Package ‘partykit’

October 14, 2022

Title  A Toolkit for Recursive Partytioning

Date  2022-06-20

Version  1.2-16

Description  A toolkit with infrastructure for representing, summarizing, and visualizing tree-structured regression and classification models. This unified infrastructure can be used for reading/coercing tree models from different sources (‘rpart’, ‘RWeka’, ‘PMML’) yielding objects that share functionality for print()/plot()/predict() methods. Furthermore, new and improved reimplementations of conditional inference trees (ctree()) and model-based recursive partitioning (mob()) from the ‘party’ package are provided based on the new infrastructure. A description of this package was published by Hothorn and Zeileis (2015) <https://jmlr.org/papers/v16/hothorn15a.html>.

Depends  R (>= 3.5.0), graphics, grid, libcoin (>= 1.0-0), mvtnorm

Imports  grDevices, stats, utils, survival, Formula (>= 1.2-1), inum (>= 1.0-0), rpart (>= 4.1-11)

Suggests  XML, pmml, rJava, sandwich, strucchange, vcd, AER, mlbench, TH.data (>= 1.0-3), coin (>= 1.1-0), RWeka (>= 0.4-19), datasets, parallel, psychotools (>= 0.3-0), psychotree, party (>= 1.3-0), randomForest

LazyData  yes

License  GPL-2 | GPL-3

URL  http://partykit.r-forge.r-project.org/partykit/

RoxygenNote  6.1.1

NeedsCompilation  yes

Author  Torsten Hothorn [aut, cre] (<https://orcid.org/0000-0001-8301-0471>), Heidi Seibold [ctb] (<https://orcid.org/0000-0002-8960-9642>), Achim Zeileis [aut] (<https://orcid.org/0000-0003-0918-3766>)

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

Repository  CRAN

Date/Publication  2022-06-20 13:30:02 UTC
R topics documented:

cforest ................................................................. 2
ctree ................................................................. 7
ctree_control ....................................................... 10
extree_data .......................................................... 13
extree_fit ............................................................ 15
glmmtree ............................................................. 16
HuntingSpiders ....................................................... 17
lmmtree ............................................................... 19
mob .................................................................... 22
mob_control ........................................................ 25
model_frame_rpart .................................................. 27
nodeapply ............................................................ 28
nodeids ............................................................... 30
panelfunctions ....................................................... 31
party ................................................................. 34
party-coercion ....................................................... 37
party-methods ....................................................... 39
party-plot .......................................................... 41
party-predict ....................................................... 43
partynode ........................................................... 45
partynode-methods ................................................ 47
partysplit ........................................................... 50
prune.modelparty ................................................... 52
varimp ............................................................... 54
WeatherPlay ......................................................... 56

Index 58

cforest Conditional Random Forests

Description

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

Usage

cforest(formula, data, weights, subset, offset, cluster, strata,
na.action = na.pass,
control = ctree_control(teststat = "quad", testtype = "Univ",
  mincriterion = 0, saveinfo = FALSE, ...),
ytrafo = NULL, scores = NULL, ntree = 500L,
perturb = list(replace = FALSE, fraction = 0.632),
mtry = ceiling(sqrt(nvar)), applyfun = NULL, cores = NULL,
trace = FALSE, ...)
## S3 method for class 'cforest'
predict(object, newdata = NULL,  
type = c("response", "prob", "weights", "node"),  
OOB = FALSE, FUN = NULL, simplify = TRUE, scale = TRUE, ...)
## S3 method for class 'cforest'
gettree(object, tree = 1L, ...)

Arguments

- **formula**: a symbolic description of the model to be fit.
- **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. 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.
- **offset**: an optional vector of offset values.
- **cluster**: an optional factor indicating independent clusters. Highly experimental, use at your own risk.
- **strata**: an optional factor for stratified sampling.
- **na.action**: a function which indicates what should happen when the data contain missing value.
- **control**: a list with control parameters, see `ctree_control`. The default values correspond to those of the default values used by `cforest` from the party package. `saveinfo = FALSE` leads to less memory hungry representations of trees. Note that arguments `mtry`, `cores` and `applyfun` in `ctree_control` are ignored for `cforest`, because they are already set.
- **ytrafo**: an optional named list of functions to be applied to the response variable(s) before testing their association with the explanatory variables. Note that this transformation is only performed once for the root node and does not take weights into account (which means, the forest bootstrap or subsetting is ignored, which is almost certainly not a good idea). Alternatively, `ytrafo` can be a function of `data` and `weights`. In this case, the transformation is computed for every node and the corresponding weights. This feature is experimental and the user interface likely to change.
- **scores**: an optional named list of scores to be attached to ordered factors.
- **ntree**: Number of trees to grow for the forest.
- **perturb**: a list with arguments `replace` and `fraction` determining which type of resampling with `replace = TRUE` referring to the n-out-of-n bootstrap and `replace = FALSE` referring to the forest bootstrap.
to sample splitting. fraction is the portion of observations to draw without replacement. Honesty (experimental): If fraction has two elements, the first fraction defines the portion of observations to be used for tree induction, the second fraction defines the portion of observations used for parameter estimation. The sum of both fractions can be smaller than one but must not exceed one. Details can be found in Section 2.4 of Wager and Athey (2018).

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 Inf or manually equal to the number of input variables.

applyfun an optional lapply-style function with arguments function(X, FUN, ...). It is used for computing the variable selection criterion. The default is to use the basic lapply function unless the cores argument is specified (see below).

cores numeric. If set to an integer the applyfun is set to mclapply with the desired number of cores.

trace a logical indicating if a progress bar shall be printed while the forest grows.

object An object as returned by cforest

newdata An optional data frame containing test data.

type a character string denoting the type of predicted value returned, ignored when argument FUN is given. For "response", the mean of a numeric response, the predicted class for a categorical response or the median survival time for a censored response is returned. For "prob" the matrix of conditional class probabilities (simplify = TRUE) or a list with the conditional class probabilities for each observation (simplify = FALSE) is returned for a categorical response. For numeric and censored responses, a list with the empirical cumulative distribution functions and empirical survivor functions (Kaplan-Meier estimate) is returned when type = "prob". "weights" returns an integer vector of prediction weights. For type = "where", a list of terminal node ids for each of the trees in the forest is returned.

OOB a logical defining out-of-bag predictions (only if newdata = NULL). If the forest was fitted with honesty, this option is ignored.

FUN a function to compute summary statistics. Predictions for each node have to be computed based on arguments (y, w) where y is the response and w are case weights.

simplify a logical indicating whether the resulting list of predictions should be converted to a suitable vector or matrix (if possible).

scale a logical indicating scaling of the nearest neighbor weights by the sum of weights in the corresponding terminal node of each tree. In the simple regression forest, predicting the conditional mean by nearest neighbor weights will be equivalent to (but slower!) the aggregation of means.

tree an integer, the number of the tree to extract from the forest.

... additional arguments.
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` perturbed samples of the learning sample. Most of the hyper parameters in `ctree_control` regulate the construction of the conditional inference trees.

Hyper parameters you might want to change are:

1. The number of randomly preselected variables `mtry`, which is fixed to the square root of 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) and Meinshausen (2006) for a description.

Predictions can be computed using `predict`. For observations with zero weights, predictions are computed from the fitted tree when `newdata = NULL`.

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., 2004, 2006a) or to multivariate and ordered responses. Using the rich `partykit` infrastructure allows additional functionality in `cforest`, such as parallel tree growing and probabilistic forecasting (for example via quantile regression forests). Also plotting of single trees from a forest is much easier now.

Unlike `cforest`, `cforest` is entirely written in R which makes customisation much easier at the price of longer computing times. However, trees can be grown in parallel with this R only implementation which renders speed less of an issue. Note that the default values are different from those used in package party, most importantly the default for `mtry` is now data-dependent. `predict(, type = "node")` replaces the `where` function and `predict(, type = "prob")` the `treeresponse` function.

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).

Value

An object of class `cforest`.

