows: 1) Test the global null hypothesis of independence between any of the input variables and the response (which may be multivariate as well). Stop if this hypothesis cannot be rejected. Otherwise select the input variable with strongest association to the resonse. This association is measured by a p-value corresponding to a test for the partial null hypothesis of a single input variable and the response. 2) Implement a binary split in the selected input variable. 3) Recursively repeate steps 1) and 2).

The implementation utilizes a unified framework for conditional inference, or permutation tests, developed by Strasser and Weber (1999). The stop criterion in step 1) is either based on multiplicity adjusted p-values (testtype == "Bonferroni" or testtype == "MonteCarlo" in ctree_control), on the univariate p-values (testtype == "Univariate"), or on values of the test statistic (testtype == "Teststatistic"). In both cases, the criterion is maximized, i.e., 1 - p-value is used. A split is implemented when the criterion exceeds the value given by mincriterion as specified in ctree_control. For example, when mincriterion = 0.95, the p-value must be smaller than 0.05 in order to split this node. This statistical approach ensures that the right sized tree is grown and no form of pruning or cross-validation or whatsoever is needed. The selection of the input variable to split in is based on the univariate p-values avoiding a variable selection bias towards input variables with many possible cutpoints.

Multiplicity-adjusted Monte-Carlo p-values are computed following a "min-p" approach. The univariate p-values based on the limiting distribution (chi-square or normal) are computed for each of the random permutations of the data. This means that one should use a quadratic test statistic when factors are in play (because the evaluation of the corresponding multivariate normal distribution is time-consuming).

By default, the scores for each ordinal factor `x` are `1:length(x)`, this may be changed using `scores = list(x = c(1,5,6))`, for example.

Predictions can be computed using `predict` or `treeresponse`. The first function accepts arguments `type = c("response", "node", "prob")` where `type = "response"` returns predicted means, predicted classes or median predicted survival times, `type = "node"` returns terminal node IDs (identical to `where`) and `type = "prob"` gives more information about the conditional distribution of the response, i.e., class probabilities or predicted Kaplan-Meier curves and is identical to `treeresponse`. For observations with zero weights, predictions are computed from the fitted tree when `newdata = NULL`.

For a general description of the methodology see Hothorn, Hornik and Zeileis (2006) and Hothorn, Hornik, van de Wiel and Zeileis (2006). Introductions for novices can be found in Strobl et al. (2009) and at https://github.com/christophM/overview-ctrees.

Value

An object of class `BinaryTree-class`.

References

Helmut Strasser and Christian Weber (1999). On the asymptotic theory of permutation statistics. *Mathematical Methods of Statistics*, **8**, 220–250.

Torsten Hothorn, Kurt Hornik, Mark A. van de Wiel and Achim Zeileis (2006). A Lego System for Conditional Inference. *The American Statistician*, **60**(3), 257–263.

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. Preprint available from https://www.zeileis.org/papers/Hothorn+Hornik+Zeileis-2006.pdf

Carolin Strobl, James Malley and Gerhard Tutz (2009). An Introduction to Recursive Partitioning: Rationale, Application, and Characteristics of Classification and Regression Trees, Bagging, and Random forests. *Psychological Methods*, **14**(4), 323–348.

Examples

```
set.seed(290875)

### regression
airq <- subset(airquality, !is.na(Ozone))
airct <- ctree(Ozone ~ ., data = airq,
               controls = ctree_control(maxsurrogate = 3))
airct
plot(airct)
mean((airq$Ozone - predict(airct))^2)
### extract terminal node ID, two ways
all.equal(predict(airct, type = "node"), where(airct))

### classification
irisct <- ctree(Species ~ .,data = iris)
irisct
plot(irisct)
```

```
table(predict(irisct), iris$Species)

### estimated class probabilities, a list
tr <- treeresponse(irisct, newdata = iris[1:10,])

### ordinal regression
data("mammoexp", package = "TH.data")
mammoct <- ctree(ME ~ ., data = mammoexp)
plot(mammoct)

### estimated class probabilities
treeresponse(mammoct, newdata = mammoexp[1:10,])

### survival analysis
if (require("TH.data") && require("survival")) {
    data("GBSG2", package = "TH.data")
    GBSG2ct <- ctree(Surv(time, cens) ~ .,data = GBSG2)
    plot(GBSG2ct)
    treeresponse(GBSG2ct, newdata = GBSG2[1:2,])
}

### if you are interested in the internals:
### generate doxygen documentation
## Not run:

    ### download src package into temp dir
    tmpdir <- tempdir()
    tgz <- download.packages("party", destdir = tmpdir)[2]
    ### extract
    untar(tgz, exdir = tmpdir)
    wd <- setwd(file.path(tmpdir, "party"))
    ### run doxygen (assuming it is there)
    system("doxygen inst/doxygen.cfg")
    setwd(wd)
    ### have fun
    browseURL(file.path(tmpdir, "party", "inst",
                        "documentation", "html", "index.html"))

## End(Not run)
```

Control ctree Hyper Parameters

Control for Conditional Inference Trees

Description

Various parameters that control aspects of the 'ctree' fit.