References

Examples

```r
## basic example: conditional inference forest for cars data
cf <- cforest(dist ~ speed, data = cars)

## prediction of fitted mean and visualization
nd <- data.frame(speed = 4:25)
nd$mean <- predict(cf, newdata = nd, type = "response")
plot(dist ~ speed, data = cars)
lines(mean ~ speed, data = nd)

## predict quantiles (aka quantile regression forest)
myquantile <- function(y, w) quantile(rep(y, w), probs = c(0.1, 0.5, 0.9))
p <- predict(cf, newdata = nd, type = "response", FUN = myquantile)
colnames(p) <- c("lower", "median", "upper")
nd <- cbind(nd, p)

## visualization with conditional (on speed) prediction intervals
plot(dist ~ speed, data = cars, type = "n")
with(nd, polygon(c(speed, rev(speed)), c(lower, rev(upper)),
                col = "lightgray", border = "transparent"))
points(dist ~ speed, data = cars)
lines(mean ~ speed, data = nd, lwd = 1.5)
lines(median ~ speed, data = nd, lty = 2, lwd = 1.5)
legend("topleft", c("mean", "median", "10% - 90% quantile"),
        lwd = c(1.5, 1.5, 1.5), lty = c(1, 2, 1),
        col = c("black", "black", "lightgray"), bty = "n")

### we may also use predicted conditional (on speed) densities
```


mydensity <- function (y, w) approxfun(density(y, weights = w/sum(w))[1:2], rule = 2)
pd <- predict(cf, newdata = nd, type = "response", FUN = mydensity)

## visualization in heatmap (instead of scatterplot)
## with fitted curves as above
dist <- -10:150
dens <- t(sapply(seq_along(pd), function(i) pd[[i]](dist)))
image(nd$speed, dist, dens, xlab = "speed", col = rev(gray.colors(9)))
lines(mean ~ speed, data = nd, lwd = 1.5)
lines(median - speed, data = nd, lty = 2, lwd = 1.5)
lines(lower ~ speed, data = nd, lty = 2)
lines(upper ~ speed, data = nd, lty = 2)

## Not run:
### 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, ntree = 50),
    OOB = TRUE, type = "response"))

### fit forest to censored response
if (require("TH.data") && require("survival")) {
    data("GBSG2", package = "TH.data")
bst <- cforest(Surv(time, cens) ~ ., data = GBSG2, ntree = 50)

    ### estimate conditional Kaplan-Meier curves
    print(predict(bst, newdata = GBSG2[1:2,], OOB = TRUE, type = "prob"))
    print(gettree(bst))
}

## End(Not run)

ctree

### 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, weights, na.action = na.pass, offset, cluster,
    control = ctree_control(...), ytrafo = NULL,
    converged = NULL, scores = NULL, doFit = TRUE, ...)

---

**ctree**

**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, weights, na.action = na.pass, offset, cluster,
    control = ctree_control(...), ytrafo = NULL,
    converged = NULL, scores = NULL, doFit = TRUE, ...)

---

**Description**

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

**Usage**

ctree(formula, data, subset, weights, na.action = na.pass, offset, cluster,
    control = ctree_control(...), ytrafo = NULL,
    converged = NULL, scores = NULL, doFit = TRUE, ...)

---

**Description**

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

**Usage**

ctree(formula, data, subset, weights, na.action = na.pass, offset, cluster,
    control = ctree_control(...), ytrafo = NULL,
    converged = NULL, scores = NULL, doFit = TRUE, ...)

---
Arguments

- **formula**: A symbolic description of the model to be fit.
- **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.
- **offset**: An optional vector of offset values.
- **cluster**: An optional factor indicating independent clusters. Highly experimental, use at your own risk.
- **na.action**: A function which indicates what should happen when the data contain missing values.
- **control**: A list with control parameters, see `ctree_control`.
- **ytrafo**: An optional named list of functions to be applied to the response variable(s) before testing their association with the explanatory variables. Note that this transformation is only performed once for the root node and does not take weights into account. Alternatively, `ytrafo` can be a function of `data` and `weights`. In this case, the transformation is computed for every node with corresponding weights. This feature is experimental and the user interface likely to change.
- **converged**: An optional function for checking user-defined criteria before splits are implemented. This is not to be used and very likely to change.
- **scores**: An optional named list of scores to be attached to ordered factors.
- **doFit**: A logical, if `FALSE`, the tree is not fitted.
- **...**: Arguments passed to `ctree_control`.

Details

Function `partykit::ctree` is a reimplementation of (most of) `party::ctree` employing the new `party` infrastructure of the `partykit` infrastructure. The vignette `vignette("ctree", package = "partykit")` explains internals of the different implementations.

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 response. 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 repeat 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"` in `ctree_control`) or on the univariate p-values (`testtype = "Univariate"`). 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 without additional (post-)pruning or cross-validation. The level of \texttt{mincriterion} can either be specified to be appropriate for the size of the data set (and 0.95 is typically appropriate for small to moderately-sized data sets) or could potentially be treated like a hyperparameter (see Section~3.4 in Hothorn, Hornik and Zeileis, 2006). 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. The test statistics in each of the nodes can be extracted with the \texttt{sctest} method. (Note that the generic is in the \texttt{strucchange} package so this either needs to be loaded or \texttt{sctest.constparty} has to be called directly.) In cases where splitting stops due to the sample size (e.g., \texttt{minsplit} or \texttt{minbucket} etc.), the test results may be empty.

Predictions can be computed using \texttt{predict}, which returns predicted means, predicted classes or median predicted survival times and more information about the conditional distribution of the response, i.e., class probabilities or predicted Kaplan-Meier curves. For observations with zero weights, predictions are computed from the fitted tree when \texttt{newdata = NULL}.

By default, the scores for each ordinal factor \(x\) are \(1 : \text{length}(x)\), this may be changed for variables in the formula using \texttt{scores = list(x = c(1, 5, 6))}, for example.


\textbf{Value}

An object of class \texttt{party}.

\textbf{References}


\textbf{Examples}

```r
### regression
airq <- subset(airquality, !is.na(Ozone))
airct <- ctree(Ozone ~ ., data = airq)
airct
plot(airct)
mean((airq$Ozone - predict(airct))^2)

### classification
irisct <- ctree(Species ~ ., data = iris)
irisct
plot(irisct)
table(predict(irisct), iris$Species)
```
### estimated class probabilities, a list

```r
tr <- predict(irisct, newdata = iris[1:10,], type = "prob")
```

### survival analysis

```r
if (require("TH.data") &
    require("survival") &
    require("coin") &
    require("Formula")) {

data("GBSG2", package = "TH.data")
(GBSG2ct <- ctree(Surv(time, cens) ~ ., data = GBSG2))
predict(GBSG2ct, newdata = GBSG2[1:2,], type = "response")
plot(GBSG2ct)
}
```

### with weight-dependent log-rank scores

```r
h <- function(y, x, start = NULL, weights, offset, estfun = TRUE, object = FALSE, ...)
{
    if (is.null(weights)) weights <- rep(1, NROW(y))
    s <- logrank_trafo(y[weights > 0,,drop = FALSE])
    r <- rep(0, length(weights))
    r[weights > 0] <- s
    list(estfun = matrix(as.double(r), ncol = 1), converged = TRUE)
}
```

```r
(GBSG2ct <- ctree(Surv(time, cens) ~ ., data = GBSG2, ytrafo = h))
```

### very much the same tree

```r
(GBSG2ct <- ctree(Surv(time, cens) ~ ., data = GBSG2))
```

### multivariate responses

```r
airct2 <- ctree(Ozone + Temp ~ ., data = airq)
plot(airct2)
```

---

**ctree_control**  
Control for Conditional Inference Trees

---

**Description**

Various parameters that control aspects of the ‘ctree’ fit.

**Usage**

```r
ctree_control(teststat = c("quadratic", "maximum"),
splitstat = c("quadratic", "maximum"),
splittest = FALSE,
testtype = c("Bonferroni", "MonteCarlo", "Univariate", "Teststatistic"),
pargs = GenzBretz(),
nmax = c(yx = Inf, z = Inf), alpha = 0.05, mincriterion = 1 - alpha,
logmincriterion = log(mincriterion), msplit = 20L, minbucket = 7L,
minprob = 0.01, stump = FALSE, maxvar = Inf, lookahead = FALSE,
MIA = FALSE, nresample = 9999L,
```

---
ctree_control

```
tol = sqrt(.Machine$double.eps), maxsurrogate = 0L, numsurrogate = FALSE,
mtry = Inf, maxdepth = Inf,
multiway = FALSE, splittry = 2L, intersplit = FALSE, majority = FALSE,
caseweights = TRUE, applyfun = NULL, cores = NULL, saveinfo = TRUE,
update = NULL, splitflavour = c("ctree", "exhaustive"))
```

**Arguments**

- **teststat**: a character specifying the type of the test statistic to be applied for variable selection.
- **splitstat**: a character specifying the type of the test statistic to be applied for splitpoint selection. Prior to version 1.2-0, maximum was implemented only.
- **splittest**: a logical changing linear (the default FALSE) to maximally selected statistics for variable selection. Currently needs testtype = "MonteCarlo".
- **testtype**: a character specifying how to compute the distribution of the test statistic. The first three options refer to p-values as criterion. Teststatistic uses the raw statistic as criterion. Bonferroni and Univariate relate to p-values from the asymptotic distribution (adjusted or unadjusted). Bonferroni-adjusted Monte-Carlo p-values are computed when both Bonferroni and MonteCarlo are given.
- **pargs**: control parameters for the computation of multivariate normal probabilities, see GenzBretz.
- **nmax**: an integer of length two defining the number of bins each variable (in the response yx and the partitioning variables z)) and is divided into prior to tree building. The default Inf does not apply any binning. Highly experimental, use at your own risk.
- **alpha**: a double, the significance level for variable selection.
- **mincriterion**: the value of the test statistic or 1 - p-value that must be exceeded in order to implement a split.
- **logmincriterion**: the value of the test statistic or 1 - p-value that must be exceeded in order to implement a split on the log-scale.
- **minsplits**: 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.
- **minprob**: proportion of observations needed to establish a terminal node.
- **stump**: a logical determining whether a stump (a tree with a maximum of three nodes only) is to be computed.
- **maxvar**: maximum number of variables the tree is allowed to split in.
- **lookahead**: a logical determining whether a split is implemented only after checking if tests in both daughter nodes can be performed.
- **MIA**: a logical determining the treatment of NA as a category in split, see Twala et al. (2008).
- **nresample**: number of permutations for testtype = "MonteCarlo".
- **tol**: tolerance for zero variances.
### ctree_control

- **maxsurrogate**: number of surrogate splits to evaluate.
- **numsurrogate**: a logical for backward-compatibility with party. If TRUE, only at least ordered variables are considered for surrogate splits.
- **mtry**: number of input variables randomly sampled as candidates at each node for random forest like algorithms. The default mtry = Inf means that no random selection takes place. If `ctree_control` is used in `cforest` this argument is ignored.
- **maxdepth**: maximum depth of the tree. The default maxdepth = Inf means that no restrictions are applied to tree sizes.
- **multiway**: a logical indicating if multiway splits for all factor levels are implemented for unordered factors.
- **splittry**: number of variables that are inspected for admissible splits if the best split doesn’t meet the sample size constraints.
- **intersplit**: a logical indicating if splits in numeric variables are simply \( x \leq a \) (the default) or interpolated \( x \leq (a + b) / 2 \). The latter feature is experimental, see Galili and Meilijson (2016).
- **majority**: if FALSE (the default), observations which can’t be classified to a daughter node because of missing information are randomly assigned (following the node distribution). If TRUE, they go with the majority (the default in the first implementation `ctree`) in package party.
- **caseweights**: a logical interpreting weights as case weights.
- **applyfun**: an optional `lapply`-style function with arguments `function(X, FUN, ...)`. It is used for computing the variable selection criterion. The default is to use the basic `lapply` function unless the `cores` argument is specified (see below). If `ctree_control` is used in `cforest` this argument is ignored.
- **cores**: numeric. If set to an integer the applyfun is set to `mclapply` with the desired number of cores. If `ctree_control` is used in `cforest` this argument is ignored.
- **saveinfo**: logical. Store information about variable selection procedure in `info` slot of each `partynode`.
- **update**: logical. If TRUE, the data transformation is updated in every node. The default always was and still is not to update unless `ytrafo` is a function.
- **splitflavour**: use exhaustive search over splits instead of maximally selected statistics (ctree). This feature may change.

### 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 variable with most extreme p-value or test statistic is selected for splitting. If this isn’t possible due to sample size constraints explained in the next paragraph, up to `splittry` other variables are inspected for possible splits.

A split is established when all of the following criteria are met: 1) the sum of the weights in the current node is larger than `minsplit`, 2) a fraction of the sum of weights of more than `minprob`
will be contained in all daughter nodes, 3) the sum of the weights in all daughter nodes exceeds minbucket, and 4) the depth of the tree is smaller than maxdepth. This avoids pathological splits deep down the tree. 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.

In each inner node, maxsurrogate surrogate splits are computed (regardless of any missing values in the learning sample). Factors in test samples whose levels were empty in the learning sample are treated as missing when computing predictions (in contrast to ctree. Note also the different behaviour of majority in the two implementations.

Value

A list.

References


Description

A routine for preprocessing data before an extensible tree can be grown by extree_fit.

Usage

extree_data(formula, data, subset, na.action = na.pass, weights, offset, cluster, strata, scores = NULL, yx = c("none", "matrix"), ytype = c("vector", "data.frame", "matrix"), nmax = c(yx = Inf, z = Inf), ...)

Arguments

formula a formula describing the model of the form y1 + y2 + ... ~ x1 + x2 + ... | z1 + z2 + ....
data an optional data.frame containing the variables in the model.
subset an optional vector specifying a subset of observations to be used in the fitting process.
na.action a function which indicates what should happen when the data contain missing values.
weights      an optional vector of weights.
offset       an optional offset vector.
cluster      an optional factor describing clusters. The interpretation depends on the specific
tree algorithm.
strata       an optional factor describing strata. The interpretation depends on the specific
tree algorithm.
scores       an optional named list of numeric scores to be assigned to ordered factors in the
             z part of the formula.
yx           a character indicating if design matrices shall be computed.
ytype        a character indicating how response variables shall be stored.
nmax         a numeric vector of length two with the maximal number of bins in the response
             and x-part (first element) and the z part. Use Inf to switch-off binning.
...          additional arguments.

Details

This internal functionality will be the basis of implementations of other tree algorithms in future
versions. Currently, only ctree relies on this function.

Value

An object of class extree_data.

Examples

data("iris")

ed <- extree_data(Species ~ Sepal.Width + Sepal.Length | Petal.Width + Petal.Length,
data = iris, nmax = c("yx" = 25, "z" = 10), yx = "matrix")

### the model.frame
mf <- model.frame(ed)
all.equal(mf, iris[, names(mf)])

### binned y ~ x part
model.frame(ed, yxonly = TRUE)

### binned Petal.Width
ed[[4, type = "index"]]

### response
ed$yx$y

### model matrix
ed$yx$x
**extree_fit**

*Fit Extensible Trees.*

**Description**

Basic infrastructure for fitting extensible trees.

**Usage**

```r
extree_fit(data, trafo, converged, selectfun = ctrl$selectfun, splitfun = ctrl$splitfun, svselectfun = ctrl$svselectfun, svsplitfun = ctrl$svsplitfun, partyvars, subset, weights, ctrl, doFit = TRUE)
```

**Arguments**

- `data` an object of class `extree_data`, see `extree_data`.
- `trafo` a function with arguments `subset`, `weights`, `info`, `estfun` and `object`.
- `converged` a function with arguments `subset`, `weights`.
- `selectfun` an optional function for selecting variables.
- `splitfun` an optional function for selecting splits.
- `svselectfun` an optional function for selecting surrogate variables.
- `svsplitfun` an optional function for selecting surrogate splits.
- `partyvars` a numeric vector assigning a weight to each partitioning variable (z in `extree_data`).
- `subset` a sorted integer vector describing a subset.
- `weights` an optional vector of weights.
- `ctrl` control arguments.
- `doFit` a logical indicating if the tree shall be grown (TRUE) or not FALSE.

**Details**

This internal functionality will be the basis of implementations of other tree algorithms in future versions. Currently, only `ctree` relies on this function.

**Value**

An object of class `partynode`. 
glmtree

Generalized Linear Model Trees

Description

Model-based recursive partitioning based on generalized linear models.

Usage

```r
glmtree(formula, data, subset, na.action, weights, offset, cluster,
        family = gaussian, epsilon = 1e-8, maxit = 25, method = "glm.fit", ...)
```

Arguments

- **formula**: symbolic description of the model (of type y ~ z1 + ... + zl or y ~ x1 + ... + xk | z1 + ... + zl; for details see below).
- **data, subset, na.action**: arguments controlling formula processing via `model.frame`.
- **weights**: optional numeric vector of weights. By default these are treated as case weights but the default can be changed in `mob_control`.
- **offset**: optional numeric vector with an a priori known component to be included in the model y ~ x1 + ... + xk (i.e., only when x variables are specified).
- **cluster**: optional vector (typically numeric or factor) with a cluster ID to be employed for clustered covariances in the parameter stability tests.
- **family, method**: specification of a family and fitting method for `glm`.
- **epsilon, maxit**: control parameters passed to `glm.control`.
- **...**: optional control parameters passed to `mob_control`.