Usage

```
ctree_control(teststat = c("quad", "max"),
              testtype = c("Bonferroni", "MonteCarlo",
                           "Univariate", "Teststatistic"),
              mincriterion = 0.95, minsplit = 20, minbucket = 7,
              stump = FALSE, nresample = 9999, maxsurrogate = 0,
            mtry = 0, savesplitstats = TRUE, maxdepth = 0, remove_weights = FALSE)
```

Arguments

`teststat`	a character specifying the type of the test statistic to be applied.
`testtype`	a character specifying how to compute the distribution of the test statistic.
`mincriterion`	the value of the test statistic (for `testtype == "Teststatistic"`), or 1 - p-value (for other values of `testtype`) that must be exceeded in order to implement a split.
`minsplit`	the minimum sum of weights in a node in order to be considered for splitting.
`minbucket`	the minimum sum of weights in a terminal node.
`stump`	a logical determining whether a stump (a tree with three nodes only) is to be computed.
`nresample`	number of Monte-Carlo replications to use when the distribution of the test statistic is simulated.
`maxsurrogate`	number of surrogate splits to evaluate. Note that currently only surrogate splits in ordered covariables are implemented.
`mtry`	number of input variables randomly sampled as candidates at each node for random forest like algorithms. The default `mtry = 0` means that no random selection takes place.
`savesplitstats`	a logical determining if the process of standardized two-sample statistics for split point estimate is saved for each primary split.
`maxdepth`	maximum depth of the tree. The default `maxdepth = 0` means that no restrictions are applied to tree sizes.
`remove_weights`	a logical determining if weights attached to nodes shall be removed after fitting the tree.

Details

The arguments `teststat`, `testtype` and `mincriterion` determine how the global null hypothesis of independence between all input variables and the response is tested (see `ctree`). The argument `nresample` is the number of Monte-Carlo replications to be used when `testtype = "MonteCarlo"`.

A split is established when the sum of the weights in both daugther nodes is larger than `minsplit`, this avoids pathological splits at the borders. When `stump = TRUE`, a tree with at most two terminal nodes is computed.

The argument `mtry > 0` means that a random forest like 'variable selection', i.e., a random selection of `mtry` input variables, is performed in each node.

It might be informative to look at scatterplots of input variables against the standardized two-sample split statistics, those are available when `savesplitstats = TRUE`. Each node is then associated with a vector whose length is determined by the number of observations in the learning sample and thus much more memory is required.

Value

An object of class `TreeControl`.

`Control Forest Hyper Parameters`

Control for Conditional Tree Forests

Description

Various parameters that control aspects of the 'cforest' fit via its 'control' argument.

Usage

```
cforest_unbiased(...)
cforest_classical(...)
cforest_control(teststat = "max",
                testtype = "Teststatistic",
                mincriterion = qnorm(0.9),
                savesplitstats = FALSE,
                ntree = 500, mtry = 5, replace = TRUE,
                fraction = 0.632, trace = FALSE, ...)
```

Arguments

`teststat`	a character specifying the type of the test statistic to be applied.
`testtype`	a character specifying how to compute the distribution of the test statistic.
`mincriterion`	the value of the test statistic (for `testtype == "Teststatistic"`), or 1 - p-value (for other values of `testtype`) that must be exceeded in order to implement a split.
`mtry`	number of input variables randomly sampled as candidates at each node for random forest like algorithms. Bagging, as special case of a random forest without random input variable sampling, can be performed by setting `mtry` either equal to `NULL` or manually equal to the number of input variables.
`savesplitstats`	a logical determining whether the process of standardized two-sample statistics for split point estimate is saved for each primary split.
`ntree`	number of trees to grow in a forest.
`replace`	a logical indicating whether sampling of observations is done with or without replacement.

`fraction`	fraction of number of observations to draw without replacement (only relevant if `replace = FALSE`).
`trace`	a logical indicating if a progress bar shall be printed while the forest grows.
`...`	additional arguments to be passed to `ctree_control`.

Details

All three functions return an object of class `ForestControl-class` defining hyper parameters to be specified via the `control` argument of `cforest`.

The arguments `teststat`, `testtype` and `mincriterion` determine how the global null hypothesis of independence between all input variables and the response is tested (see `ctree`). The argument `nresample` is the number of Monte-Carlo replications to be used when `testtype = "MonteCarlo"`.

A split is established when the sum of the weights in both daugther nodes is larger than `minsplit`, this avoids pathological splits at the borders. When `stump = TRUE`, a tree with at most two terminal nodes is computed.

The `mtry` argument regulates a random selection of `mtry` input variables in each node. Note that here `mtry` is fixed to the value 5 by default for merely technical reasons, while in `randomForest` the default values for classification and regression vary with the number of input variables. Make sure that `mtry` is defined properly before using `cforest`.

It might be informative to look at scatterplots of input variables against the standardized two-sample split statistics, those are available when `savesplitstats = TRUE`. Each node is then associated with a vector whose length is determined by the number of observations in the learning sample and thus much more memory is required.

The number of trees `ntree` can be increased for large numbers of input variables.

Function `cforest_unbiased` returns the settings suggested for the construction of unbiased random forests (`teststat = "quad", testtype = "Univ", replace = FALSE`) by Strobl et al. (2007) and is the default since version 0.9-90. Hyper parameter settings mimicing the behaviour of `randomForest` are available in `cforest_classical` which have been used as default up to version 0.9-14.

Please note that `cforest`, in contrast to `randomForest`, doesn't grow trees of maximal depth. To grow large trees, set `mincriterion = 0`.

Value

An object of class `ForestControl-class`.

References

Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis and Torsten Hothorn (2007). Bias in Random Forest Variable Importance Measures: Illustrations, Sources and a Solution. *BMC Bioinformatics*, **8**, 25. DOI: 10.1186/1471-2105-8-25

Fit Methods *Fit 'StatModel' Objects to Data*

Description

Fit a 'StatModel' model to objects of class 'LearningSample'.

Methods

fit `signature(model = "StatModel", data = "LearningSample")`: fit `model` to `data`.

ForestControl-class *Class "ForestControl"*

Description

Objects of this class represent the hyper parameter setting for forest growing.

Objects from the Class

Objects can be created by `cforest_control`.

Slots

`ntree`: number of trees in the forest.

`replace`: sampling with or without replacement.

`fraction`: fraction of observations to sample without replacement.

`trace`: logical indicating if a progress bar shall be printed.

`varctrl`: Object of class `"VariableControl"`

`splitctrl`: Object of class `"SplitControl"`

`gtctrl`: Object of class `"GlobalTestControl"`

`tgctrl`: Object of class `"TreeGrowControl"`

Extends

Class `"TreeControl"`, directly.

Methods

No methods defined with class "ForestControl" in the signature.

`Initialize Methods` *Methods for Function initialize in Package 'party'*

Description

Methods for function `initialize` in package **party** – those are internal functions not to be called by users.

Methods

.Object = "ExpectCovarInfluence" `new("ExpectCovarInfluence")`

.Object = "ExpectCovar" `new("ExpectCovar")`

.Object = "LinStatExpectCovar" `new("LinStatExpectCovar")`

.Object = "LinStatExpectCovarMPinv" `new("LinStatExpectCovarMPinv")`

.Object = "VariableFrame" `new("VariableFrame")`

`initVariableFrame-methods`
Set-up VariableFrame objects

Description

Set-up VariableFrame objects

Methods

These methods are not to be called by the user.

`signature(obj = "data.frame")` converges a data frame to VariableFrame

`signature(obj = "matrix")` converges a matrix to VariableFrame

LearningSample Class *Class "LearningSample"*

Description

Objects of this class represent data for fitting tree-based models.

Objects from the Class

Objects can be created by calls of the form `new("LearningSample", ...)`.

Slots

`responses`: Object of class `"VariableFrame"` with the response variables.

`inputs`: Object of class `"VariableFrame"` with the input variables.

`weights`: Object of class `"numeric"`, a vector of case counts or weights.

`nobs`: Object of class `"integer"`, the number of observations.

`ninputs`: Object of class `"integer"`, the number of input variables.

Methods

No methods defined with class "LearningSample" in the signature.

mob *Model-based Recursive Partitioning*

Description

MOB is an algorithm for model-based recursive partitioning yielding a tree with fitted models associated with each terminal node.

Usage

```
mob(formula, weights, data = list(), na.action = na.omit, model = glinearModel,
  control = mob_control(), ...)

## S3 method for class 'mob'
predict(object, newdata = NULL, type = c("response", "node"), ...)
## S3 method for class 'mob'
summary(object, node = NULL, ...)
## S3 method for class 'mob'
coef(object, node = NULL, ...)
## S3 method for class 'mob'
sctest(x, node = NULL, ...)
```

Arguments

formula	A symbolic description of the model to be fit. This should be of type y ~ x1 + ... + xk | z1 + ... + zl where the variables before the | are passed to the model and the variables after the | are used for partitioning.
weights	An optional vector of weights to be used in the fitting process. Only non-negative integer valued weights are allowed (default = 1).
data	A data frame containing the variables in the model.
na.action	A function which indicates what should happen when the data contain NAs, defaulting to na.omit.
model	A model of class "StatModel". See details for requirements.
control	A list with control parameters as returned by mob_control.
...	Additional arguments passed to the fit call for the model.
object, x	A fitted mob object.
newdata	A data frame with new inputs, by default the learning data is used.
type	A character string specifying whether the response should be predicted (inherited from the predict method for the model) or the ID of the associated terminal node.
node	A vector of node IDs for which the corresponding method should be applied.

Details

Model-based partitioning fits a model tree using the following algorithm:

1. fit a model (default: a generalized linear model "StatModel" with formula y ~ x1 + ... + xk for the observations in the current node.
2. Assess the stability of the model parameters with respect to each of the partitioning variables z1, ..., zl. If there is some overall instability, choose the variable z associated with the smallest p value for partitioning, otherwise stop. For performing the parameter instability fluctuation test, a estfun method and a weights method is needed.
3. Search for the locally optimal split in z by minimizing the objective function of the model. Typically, this will be something like deviance or the negative logLik and can be specified in mob_control.
4. Re-fit the model in both children, using reweight and repeat from step 2.

More details on the conceptual design of the algorithm can be found in Zeileis, Hothorn, Hornik (2008) and some illustrations are provided in vignette("MOB").

For the fitted MOB tree, several standard methods are inherited if they are available for fitted models, such as print, predict, residuals, logLik, deviance, weights, coef and summary. By default, the latter four return the result (deviance, weights, coefficients, summary) for all terminal nodes, but take a node argument that can be set to any node ID. The sctest method extracts the results of the parameter stability tests (aka structural change tests) for any given node, by default for all nodes. Some examples are given below.

Value

An object of class mob inheriting from BinaryTree-class. Every node of the tree is additionally associated with a fitted model.

References

Achim Zeileis, Torsten Hothorn, and Kurt Hornik (2008). Model-Based Recursive Partitioning. *Journal of Computational and Graphical Statistics*, **17**(2), 492–514.