Details

Convenience interface for fitting MOBs (model-based recursive partitions) via the `mob` function. `glmtree` internally sets up a model fit function for `mob`, using `glm.fit`. Then `mob` is called using the negative log-likelihood as the objective function.

Compared to calling `mob` by hand, the implementation tries to avoid unnecessary computations while growing the tree. Also, it provides a more elaborate plotting function.

Value

An object of class `glmtree` inheriting from `modelparty`. The `info` element of the overall `party` and the individual nodes contain various informations about the models.

References

See Also

mob, mob_control, lmtree

Examples

if(require("mlbench")) {

## Pima Indians diabetes data
data("PimaIndiansDiabetes", package = "mlbench")

## recursive partitioning of a logistic regression model
pid_tree2 <- glmtree(diabetes ~ glucose | pregnant + pressure + triceps + insulin + mass + pedigree + age,
data = PimaIndiansDiabetes, family = binomial)

## printing whole tree or individual nodes
print(pid_tree2)
print(pid_tree2, node = 1)

## visualization
plot(pid_tree2)
plot(pid_tree2, tp_args = list(cdplot = TRUE))
plot(pid_tree2, terminal_panel = NULL)

## estimated parameters
coef(pid_tree2)
coef(pid_tree2, node = 5)
summary(pid_tree2, node = 5)

## deviance, log-likelihood and information criteria
deviance(pid_tree2)
logLik(pid_tree2)
AIC(pid_tree2)
BIC(pid_tree2)

## different types of predictions
pid <- head(PimaIndiansDiabetes)
predict(pid_tree2, newdata = pid, type = "node")
predict(pid_tree2, newdata = pid, type = "response")
predict(pid_tree2, newdata = pid, type = "link")
}

HuntingSpiders

Abundance of Hunting Spiders

Description

Abundances for 12 species of hunting spiders along with environmental predictors, all rated on a 0–9 scale.
Usage

data("HuntingSpiders")

Format

A data frame containing 28 observations on 18 variables (12 species abundances and 6 environmental predictors).

- **arct.lute** numeric. Abundance of species *Arctosa lutetiana* (on a scale 0–9).
- **pard.lugu** numeric. Abundance of species *Pardosa lugubris* (on a scale 0–9).
- **zora.spin** numeric. Abundance of species *Zora spinimana* (on a scale 0–9).
- **pard.nigr** numeric. Abundance of species *Pardosa nigriceps* (on a scale 0–9).
- **pard.pull** numeric. Abundance of species *Pardosa pullata* (on a scale 0–9).
- **aulo.albi** numeric. Abundance of species *Aulonia albimana* (on a scale 0–9).
- **troc.terr** numeric. Abundance of species *Trochosa terricola* (on a scale 0–9).
- **alop.cune** numeric. Abundance of species *Alopecosa cuneata* (on a scale 0–9).
- **pard.mont** numeric. Abundance of species *Pardosa monticola* (on a scale 0–9).
- **alop.acce** numeric. Abundance of species *Alopecosa accentuata* (on a scale 0–9).
- **alop.fabr** numeric. Abundance of species *Alopecosa fabrilis* (on a scale 0–9).
- **arct.peri** numeric. Abundance of species *Arctosa perita* (on a scale 0–9).
- **water** numeric. Environmental predictor on a scale 0–9.
- **sand** numeric. Environmental predictor on a scale 0–9.
- **moss** numeric. Environmental predictor on a scale 0–9.
- **reft** numeric. Environmental predictor on a scale 0–9.
- **twigs** numeric. Environmental predictor on a scale 0–9.
- **herbs** numeric. Environmental predictor on a scale 0–9.

Details

The data were originally analyzed by Van der Aart and Smeenk-Enserink (1975). De’ath (2002) transformed all variables to the 0–9 scale and employed multivariate regression trees.

Source

Package mvpart (currently archived, see https://CRAN.R-project.org/package=mvpart).

References


Examples

```r
## load data
data("HuntingSpiders", package = "partykit")

## fit multivariate tree for 12-dimensional species abundance
## (warnings by mvtnorm are suppressed)
suppressWarnings(sptree <- ctree(arct.lute + pard.lugu + zora.spin + pard.nigr + pard.pull +
aulo.albi + troc.terr + alop.cune + pard.mont + alop.acce + alop.fabr +
arct.peri - herbs + reft + moss + sand + twigs + water, data = HuntingSpiders,
teststat = "max", msplit = 5))
plot(sptree, terminal_panel = node_barplot)
```

---

**lmtree**  
*Linear Model Trees*

**Description**

Model-based recursive partitioning based on least squares regression.

**Usage**

```r
lmtree(formula, data, subset, na.action, weights, offset, cluster, ...)
```

**Arguments**

- `formula`  
  symbolic description of the model (of type `y ~ z1 + ... + zl` or `y ~ x1 + ... + xk | z1 + ... + zl`; for details see below).

- `data`, `subset`, `na.action`  
  arguments controlling formula processing via `model.frame`.

- `weights`  
  optional numeric vector of weights. By default these are treated as case weights but the default can be changed in `mob_control`.

- `offset`  
  optional numeric vector with an a priori known component to be included in the model `y ~ x1 + ... + xk` (i.e., only when `x` variables are specified).

- `cluster`  
  optional vector (typically numeric or factor) with a cluster ID to be employed for clustered covariances in the parameter stability tests.

- `...`  
  optional control parameters passed to `mob_control`.

**Details**

Convenience interface for fitting MOBs (model-based recursive partitions) via the `mob` function. `lmtree` internally sets up a model fit function for `mob`, using either `lm.fit` or `lm.wfit` (depending on whether weights are used or not). Then `mob` is called using the residual sum of squares as the objective function.

Compared to calling `mob` by hand, the implementation tries to avoid unnecessary computations while growing the tree. Also, it provides a more elaborate plotting function.
Value

An object of class `lm-tree` inheriting from `modelparty`. The info element of the overall party and the individual nodes contain various informations about the models.

References


See Also

`mob`, `mob_control`, `glm-tree`

Examples

```r
if(require("mlbench")) {

## Boston housing data
data("BostonHousing", package = "mlbench")
BostonHousing <- transform(BostonHousing,
    chas = factor(chas, levels = 0:1, labels = c("no", "yes")),
    rad = factor(rad, ordered = TRUE))

## linear model tree
bh_tree <- lmtree(medv ~ log(lstat) + I(rm^2) | zn +
    indus + chas + nox + age + dis + rad + tax + crim + b + pptratio,
    data = BostonHousing, minsize = 40)

## printing whole tree or individual nodes
print(bh_tree)
print(bh_tree, node = 7)

## plotting
plot(bh_tree)
plot(bh_tree, tp_args = list(which = "log(lstat")))
plot(bh_tree, terminal_panel = NULL)

## estimated parameters
coef(bh_tree)
coef(bh_tree, node = 9)
summary(bh_tree, node = 9)

## various ways for computing the mean squared error (on the training data)
mean((BostonHousing$medv - fitted(bh_tree))^2)
mean(residuals(bh_tree)^2)
deviance(bh_tree)/sum(weights(bh_tree))
deviance(bh_tree)/nobs(bh_tree)

## log-likelihood and information criteria
logLik(bh_tree)
AIC(bh_tree)
BIC(bh_tree)
}
```
## (Note that this penalizes estimation of error variances, which
## were treated as nuisance parameters in the fitting process.)

## different types of predictions
bh <- BostonHousing[c(1, 10, 50), ]
predict(bh_tree, newdata = bh, type = "node")
predict(bh_tree, newdata = bh, type = "response")
predict(bh_tree, newdata = bh, type = function(object) summary(object)$r.squared)

}  

if(require("AER")) {

## Demand for economics journals data
data("Journals", package = "AER")
Journals <- transform(Journals,
  age = 2000 - foundingyear,
  chars = charpp * pages)

## linear regression tree (OLS)
j_tree <- lmtree(log(subs) ~ log(price/citations) | price + citations +
  age + chars + society, data = Journals, minsize = 10, verbose = TRUE)

## printing and plotting
j_tree
plot(j_tree)

## coefficients and summary
coef(j_tree, node = 1:3)
summary(j_tree, node = 1:3)

}

if(require("AER")) {

## Beauty and teaching ratings data
data("TeachingRatings", package = "AER")

## linear regression (WLS)
## null model
tr_null <- lm(eval ~ 1, data = TeachingRatings, weights = students, 
  subset = credits == "more")

## main effects
tr_lm <- lm(eval ~ beauty + gender + minority + native + tenure + division,
  data = TeachingRatings, weights = students, subset = credits == "more")

## tree
tr_tree <- lmtree(eval ~ beauty | minority + age + gender + division + native + tenure,
  data = TeachingRatings, weights = students, subset = credits == "more",
  caseweights = FALSE)

## visualization
plot(tr_tree)

## beauty slope coefficient
coef(tr_lm)[2]
coef(tr_tree)[, 2]

## R-squared
1 - deviance(tr_lm)/deviance(tr_null)
1 - deviance(tr_tree)/deviance(tr_null)
}

---

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, data, subset, na.action, weights, offset, cluster,
    fit, control = mob_control(), ...)