See Also

plot.mob, mob_control

Examples

```
set.seed(290875)

if(require("mlbench")) {

## recursive partitioning of a linear regression model
## load data
data("BostonHousing", package = "mlbench")
## and transform variables appropriately (for a linear regression)
BostonHousing$lstat <- log(BostonHousing$lstat)
BostonHousing$rm <- BostonHousing$rm^2
## as well as partitioning variables (for fluctuation testing)
BostonHousing$chas <- factor(BostonHousing$chas, levels = 0:1,
                             labels = c("no", "yes"))
BostonHousing$rad <- factor(BostonHousing$rad, ordered = TRUE)

## partition the linear regression model medv ~ lstat + rm
## with respect to all remaining variables:
fmBH <- mob(medv ~ lstat + rm | zn + indus + chas + nox + age +
                                dis + rad + tax + crim + b + ptratio,
  control = mob_control(minsplit = 40), data = BostonHousing,
  model = linearModel)

## print the resulting tree
fmBH
## or better visualize it
plot(fmBH)

## extract coefficients in all terminal nodes
coef(fmBH)
## look at full summary, e.g., for node 7
summary(fmBH, node = 7)
## results of parameter stability tests for that node
sctest(fmBH, node = 7)
## -> no further significant instabilities (at 5% level)

## compute mean squared error (on training data)
mean((BostonHousing$medv - fitted(fmBH))^2)
```

```
mean(residuals(fmBH)^2)
deviance(fmBH)/sum(weights(fmBH))

## evaluate logLik and AIC
logLik(fmBH)
AIC(fmBH)
## (Note that this penalizes estimation of error variances, which
## were treated as nuisance parameters in the fitting process.)


## recursive partitioning of a logistic regression model
## load data
data("PimaIndiansDiabetes", package = "mlbench")
## partition logistic regression diabetes ~ glucose
## wth respect to all remaining variables
fmPID <- mob(diabetes ~ glucose | pregnant + pressure + triceps +
                                  insulin + mass + pedigree + age,
  data = PimaIndiansDiabetes, model = glinearModel,
  family = binomial())

## fitted model
coef(fmPID)
plot(fmPID)
plot(fmPID, tp_args = list(cdplot = TRUE))
}
```

---

mob_control *Control Parameters for Model-based Partitioning*

Description

Various parameters that control aspects the fitting algorithm for recursively partitioned mob models.

Usage

```
mob_control(alpha = 0.05, bonferroni = TRUE, minsplit = 20, trim = 0.1,
  objfun = deviance, breakties = FALSE, parm = NULL, verbose = FALSE)
```

Arguments

alpha	numeric significance level. A node is splitted when the (possibly Bonferroni-corrected) p value for any parameter stability test in that node falls below alpha.
bonferroni	logical. Should p values be Bonferroni corrected?
minsplit	integer. The minimum number of observations (sum of the weights) in a node.
trim	numeric. This specifies the trimming in the parameter instability test for the numerical variables. If smaller than 1, it is interpreted as the fraction relative to the current node size.

objfun	function. A function for extracting the minimized value of the objective function from a fitted model in a node.
breakties	logical. Should ties in numeric variables be broken randomly for computing the associated parameter instability test?
parm	numeric or character. Number or name of model parameters included in the parameter instability tests (by default all parameters are included).
verbose	logical. Should information about the fitting process of mob (such as test statistics, p values, selected splitting variables and split points) be printed to the screen?

Details

See mob for more details and references.

Value

A list of class mob_control containing the control parameters.

See Also

mob

Panel Generating Functions

Panel-Generators for Visualization of Party Trees

Description

The plot method for BinaryTree and mob objects are rather flexible and can be extended by panel functions. Some pre-defined panel-generating functions of class grapcon_generator for the most important cases are documented here.

Usage

```
node_inner(ctreeobj, digits = 3, abbreviate = FALSE,
  fill = "white", pval = TRUE, id = TRUE)
node_terminal(ctreeobj, digits = 3, abbreviate = FALSE,
  fill = c("lightgray", "white"), id = TRUE)
edge_simple(treeobj, digits = 3, abbreviate = FALSE)
node_surv(ctreeobj, ylines = 2, id = TRUE, ...)
node_barplot(ctreeobj, col = "black", fill = NULL, beside = NULL,
  ymax = NULL, ylines = NULL, widths = 1, gap = NULL,
  reverse = NULL, id = TRUE)
node_boxplot(ctreeobj, col = "black", fill = "lightgray",
  width = 0.5,  yscale = NULL, ylines = 3, cex = 0.5, id = TRUE)
node_hist(ctreeobj, col = "black", fill = "lightgray",
  freq = FALSE, horizontal = TRUE, xscale = NULL, ymax = NULL,
```

```
  ylines = 3, id = TRUE, ...)
node_density(ctreeobj, col = "black", rug = TRUE,
  horizontal = TRUE, xscale = NULL, yscale = NULL, ylines = 3,
  id = TRUE)
node_scatterplot(mobobj, which = NULL, col = "black",
  linecol = "red", cex = 0.5, pch = NULL, jitter = FALSE,
  xscale = NULL, yscale = NULL, ylines = 1.5, id = TRUE,
  labels = FALSE)
node_bivplot(mobobj, which = NULL, id = TRUE, pop = TRUE,
  pointcol = "black", pointcex = 0.5,
  boxcol = "black", boxwidth = 0.5, boxfill = "lightgray",
  fitmean = TRUE, linecol = "red",
  cdplot = FALSE, fivenum = TRUE, breaks = NULL,
  ylines = NULL, xlab = FALSE, ylab = FALSE, margins = rep(1.5, 4), ...)
```

Arguments

ctreeobj	an object of class BinaryTree.
treeobj	an object of class BinaryTree or mob.
mobobj	an object of class mob.
digits	integer, used for formating numbers.
abbreviate	logical indicating whether strings should be abbreviated.
col, pointcol	a color for points and lines.
fill	a color to filling rectangles.
pval	logical. Should p values be plotted?
id	logical. Should node IDs be plotted?
ylines	number of lines for spaces in y-direction.
widths	widths in barplots.
width, boxwidth	width in boxplots.
gap	gap between bars in a barplot (node_barplot).
yscale	limits in y-direction
xscale	limits in x-direction
ymax	upper limit in y-direction
beside	logical indicating if barplots should be side by side or stacked.
reverse	logical indicating whether the order of levels should be reversed for barplots.
horizontal	logical indicating if the plots should be horizontal.
freq	logical; if TRUE, the histogram graphic is a representation of frequencies. If FALSE, probabilities are plotted.
rug	logical indicating if a rug representation should be added.
which	numeric or character vector indicating which of the regressor variables should be plotted (default = all).
linecol	color for fitted model lines.

cex, pointcex	character extension of points in scatter plots.
pch	plotting character of points in scatter plots.
jitter	logical. Should the points be jittered in y-direction?
labels	logical. Should axis labels be plotted?
pop	logical. Should the panel viewports be popped?
boxcol	color for box plot borders.
boxfill	fill color for box plots.
fitmean	logical. Should lines for the predicted means from the model be added?
cdplot	logical. Should CD plots (or spinograms) be used for visualizing the dependence of a categorical on a numeric variable?
fivenum	logical. When using spinograms, should the five point summary of the explanatory variable be used for determining the breaks?
breaks	a (list of) numeric vector(s) of breaks for the spinograms. If set to NULL (the default), the breaks are chosen according to the fivenum argument.
xlab, ylab	character with x- and y-axis labels. Can also be logical: if FALSE axis labels are surpressed, if TRUE they are taken from the underlying data. Can be a vector of labels for xlab.
margins	margins of the viewports.
...	additional arguments passed to callies.

Details

The plot methods for BinaryTree and mob objects provide an extensible framework for the visualization of binary regression trees. The user is allowed to specify panel functions for plotting terminal and inner nodes as well as the corresponding edges. The panel functions to be used should depend only on the node being visualzied, however, for setting up an appropriate panel function, information from the whole tree is typically required. Hence, **party** adopts the framework of grapcon_generator (graphical appearance control) from the **vcd** package (Meyer, Zeileis and Hornik, 2005) and provides several panel-generating functions. For convenience, the panel-generating functions node_inner and edge_simple return panel functions to draw inner nodes and left and right edges. For drawing terminal nodes, the functions returned by the other panel functions can be used. The panel generating function node_terminal is a terse text-based representation of terminal nodes.

Graphical representations of terminal nodes are available and depend on the kind of model and the measurement scale of the variables modelled.

For univariate regressions (typically fitted by ctree), node_surv returns a functions that plots Kaplan-Meier curves in each terminal node; node_barplot, node_boxplot, node_hist and node_density can be used to plot bar plots, box plots, histograms and estimated densities into the terminal nodes.

For multivariate regressions (typically fitted by mob), node_bivplot returns a panel function that creates bivariate plots of the response against all regressors in the model. Depending on the scale of the variables involved, scatter plots, box plots, spinograms (or CD plots) and spine plots are created. For the latter two spine and cd_plot from the **vcd** package are re-used.

References

David Meyer, Achim Zeileis, and Kurt Hornik (2006). The Strucplot Framework: Visualizing Multi-Way Contingency Tables with vcd. *Journal of Statistical Software*, **17**(3). doi:10.18637/jss.v017.i03

Examples

```
set.seed(290875)

airq <- subset(airquality, !is.na(Ozone))
airct <- ctree(Ozone ~ ., data = airq)

## default: boxplots
plot(airct)

## change colors
plot(airct, tp_args = list(col = "blue", fill = hsv(2/3, 0.5, 1)))
## equivalent to
plot(airct, terminal_panel = node_boxplot(airct, col = "blue",
                                          fill = hsv(2/3, 0.5, 1)))

### very simple; the mean is given in each terminal node
plot(airct, type = "simple")

### density estimates
plot(airct, terminal_panel = node_density)

### histograms
plot(airct, terminal_panel = node_hist(airct, ymax = 0.06,
                                       xscale = c(0, 250)))
```

`Plot BinaryTree` *Visualization of Binary Regression Trees*

Description

`plot` method for `BinaryTree` objects with extended facilities for plugging in panel functions.

Usage

```
## S3 method for class 'BinaryTree'
plot(x, main = NULL, type = c("extended", "simple"),
     terminal_panel = NULL, tp_args = list(),
     inner_panel = node_inner, ip_args = list(),
     edge_panel = edge_simple, ep_args = list(),
     drop_terminal = (type[1] == "extended"),
     tnex = (type[1] == "extended") + 1, newpage = TRUE,
     pop = TRUE, ...)
```