Arguments

- **formula**: symbolic description of the model (of type \( y \sim z_1 + \ldots + z_l \) or \( y \sim x_1 + \ldots + x_k \mid z_1 + \ldots + z_l \); for details see below).
- **data, subset, na.action**: arguments controlling formula processing via `model.frame`.
- **weights**: optional numeric vector of weights. By default these are treated as case weights but the default can be changed in `mob_control`.
- **offset**: optional numeric vector with an a priori known component to be included in the model \( y \sim x_1 + \ldots + x_k \) (i.e., only when \( x \) variables are specified).
- **cluster**: optional vector (typically numeric or factor) with a cluster ID to be passed on to the `fit` function and employed for clustered covariances in the parameter stability tests.
- **fit**: function. A function for fitting the model within each node. For details see below.
- **control**: A list with control parameters as returned by `mob_control`.
- **...**: Additional arguments passed to the `fit` function.
Details

Model-based partitioning fits a model tree using two groups of variables: (1) The model variables which can be just a (set of) response(s) $y$ or additionally include regressors $x_1, \ldots, x_k$. These are used for estimating the model parameters. (2) Partitioning variables $z_1, \ldots, z_l$, which are used for recursively partitioning the data. The two groups of variables are either specified as $y \sim z_1 + \ldots + z_l$ (when there are no regressors) or $y \sim x_1 + \ldots + x_k \mid z_1 + \ldots + z_l$ (when the model part contains regressors). Both sets of variables may in principle be overlapping.

To fit a tree model the following algorithm is used.

1. Fit a model to the $y$ or $y$ and $x$ variables using the observations in the current node
2. Assess the stability of the model parameters with respect to each of the partitioning variables $z_1, \ldots, z_l$. If there is some overall instability, choose the variable $z$ associated with the smallest $p$ value for partitioning, otherwise stop.
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.
4. Refit the model in both kid subsamples 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 specifying the fit function two approaches are possible:

(1) It can be a function fit($y, x = \text{NULL}, \text{start} = \text{NULL}, \text{weights} = \text{NULL}, \text{offset} = \text{NULL}, \ldots$). The arguments $y, x, \text{weights}, \text{offset}$ will be set to the corresponding elements in the current node of the tree. Additionally, starting values will sometimes be supplied via start. Of course, the fit function can choose to ignore any arguments that are not applicable, e.g., if there are no regressors $x$ in the model or if starting values or not supported. The returned object needs to have a class that has associated coef, logLik, and estfun methods for extracting the estimated parameters, the maximized log-likelihood, and the empirical estimating function (i.e., score or gradient contributions), respectively.

(2) It can be a function fit($y, x = \text{NULL}, \text{start} = \text{NULL}, \text{weights} = \text{NULL}, \text{offset} = \text{NULL}, \ldots, \text{estfun} = \text{FALSE}, \text{object} = \text{FALSE}$). The arguments have the same meaning as above but the returned object needs to have a different structure. It needs to be a list with elements coefficients (containing the estimated parameters), objfun (containing the minimized objective function), estfun (the empirical estimating functions), and object (the fitted model object). The elements estfun, or object should be NULL if the corresponding argument is set to FALSE.

Internally, a function of type (2) is set up by mob() in case a function of type (1) is supplied. However, to save computation time, a function of type (2) may also be specified directly.

For the fitted MOB tree, several standard methods are provided such as print, predict, residuals, logLik, deviance, weights, coef and summary. Some of these rely on reusing the corresponding methods for the individual model objects in the terminal nodes. Functions such as coef, print, summary also take a node argument that can specify the node IDs to be queried. Some examples are given below.

More details can be found in vignette("mob", package = "partykit"). An overview of the connections to other functions in the package is provided by Hothorn and Zeileis (2015).
Value

An object of class `mobparty` inheriting from `party`. The info element of the overall party and the individual nodes contain various informations about the models.

References


See Also

`mob_control`, `lmtree`, `glmtree`

Examples

```r
if(require("mlbench")) {

## Pima Indians diabetes data
data("PimaIndiansDiabetes", package = "mlbench")

## a simple basic fitting function (of type 1) for a logistic regression
logit <- function(y, x, start = NULL, weights = NULL, offset = NULL, ...) {
  glm(y ~ 0 + x, family = binomial, start = start, ...)
}

## set up a logistic regression tree
pid_tree <- mob(diabetes ~ glucose | pregnant + pressure + triceps + insulin +
  mass + pedigree + age, data = PimaIndiansDiabetes, fit = logit)

## see lmtree() and glmtree() for interfaces with more efficient fitting functions

## print tree
print(pid_tree)

## print information about (some) nodes
print(pid_tree, node = 3:4)

## visualization
plot(pid_tree)

## coefficients and summary
coef(pid_tree)
coef(pid_tree, node = 1)
summary(pid_tree, node = 1)

## average deviance computed in different ways
mean(residuals(pid_tree)^2)
deviance(pid_tree)/sum(weights(pid_tree))
deviance(pid_tree)/nobs(pid_tree)
```
## log-likelihood and information criteria

```r
logLik(pid_tree)
AIC(pid_tree)
BIC(pid_tree)
```

## predicted nodes

```r
predict(pid_tree, newdata = head(PimaIndiansDiabetes, 6), type = "node")
```

```r
## other types of predictions are possible using lmtree()/glmmtree()
```

---

## Description

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

## Usage

```r
mob_control(alpha = 0.05, bonferroni = TRUE, minsize = NULL, maxdepth = Inf,
mtry = Inf, trim = 0.1, breakties = FALSE, parm = NULL, dfsplit = TRUE, prune = NULL,
restart = TRUE, verbose = FALSE, caseweights = TRUE, ytype = "vector", xtype = "matrix",
terminal = "object", inner = terminal, model = TRUE, numsplit = "left",
catsplit = "binary", vcov = "opg", ordinal = "chisq", nrep = 10000,
minsplit = minsize, minbucket = minsize, applyfun = NULL, cores = NULL)
```

## 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 $\alpha$ (and the stopping criteria minsize and maxdepth are not fulfilled).

- **bonferroni** logical. Should $p$ values be Bonferroni corrected?

- **minsize, minsplit, minbucket** integer. The minimum number of observations in a node. If NULL, the default is to use 10 times the number of parameters to be estimated (divided by the number of responses per observation if that is greater than 1). minsize is the recommended name and minsplit/minbucket are only included for backward compatibility with previous versions of mob and compatibility with ctree, respectively.

- **maxdepth** integer. The maximum depth of the tree.

- **mtry** integer. The number of partitioning variables randomly sampled as candidates in each node for forest-style algorithms. If mtry is greater than the number of partitioning variables, no random selection is performed. (Thus, by default all available partitioning variables are considered.)

- **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.
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).

dfsplit logical or numeric. as.integer(dfsplit) is the degrees of freedom per selected split employed when computing information criteria etc.

prune character, numeric, or function for specifying post-pruning rule. If prune is NULL (the default), no post-pruning is performed. For likelihood-based mob() trees, prune can be set to "AIC" or "BIC" for post-pruning based on the corresponding information criteria. More general rules (also in scenarios that are not likelihood-based), can be specified by function arguments to prune, for details see below.

restart logical. When determining the optimal split point in a numerical variable: Should model estimation be restarted with NULL starting values for each split? The default is TRUE. If FALSE, then the parameter estimates from the previous split point are used as starting values for the next split point (because in practice the difference are often not huge). (Note that in that case a for loop is used instead of the applyfun for fitting models across sample splits.)