Arguments

x	an object of class BinaryTree.
main	an optional title for the plot.
type	a character specifying the complexity of the plot: extended tries to visualize the distribution of the response variable in each terminal node whereas simple only gives some summary information.
terminal_panel	an optional panel function of the form function(node) plotting the terminal nodes. Alternatively, a panel generating function of class "grapcon_generator" that is called with arguments x and tp_args to set up a panel function. By default, an appropriate panel function is chosen depending on the scale of the dependent variable.
tp_args	a list of arguments passed to terminal_panel if this is a "grapcon_generator" object.
inner_panel	an optional panel function of the form function(node) plotting the inner nodes. Alternatively, a panel generating function of class "grapcon_generator" that is called with arguments x and ip_args to set up a panel function.
ip_args	a list of arguments passed to inner_panel if this is a "grapcon_generator" object.
edge_panel	an optional panel function of the form function(split, ordered = FALSE, left = TRUE) plotting the edges. Alternatively, a panel generating function of class "grapcon_generator" that is called with arguments x and ip_args to set up a panel function.
ep_args	a list of arguments passed to edge_panel if this is a "grapcon_generator" object.
drop_terminal	a logical indicating whether all terminal nodes should be plotted at the bottom.
tnex	a numeric value giving the terminal node extension in relation to the inner nodes.
newpage	a logical indicating whether grid.newpage() should be called.
pop	a logical whether the viewport tree should be popped before return.
...	additional arguments passed to callies.

Details

This plot method for BinaryTree objects provides an extensible framework for the visualization of binary regression trees. The user is allowed to specify panel functions for plotting terminal and inner nodes as well as the corresponding edges. Panel functions for plotting inner nodes, edges and terminal nodes are available for the most important cases and can serve as the basis for user-supplied extensions, see node_inner and vignette("party").

More details on the ideas and concepts of panel-generating functions and "grapcon_generator" objects in general can be found in Meyer, Zeileis and Hornik (2005).

References

David Meyer, Achim Zeileis, and Kurt Hornik (2006). The Strucplot Framework: Visualizing Multi-Way Contingency Tables with vcd. *Journal of Statistical Software*, **17**(3). doi:10.18637/jss.v017.i03

See Also

node_inner, node_terminal, edge_simple, node_surv, node_barplot, node_boxplot, node_hist, node_density

Examples

```
set.seed(290875)

airq <- subset(airquality, !is.na(Ozone))
airct <- ctree(Ozone ~ ., data = airq)

### regression: boxplots in each node
plot(airct, terminal_panel = node_boxplot, drop_terminal = TRUE)

if(require("TH.data")) {
## classification: barplots in each node
data("GlaucomaM", package = "TH.data")
glauct <- ctree(Class ~ ., data = GlaucomaM)
plot(glauct)
plot(glauct, inner_panel = node_barplot,
  edge_panel = function(ctreeobj, ...) { function(...) invisible() },
  tnex = 1)

## survival: Kaplan-Meier curves in each node
data("GBSG2", package = "TH.data")
library("survival")
gbsg2ct <- ctree(Surv(time, cens) ~ ., data = GBSG2)
plot(gbsg2ct)
plot(gbsg2ct, type = "simple")
}
```

---

plot.mob *Visualization of MOB Trees*

Description

plot method for mob objects with extended facilities for plugging in panel functions.

Usage

```
## S3 method for class 'mob'
plot(x, terminal_panel = node_bivplot, tnex = NULL, ...)
```

Arguments

x	an object of class mob.
terminal_panel	a panel function or panel-generating function of class "grapcon_generator". See plot.BinaryTree for more details.

tnex	a numeric value giving the terminal node extension in relation to the inner nodes.
...	further arguments passed to plot.BinaryTree.

Details

This plot method for mob objects simply calls the plot.BinaryTree method, setting a different terminal_panel function by default (node_bivplot) and tnex value.

See Also

node_bivplot, node_scatterplot, plot.BinaryTree, mob

Examples

```
set.seed(290875)

if(require("mlbench")) {

## recursive partitioning of a linear regression model
## load data
data("BostonHousing", package = "mlbench")
## and transform variables appropriately (for a linear regression)
BostonHousing$lstat <- log(BostonHousing$lstat)
BostonHousing$rm <- BostonHousing$rm^2
## as well as partitioning variables (for fluctuation testing)
BostonHousing$chas <- factor(BostonHousing$chas, levels = 0:1,
                             labels = c("no", "yes"))
BostonHousing$rad <- factor(BostonHousing$rad, ordered = TRUE)

## partition the linear regression model medv ~ lstat + rm
## with respect to all remaining variables:
fm <- mob(medv ~ lstat + rm | zn + indus + chas + nox + age + dis +
                              rad + tax + crim + b + ptratio,
  control = mob_control(minsplit = 40), data = BostonHousing,
  model = linearModel)

## visualize medv ~ lstat and medv ~ rm
plot(fm)

## visualize only one of the two regressors
plot(fm, tp_args = list(which = "lstat"), tnex = 2)
plot(fm, tp_args = list(which = 2), tnex = 2)

## omit fitted mean lines
plot(fm, tp_args = list(fitmean = FALSE))

## mixed numerical and categorical regressors
fm2 <- mob(medv ~ lstat + rm + chas | zn + indus + nox + age +
                                      dis + rad,
  control = mob_control(minsplit = 100), data = BostonHousing,
  model = linearModel)
plot(fm2)
```

```
## recursive partitioning of a logistic regression model
data("PimaIndiansDiabetes", package = "mlbench")
fmPID <- mob(diabetes ~ glucose | pregnant + pressure + triceps +
                                  insulin + mass + pedigree + age,
  data = PimaIndiansDiabetes, model = glinearModel,
  family = binomial())
## default plot: spinograms with breaks from five point summary
plot(fmPID)
## use the breaks from hist() instead
plot(fmPID, tp_args = list(fivenum = FALSE))
## user-defined breaks
plot(fmPID, tp_args = list(breaks = 0:4 * 50))
## CD plots instead of spinograms
plot(fmPID, tp_args = list(cdplot = TRUE))
## different smoothing bandwidth
plot(fmPID, tp_args = list(cdplot = TRUE, bw = 15))

}
```

`prettytree` *Print a tree.*

Description

Produces textual output representing a tree.