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?

caseweights logical. Should weights be interpreted as case weights? If TRUE, the number of observations is sum(weights), otherwise it is sum(weights > 0).

ytype, xtype character. Specification of how mob should preprocess y and x variables. Possible choice are: "vector" (for y only), i.e., only one variable; "matrix", i.e., the model matrix of all variables; "data.frame", i.e., a data frame of all variables.

terminal, inner character. Specification of which additional information ("estfun", "object", or both) should be stored in each node. If NULL, no additional information is stored.

model logical. Should the full model frame be stored in the resulting object?

numsplit character indicating how splits for numeric variables should be justified. Because any splitpoint in the interval between the last observation from the left child segment and the first observation from the right child segment leads to the same observed split, two options are available in mob_control: Either, the split is "left"-justified (the default for backward compatibility) or "center"-justified using the midpoint of the possible interval.

catsplit character indicating how (unordered) categorical variables should be splitted. By default the best "binary" split is searched (by minimizing the objective function). Alternatively, if set to "multiway", the node is simply splitted into all levels of the categorical variable.

vcov character indicating which type of covariance matrix estimator should be employed in the parameter instability tests. The default is the outer product of gradients ("opg"). Alternatively, vcov = "info" employs the information matrix and vcov = "sandwich" the sandwich matrix (both of which are only sensible for maximum likelihood estimation).
character indicating which type of parameter instability test should be employed for ordinal partitioning variables (i.e., ordered factors). This can be "chisq", "max", or "L2". If "chisq" then the variable is treated as unordered and a chi-squared test is performed. If "L2", then a maxLM-type test as for numeric variables is carried out but correcting for ties. This requires simulation of p-values via `catL2BB` and requires some computation time. For "max" a weighted double maximum test is used that computes p-values via `pmvnorm`.

numeric. Number of replications in the simulation of p-values for the ordinal "L2" statistic (if used).

an optional `lapply`-style function with arguments `function(X, FUN, ...)`. It is used for refitting the model across potential sample splits. The default is to use the basic `lapply` function unless the `cores` argument is specified (see below).

numeric. If set to an integer the `applyfun` is set to `mclapply` with the desired number of cores.

See `mob` for more details and references.

For post-pruning, `prune` can be set to a function(`objfun`, `df`, `nobs`) which either returns `TRUE` to signal that a current node can be pruned or `FALSE`. All supplied arguments are of length two: `objfun` is the sum of objective function values in the current node and its child nodes, respectively. `df` is the degrees of freedom in the current node and its child nodes, respectively. `nobs` is vector with the number of observations in the current node and the total number of observations in the dataset, respectively.

If the objective function employed in the `mob()` call is the negative log-likelihood, then a suitable function is set up on the fly by comparing `(2 * objfun + penalty * df)` in the current and the daughter nodes. The penalty can then be set via a numeric or character value for `prune`: AIC is used if `prune = "AIC"` or `prune = 2` and BIC if `prune = "BIC"` or `prune = log(n)`.

A list of class `mob_control` containing the control parameters.

See Also

`mob`

A model.frame method for rpart objects.

`model_frame_rpart(formula, ...)`
Arguments

- formula: an object of class rpart.
- ...: additional arguments.

Details

A model.frame method for rpart objects. Because it is no longer possible to overwrite existing methods, the function name is a little different here.

Value

A model frame.

Description

Returns a list of values obtained by applying a function to party or partynode objects.

Usage

nodeapply(obj, ids = 1, FUN = NULL, ...)

## S3 method for class 'partynode'
nodeapply(obj, ids = 1, FUN = NULL, ...)

## S3 method for class 'party'
nodeapply(obj, ids = 1, FUN = NULL, by_node = TRUE, ...)

Arguments

- obj: an object of class partynode or party.
- ids: integer vector of node identifiers to apply over.
- FUN: a function to be applied to nodes. By default, the node itself is returned.
- by_node: a logical indicating if FUN is applied to subsets of party objects or partynode objects (default).
- ...: additional arguments.

Details

Function FUN is applied to all nodes with node identifiers in ids for a partynode object. The method for party by default calls the nodeapply method on its node slot. If by_node is FALSE, it is applied to a party object with root node ids.

Value

A list of results of length length(ids).
Examples

```r
## a tree as flat list structure
nodelist <- list(
  # root node
  list(id = 1L, split = partysplit(varid = 4L, breaks = 1.9),
       kids = 2:3),
  # V4 <= 1.9, terminal node
  list(id = 2L, info = "terminal A"),
  # V4 > 1.9
  list(id = 3L, split = partysplit(varid = 5L, breaks = 1.7),
       kids = c(4L, 7L)),
  # V5 <= 1.7
  list(id = 4L, split = partysplit(varid = 4L, breaks = 4.8),
       kids = 5:6),
  # V4 <= 4.8, terminal node
  list(id = 5L, info = "terminal B"),
  # V4 > 4.8, terminal node
  list(id = 6L, info = "terminal C"),
  # V5 > 1.7, terminal node
  list(id = 7L, info = "terminal D")
)
## convert to a recursive structure
node <- as.partynode(nodelist)
## return root node
nodeapply(node)
## return info slots of terminal nodes
nodeapply(node, ids = nodeids(node, terminal = TRUE),
          FUN = function(x) info_node(x))
## fit tree using rpart
library("rpart")
rp <- rpart(Kyphosis ~ Age + Number + Start, data = kyphosis)
## coerce to \code{constparty}
node <- as.party(rp)
## extract nodeids
nodeids(rp)
unlist(nodeapply(node_party(rp), ids = nodeids(rp),
                FUN = id_node))
unlist(nodeapply(rp, ids = nodeids(rp), FUN = id_node))
## but root nodes of party objects always have id = 1
unlist(nodeapply(rp, ids = nodeids(rp), FUN = function(x)
                id_node(node_party(x)), by_node = FALSE))
```
nodeids

Extract Node Identifiers

Description

Extract unique identifiers from inner and terminals nodes of a partynode object.

Usage

```r
nodeids(obj, ...)  
## S3 method for class 'partynode'
nodeids(obj, from = NULL, terminal = FALSE, ...)
## S3 method for class 'party'
nodeids(obj, from = NULL, terminal = FALSE, ...)
get_paths(obj, i)
```

Arguments

- `obj` an object of class partynode or party.
- `from` an integer specifying node to start from.
- `terminal` logical specifying if only node identifiers of terminal nodes are returned.
- `i` a vector of node identifiers.
- `...` additional arguments.

Details

The identifiers of each node are extracted from `nodeids`. `get_paths` returns the paths for extracting the corresponding nodes using list subsets.

Value

A vector of node identifiers.

Examples

```r
## a tree as flat list structure
nodelist <- list(
  # root node
  list(id = 1L, split = partysplit(varid = 4L, breaks = 1.9),
       kids = 2:3),
  # V4 <= 1.9, terminal node
  list(id = 2L),
  # V4 > 1.9
  list(id = 3L, split = partysplit(varid = 1L, breaks = 1.7),
       kids = c(4L, 7L)),
  # V1 <= 1.7
)
Panelfunctions

Panelfunctions for Visualization of Party Trees

Description

The plot method for party and constparty 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(obj, id = TRUE, pval = TRUE, abbreviate = FALSE, fill = "white", gp = gpar())
panelfunctions

node_terminal(obj, digits = 3, abbreviate = FALSE, fill = c("lightgray", "white"); id = TRUE, just = c("center", "top"), top = 0.85, align = c("center", "left", "right"); gp = NULL, FUN = NULL, height = NULL, width = NULL)

edge_simple(obj, digits = 3, abbreviate = FALSE, justmin = Inf, just = c("alternate", "increasing", "decreasing", "equal"), fill = "white")

node_boxplot(obj, col = "black", fill = "lightgray", bg = "white", width = 0.5, yscale = NULL, ylines = 3, cex = 0.5, id = TRUE, mainlab = NULL, gp = gpar())

node_barplot(obj, col = "black", fill = NULL, bg = "white", beside = NULL, ymax = NULL, ylines = NULL, widths = 1, gap = NULL, reverse = NULL, rot = 0, just = c("center", "top"), id = TRUE, mainlab = NULL, text = c("none", "horizontal", "vertical"), gp = gpar())

node_surv(obj, col = "black", bg = "white", yscale = c(0, 1), ylines = 2, id = TRUE, mainlab = NULL, gp = gpar(), ...)

node_ecdf(obj, col = "black", bg = "white", ylines = 2, id = TRUE, mainlab = NULL, gp = gpar(), ...)

node_bivplot(mobobj, which = NULL, id = TRUE, pop = TRUE, pointcol = "black", pointcex = 0.5, boxcol = "black", boxwidth = 0.5, boxfill = "lightgray", bg = "white", fitmean = TRUE, linecol = "red", cdpplot = FALSE, fivenum = TRUE, breaks = NULL, ylines = NULL, xlab = FALSE, ylab = FALSE, margins = rep(1.5, 4), mainlab = NULL, ...)
justification of terminal panel viewport (node_terminal), or labels (edge_simple, node_barplot).

justmin
minimum average edge label length to employ justification via just in edge_panel, otherwise just = "equal" is used. Thus, by default "equal" justification is always used but other justifications could be employed for finite justmin.

top
in case of top justification, the npc coordinate at which the viewport is justified.

align
alignment of text within terminal panel viewport.

ylines
number of lines for spaces in y-direction.

widths
widths in barplots.

boxwidth
width in boxplots (called width in node_boxplot).

gap
gap between bars in a barplot (node_barplot).

yscale
limits in y-direction

ymax
upper limit in y-direction

cex, pointcex
character extension of points in scatter plots.

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.

rot
arguments passed to grid.text for the x-axis labeling.

gp
graphical parameters.

FUN
function for formatting the info, passed to formatinfo_node.

height, width
numeric, number of lines/columns for printing text.

mobobj
an object of class modelparty as computed by mob.

which
numeric or character. Optional selection of subset of regressor variables. By default one panel for each regressor variable is drawn.

pop
logical. Should the viewports in the individual nodes be popped after drawing?

fitmean
logical. Should the fitted mean function be visualized?

cdplot
logical. Should a CD plot (or a spineplot) be drawn when the response variable is categorical?

fivenum
logical. Should the five-number summary be used for splitting the x-axis in spineplots?

breaks
numeric. Optional numeric vector with breaks for the x-axis in splineplots.

xlab, ylab
character. Optional annotation for x-axis and y-axis.

margins
numeric. Margins around drawing area in viewport.

mainlab
character or function. An optional title for the plot. Either a character or a function(id, nobs).

varlab
logical. Should the individual variable labels be attached to the mainlab for multivariate responses?

text
logical or character. Should percentage labels be drawn for each bar? The default is "none" or equivalently FALSE. Can be set to TRUE (or "horizontal") or alternatively "vertical".

additional arguments passed to callies (for example to survfit).
Details

The plot methods for party and constparty 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 visualized, 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 modeled.

For univariate regressions (typically fitted by ), node_surv returns a functions that plots Kaplan-Meier curves in each terminal node; node_barplot, node_boxplot, node_hist, node_ecdf and node_density can be used to plot bar plots, box plots, histograms, empirical cumulative distribution functions 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.

For multivariate responses in ctree, the panel function node_mvar generates one plot for each response.

References


party Recursive Partytioning

Description

A class for representing decision trees and corresponding accessor functions.

Usage

party(node, data, fitted = NULL, terms = NULL, names = NULL, info = NULL)
## S3 method for class 'party'
names(x)
## S3 replacement method for class 'party'
names(x) <- value
```r
# Default S3 method:
data_party(party, id = 1L)
node_party(party)
is.constparty(party)
is.simpleparty(party)
```

### Arguments

- **node**
  - an object of class `partynode`.
- **data**
  - a (potentially empty) `data.frame`.
- **fitted**
  - an optional `data.frame` with `nrow(data)` rows (only if `nrow(data) != 0`) and containing at least the fitted terminal node identifiers as element (fitted). In addition, weights may be contained as element (weights) and responses as (response).
- **terms**
  - an optional `terms` object.
- **names**
  - an optional vector of names to be assigned to each node of node.
- **info**
  - additional information.
- **x**
  - an object of class `party`.
- **party**
  - an object of class `party`.
- **value**
  - a character vector of up to the same length as x, or NULL.
- **id**
  - a node identifier.

### Details

Objects of class `party` basically consist of a `partynode` object representing the tree structure in a recursive way and data. The data argument takes a `data.frame` which, however, might have zero columns. Optionally, a `data.frame` with at least one variable (fitted) containing the terminal node numbers of data used for fitting the tree may be specified along with a `terms` object or any additional (currently unstructured) information as `info`. Argument `names` defines names for all nodes in node.

Method names can be used to extract or alter names for nodes. Function `node_party` returns the node element of a `party` object. Further methods for `party` objects are documented in `party-methods` and `party-predict`. Trees of various flavors can be coerced to `party`, see `party-coercion`.

Two classes inherit from class `party` and impose additional assumptions on the structure of this object: Class `constparty` requires that the fitted slot contains a partitioning of the learning sample as a factor ("fitted") and the response values of all observations in the learning sample as ("response"). This structure is most flexible and allows for graphical display of the response values in terminal nodes as well as for computing predictions based on arbitrary summary statistics.

Class `simpleparty` assumes that certain pre-computed information about the distribution of the response variable is contained in the `info` slot nodes. At the moment, no formal class is used to describe this information.
Value

The constructor returns an object of class `party`:

- `node` an object of class `partynode`.
- `data` a (potentially empty) `data.frame`.
- `fitted` an optional `data.frame` with `nrow(data)` rows (only if `nrow(data) != 0`) and containing at least the fitted terminal node identifiers as element `(fitted)`. In addition, weights may be contained as element `(weights)` and responses as `(response)`.
- `terms` an optional `terms` object.
- `names` an optional vector of names to be assigned to each node of `node`.
- `info` additional information.

`names` can be used to set and retrieve names of nodes and `node_party` returns an object of class `partynode`. `data_party` returns a data frame with observations contained in node `id`.

References


Examples

```r
### data ###
## artificial WeatherPlay data
data("WeatherPlay", package = "partykit")
str(WeatherPlay)

### splits ###
## split in overcast, humidity, and windy
sp_o <- partysplit(1L, index = 1:3)
sp_h <- partysplit(3L, breaks = 75)
sp_w <- partysplit(4L, index = 1:2)

## query labels
character_split(sp_o)

### nodes ###
## set up partynode structure
pn <- partynode(1L, split = sp_o, kids = list(
  partynode(2L, split = sp_h, kids = list(
    partynode(3L, info = "yes"),
    partynode(4L, info = "no"))),
  partynode(5L, info = "yes"),
  partynode(6L, split = sp_w, kids = list(
    partynode(7L, info = "yes"),
    partynode(8L, info = "no")))))
pn
```
### tree ###

**party**: associate recursive partynode structure with data

```r
py <- party(pn, WeatherPlay)
py
plot(py)
```

### variations ###

#### tree stump

```r
n1 <- partynode(id = 1L, split = sp_o, kids = lapply(2L:4L, partynode))
print(n1, data = WeatherPlay)
```

#### query fitted nodes and kids ids

```r
fitted_node(n1, data = WeatherPlay)
kidids_node(n1, data = WeatherPlay)
```

#### tree with full data sets

```r
t1 <- party(n1, data = WeatherPlay)
```

#### tree with empty data set

```r
party(n1, data = WeatherPlay[0, ])
```

#### constant-fit tree

```r
t2 <- party(n1,
data = WeatherPlay,
fitted = data.frame(
  "(fitted)" = fitted_node(n1, data = WeatherPlay),
  "(response)" = WeatherPlay$play,
  check.names = FALSE),
terms = terms(play ~ ., data = WeatherPlay),
)
t2 <- as.constparty(t2)
t2
plot(t2)
```

---

**party-coercion**

### Coercion Functions

**Description**

Functions coercing various objects to objects of class party.

**Usage**

```r
as.party(obj, ...)
```

#### S3 method for class 'rpart'

```r
as.party(obj, data = TRUE, ...)
```

#### S3 method for class 'Weka_tree'

---
as.party(obj, data = TRUE, ...)  
## S3 method for class 'XMLNode'
as.party(obj, ...)
pmmlTreeModel(file, ...)
as.constparty(obj, ...)
as.simpleparty(obj, ...)
## S3 method for class 'party'
as.simpleparty(obj, ...)
## S3 method for class 'simpleparty'
as.simpleparty(obj, ...)
## S3 method for class 'constparty'
as.simpleparty(obj, ...)
## S3 method for class 'XMLNode'
as.simpleparty(obj, ...)

Arguments

obj an object of class `rpart`, `Weka_tree`, `XMLnode` or objects inheriting from `party`.
data logical. Should the model frame associated with the fitted obj be included in the data of the party?
file a file name of a XML file containing a PMML description of a tree.
... additional arguments.

Details

Trees fitted using functions `rpart` or `J48` are coerced to `party` objects. By default, objects of class `constparty` are returned.

When information about the learning sample is available, `party` objects can be coerced to objects of class `constparty` or `simpleparty` (see `party` for details).

Value

All methods return objects of class `party`.

Examples