Usage

```
prettytree(x, inames = NULL, ilevels = NULL)
```

Arguments

`x`	a recursive list representing a tree.
`inames`	optional variable names.
`ilevels`	an optional list of levels for factors.

Details

This function is normally not called by users but needed in some reverse dependencies of party.

RandomForest-class *Class "RandomForest"*

Description

A class for representing random forest ensembles.

Objects from the Class

Objects can be created by calls of the form `new("RandomForest", ...)`.

Slots

`ensemble`: Object of class `"list"`, each element being an object of class `"BinaryTree"`.

`data`: an object of class `"ModelEnv"`.

`initweights`: a vector of initial weights.

`weights`: a list of weights defining the sub-samples.

`where`: a matrix of integers vectors of length n (number of observations in the learning sample) giving the number of the terminal node the corresponding observations is element of (in each tree).

`data`: an object of class `"ModelEnv"`.

`responses`: an object of class `"VariableFrame"` storing the values of the response variable(s).

`cond_distr_response`: a function computing the conditional distribution of the response.

`predict_response`: a function for computing predictions.

`prediction_weights`: a function for extracting weights from terminal nodes.

`get_where`: a function for determining the number of terminal nodes observations fall into.

`update`: a function for updating weights.

Methods

treeresponse `signature(object = "RandomForest")`: ...

weights `signature(object = "RandomForest")`: ...

where `signature(object = "RandomForest")`: ...

Examples

```
set.seed(290875)

### honest (i.e., out-of-bag) cross-classification of
### true vs. predicted classes
data("mammoexp", package = "TH.data")
table(mammoexp$ME, predict(cforest(ME ~ ., data = mammoexp,
                           control = cforest_unbiased(ntree = 50)),
                           OOB = TRUE))
```

readingSkills *Reading Skills*

Description

A toy data set illustrating the spurious correlation between reading skills and shoe size in schoolchildren.

Usage

```
data("readingSkills")
```

Format

A data frame with 200 observations on the following 4 variables.

`nativeSpeaker` a factor with levels `no` and `yes`, where `yes` indicates that the child is a native speaker of the language of the reading test.

`age` age of the child in years.

`shoeSize` shoe size of the child in cm.

`score` raw score on the reading test.

Details

In this artificial data set, that was generated by means of a linear model, `age` and `nativeSpeaker` are actual predictors of the `score`, while the spurious correlation between `score` and `shoeSize` is merely caused by the fact that both depend on `age`.

The true predictors can be identified, e.g., by means of partial correlations, standardized beta coefficients in linear models or the conditional random forest variable importance, but not by means of the standard random forest variable importance (see example).

Examples

```
set.seed(290875)
readingSkills.cf <- cforest(score ~ ., data = readingSkills,
    control = cforest_unbiased(mtry = 2, ntree = 50))

# standard importance
varimp(readingSkills.cf)
# the same modulo random variation
varimp(readingSkills.cf, pre1.0_0 = TRUE)

# conditional importance, may take a while...
varimp(readingSkills.cf, conditional = TRUE)
```

reweight *Re-fitting Models with New Weights*

Description

Generic function for re-fitting a model object using the same observations but different weights.

Usage

```
reweight(object, weights, ...)
```

Arguments

object	a fitted model object.
weights	a vector of weights.
...	arguments passed to methods.

Details

The method is not unsimilar in spirit to update, but much more narrowly focused. It should return an updated fitted model derived from re-fitting the model on the same observations but using different weights.

Value

The re-weighted fitted model object.

See Also

update

Examples

```
## fit cars regression
mf <- dpp(linearModel, dist ~ speed, data = cars)
fm <- fit(linearModel, mf)
fm

## re-fit, excluding the last 4 observations
ww <- c(rep(1, 46), rep(0, 4))
reweight(fm, ww)
```

SplittingNode Class *Class "SplittingNode"*

Description

A list representing the inner node of a binary tree.

Extends

Class "list", from data part. Class "vector", by class "list". See BinaryTree-class for more details.

Transformations *Function for Data Transformations*

Description

Transformations of Response or Input Variables

Usage

```
ptrafo(data, numeric_trafo = id_trafo, factor_trafo = ff_trafo,
    ordered_trafo = of_trafo, surv_trafo = logrank_trafo,
    var_trafo = NULL)
ff_trafo(x)
```

Arguments

data	an object of class data.frame.
numeric_trafo	a function to by applied to numeric elements of data returning a matrix with nrow(data) rows and an arbitrary number of columns.
ordered_trafo	a function to by applied to ordered elements of data returning a matrix with nrow(data) rows and an arbitrary number of columns (usually some scores).
factor_trafo	a function to by applied to factor elements of data returning a matrix with nrow(data) rows and an arbitrary number of columns (usually a dummy or contrast matrix).
surv_trafo	a function to by applied to elements of class Surv of data returning a matrix with nrow(data) rows and an arbitrary number of columns.
var_trafo	an optional named list of functions to be applied to the corresponding variables in data.
x	a factor

Details

trafo applies its arguments to the elements of data according to the classes of the elements. See Transformations for more documentation and examples.

In the presence of missing values, one needs to make sure that all user-supplied functions deal with that.

Value

A named matrix with nrow(data) rows and arbitrary number of columns.

Examples

```
### rank a variable
ptrafo(data.frame(y = 1:20),
       numeric_trafo = function(x) rank(x, na.last = "keep"))

### dummy coding of a factor
ptrafo(data.frame(y = gl(3, 9)))
```

TreeControl Class *Class "TreeControl"*

Description

Objects of this class represent the hyper parameter setting for tree growing.

Objects from the Class

Objects can be created by ctree_control.

Slots

varctrl: Object of class "VariableControl".

splitctrl: Object of class "SplitControl".

gtctrl: Object of class "GlobalTestControl".

tgctrl: Object of class "TreeGrowControl".

Methods

No methods defined with class "TreeControl" in the signature.

varimp *Variable Importance*

Description

Standard and conditional variable importance for 'cforest', following the permutation principle of the 'mean decrease in accuracy' importance in 'randomForest'.

Usage

```
varimp(object, mincriterion = 0, conditional = FALSE,
       threshold = 0.2, nperm = 1, OOB = TRUE, pre1.0_0 = conditional)
varimpAUC(...)
```

Arguments

object	an object as returned by cforest.
mincriterion	the value of the test statistic or 1 - p-value that must be exceeded in order to include a split in the computation of the importance. The default mincriterion = 0 guarantees that all splits are included.
conditional	a logical determining whether unconditional or conditional computation of the importance is performed.
threshold	the threshold value for (1 - p-value) of the association between the variable of interest and a covariate, which must be exceeded inorder to include the covariate in the conditioning scheme for the variable of interest (only relevant if conditional = TRUE). A threshold value of zero includes all covariates.
nperm	the number of permutations performed.
OOB	a logical determining whether the importance is computed from the out-of-bag sample or the learning sample (not suggested).
pre1.0_0	Prior to party version 1.0-0, the actual data values were permuted according to the original permutation importance suggested by Breiman (2001). Now the assignments to child nodes of splits in the variable of interest are permuted as described by Hapfelmeier et al. (2012), which allows for missing values in the explanatory variables and is more efficient wrt memory consumption and computing time. This method does not apply to conditional variable importances.
...	Arguments to varImpAUC.

Details

Function varimp can be used to compute variable importance measures similar to those computed by importance. Besides the standard version, a conditional version is available, that adjusts for correlations between predictor variables.

If conditional = TRUE, the importance of each variable is computed by permuting within a grid defined by the covariates that are associated (with 1 - p-value greater than threshold) to the variable of interest. The resulting variable importance score is conditional in the sense of beta coefficients in

regression models, but represents the effect of a variable in both main effects and interactions. See Strobl et al. (2008) for details.

Note, however, that all random forest results are subject to random variation. Thus, before interpreting the importance ranking, check whether the same ranking is achieved with a different random seed – or otherwise increase the number of trees `ntree` in `ctree_control`.

Note that in the presence of missings in the predictor variables the procedure described in Hapfelmeier et al. (2012) is performed.

Function `varimpAUC` is a wrapper for `varImpAUC` which implements AUC-based variables importances as described by Janitza et al. (2012). Here, the area under the curve instead of the accuracy is used to calculate the importance of each variable. This AUC-based variable importance measure is more robust towards class imbalance.

For right-censored responses, `varimp` uses the integrated Brier score as a risk measure for computing variable importances. This feature is extremely slow and experimental; use at your own risk.

Value

A vector of 'mean decrease in accuracy' importance scores.

References

Leo Breiman (2001). Random Forests. *Machine Learning*, 45(1), 5–32.

Alexander Hapfelmeier, Torsten Hothorn, Kurt Ulm, and Carolin Strobl (2012). A New Variable Importance Measure for Random Forests with Missing Data. *Statistics and Computing*, doi:10.1007/s1122201293491

Torsten Hothorn, Kurt Hornik, and Achim Zeileis (2006b). Unbiased Recursive Partitioning: A Conditional Inference Framework. *Journal of Computational and Graphical Statistics*, **15** (3), 651-674. Preprint available from https://www.zeileis.org/papers/Hothorn+Hornik+Zeileis-2006.pdf

Silke Janitza, Carolin Strobl and Anne-Laure Boulesteix (2013). An AUC-based Permutation Variable Importance Measure for Random Forests. BMC Bioinformatics.2013, **14** 119. doi:10.1186/1471210514119

Carolin Strobl, Anne-Laure Boulesteix, Thomas Kneib, Thomas Augustin, and Achim Zeileis (2008). Conditional Variable Importance for Random Forests. *BMC Bioinformatics*, **9**, 307. doi:10.1186/147121059307

Examples

```
set.seed(290875)
readingSkills.cf <- cforest(score ~ ., data = readingSkills,
    control = cforest_unbiased(mtry = 2, ntree = 50))

# standard importance
varimp(readingSkills.cf)
# the same modulo random variation
varimp(readingSkills.cf, pre1.0_0 = TRUE)
```

```
    # conditional importance, may take a while...
    varimp(readingSkills.cf, conditional = TRUE)

    ## Not run:
    data("GBSG2", package = "TH.data")
    ### add a random covariate for sanity check
    set.seed(29)
    GBSG2$rand <- runif(nrow(GBSG2))
    object <- cforest(Surv(time, cens) ~ ., data = GBSG2,
                      control = cforest_unbiased(ntree = 20))
    vi <- varimp(object)
    ### compare variable importances and absolute z-statistics
    layout(matrix(1:2))
    barplot(vi)
    barplot(abs(summary(coxph(Surv(time, cens) ~ ., data = GBSG2))$coeff[,"z"]))
    ### looks more or less the same

## End(Not run)
```

Index