```r
## fit tree using rpart
library("rpart")
rp <- rpart(Kyphosis ~ Age + Number + Start, data = kyphosis)

## coerce to 'constparty'
as.party(rp)
```
Methods for Party Objects

Description

Methods for computing on party objects.

Usage

```r
## S3 method for class 'party'
print(x,
   terminal_panel = function(node)
      formatinfo_node(node, default = "*", prefix = ": "),
   tp_args = list(),
   inner_panel = function(node) "", ip_args = list(),
   header_panel = function(party) ",",
   footer_panel = function(party) ",",
   digits = getOption("digits") - 2, ...)
## S3 method for class 'simpleparty'
print(x, digits = getOption("digits") - 4,
   header = NULL, footer = TRUE, ...)
## S3 method for class 'constparty'
print(x, FUN = NULL, digits = getOption("digits") - 4,
   header = NULL, footer = TRUE, ...)
## S3 method for class 'party'
length(x)
## S3 method for class 'party'
x[i, ...]
## S3 method for class 'party'
x[[i, ...]]
## S3 method for class 'party'
depth(x, root = FALSE, ...)
## S3 method for class 'party'
width(x, ...)
## S3 method for class 'party'
nodeprune(x, ids, ...)
```

Arguments

- `x` an object of class `party`.
- `i` an integer specifying the root of the subtree to extract.
- `terminal_panel` a panel function for printing terminal nodes.
- `tp_args` a list containing arguments to `terminal_panel`.
- `inner_panel` a panel function for printing inner nodes.
- `ip_args` a list containing arguments to `inner_panel`.

```
header_panel  a panel function for printing the header.
footer_panel  a panel function for printing the footer.
digits       number of digits to be printed.
header       header to be printed.
footer       footer to be printed.
FUN          a function to be applied to nodes.
root         a logical. Should the root count be counted in depth?
ids          a vector of node ids (or their names) to be pruned-off.
...          additional arguments.

Details

length gives the number of nodes in the tree (in contrast to the length method for \texttt{partynode} objects which returns the number of kid nodes in the root), depth the depth of the tree and width the number of terminal nodes. The subset methods extract subtrees and the \texttt{print} method generates a textual representation of the tree. \texttt{nodeprune} prunes-off nodes and makes sure that the node ids of the resulting tree are in pre-order starting with root node id 1. For \texttt{constparty} objects, the fitted slot is also changed.

Examples

```r
## a tree as flat list structure
nodelist <- list(
  # root node
  list(id = 1L, split = partysplit(varid = 4L, breaks = 1.9),
       kids = 2:3),
  # V4 <= 1.9, terminal node
  list(id = 2L),
  # V4 > 1.9
  list(id = 3L, split = partysplit(varid = 5L, breaks = 1.7),
       kids = c(4L, 7L)),
  # V5 <= 1.7
  list(id = 4L, split = partysplit(varid = 4L, breaks = 4.8),
       kids = 5:6),
  # V4 <= 4.8, terminal node
  list(id = 5L),
  # V4 > 4.8, terminal node
  list(id = 6L),
  # V5 > 1.7, terminal node
  list(id = 7L)
)

## convert to a recursive structure
node <- as.partynode(nodelist)

## set up party object
data("iris")
tree <- party(node, data = iris,
```

party-plot

Visualization of Trees

Description

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

Usage

## S3 method for class 'party'
plot(x, main = NULL,
     terminal_panel = node_terminal, tp_args = list(),
     inner_panel = node_inner, ip_args = list(),
     edge_panel = edge_simple, ep_args = list(),
     drop_terminal = FALSE, tnex = 1,
     newpage = TRUE, pop = TRUE, gp = gpar(),
     margins = NULL, ...)

## S3 method for class 'constparty'
plot(x, main = NULL,
     terminal_panel = NULL, tp_args = list(),
     inner_panel = node_inner, ip_args = list(),
     edge_panel = edge_simple, ep_args = list(),
     type = c("extended", "simple"),
     drop_terminal = NULL,
     tnex = NULL, newpage = TRUE, pop = TRUE, gp = gpar(),
     ...)

## S3 method for class 'simpleparty'
plot(x, digits = getOption("digits") - 4, tp_args = NULL, ...)
Arguments

x an object of class party or constparty.
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.
ininner_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.
gp graphical parameters.
margins numeric vector of margin sizes.
digits number of digits to be printed.
... additional arguments passed to callsies.

Details

This plot method for party 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.

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).
party-predict

References


See Also

node_inner, node_terminal, edge_simple, node_barplot, node_boxplot.

---

party-predict  Tree Predictions

Description

Compute predictions from party objects.

Usage

## S3 method for class 'party'
predict(object, newdata = NULL, perm = NULL, ...)
predict_party(party, id, newdata = NULL, ...)
## Default S3 method:
predict_party(party, id, newdata = NULL, FUN = NULL, ...)
## S3 method for class 'constparty'
predict_party(party, id, newdata = NULL,
            type = c("response", "prob", "quantile", "density", "node"),
            at = if (type == "quantile") c(0.1, 0.5, 0.9),
            FUN = NULL, simplify = TRUE, ...)
## S3 method for class 'simpleparty'
predict_party(party, id, newdata = NULL,
            type = c("response", "prob", "node"), ...)

Arguments

object  objects of class party.
newdata  an optional data frame in which to look for variables with which to predict, if omitted, the fitted values are used.
perm  an optional character vector of variable names. Splits of nodes with a primary split in any of these variables will be permuted (after dealing with surrogates). Note that surrogate split in the perm variables will no be permuted.
party  objects of class party.
id  a vector of terminal node identifiers.
type  a character string denoting the type of predicted value returned, ignored when argument FUN is given. For "response", the mean of a numeric response, the predicted class for a categorical response or the median survival time for a censored response is returned. For "prob" the matrix of conditional class probabilities (simplify = TRUE) or a list with the conditional class probabilities for each
observation (simplify = FALSE) is returned for a categorical response. For numeric and censored responses, a list with the empirical cumulative distribution functions and empirical survivor functions (Kaplan-Meier estimate) is returned when type = "prob". "node" returns an integer vector of terminal node identifiers.

**FUN**

A function to extract (default method) or compute (constparty method) summary statistics. For the default method, this is a function of a terminal node only, for the constparty method, predictions for each node have to be computed based on arguments \((y, w)\) where \(y\) is the response and \(w\) are case weights.

**at**

If the return value is a function (as the empirical cumulative distribution function or the empirical quantile function), this function is evaluated at values at and these numeric values are returned. If at is NULL, the functions themselves are returned in a list.

**simplify**

A logical indicating whether the resulting list of predictions should be converted to a suitable vector or matrix (if possible).

... additional arguments.

**Details**

The `predict` method for `party` objects computes the identifiers of the predicted terminal nodes, either for new data in `newdata` or for the learning samples (only possible for objects of class `constparty`). These identifiers are delegated to the corresponding `predict_party` method which computes (via `FUN` for class `constparty`) or extracts (class `simpleparty`) the actual predictions.

**Value**

A list of predictions, possibly simplified to a numeric vector, numeric matrix or factor.

**Examples**

```r
## fit tree using rpart
library("rpart")
rp <- rpart(skips ~ Opening + Solder + Mask + PadType + Panel,
data = solder, method = 'anova')

## coerce to 'constparty'
pr <- as.party(rp)

## mean predictions
predict(pr, newdata = solder[c(3, 541, 640),])

## ecdf
predict(pr, newdata = solder[c(3, 541, 640),], type = "prob")

## terminal node identifiers
predict(pr, newdata = solder[c(3, 541, 640),], type = "node")

## median predictions
predict(pr, newdata = solder[c(3, 541, 640),],
```

**partynode**

\[\text{FUN} = \text{function}(y, w = 1) \text{median}(y)\]

---

**Description**

A class for representing inner and terminal nodes in trees and functions for data partitioning.

**Usage**

\[
\begin{align*}
\text{partynode}(\text{id}, \text{split} = \text{NULL}, \text{kids} = \text{NULL}, \text{surrogates} = \text{NULL}, \\
\text{info} = \text{NULL})
\end{align*}
\]

\[
\begin{align*}
\text{kidids_node}(\text{node}, \text{data}, \text{vmatch} = 1: \text{ncol}(\text{data}), \\
\text{obs} = \text{NULL}, \text{perm} = \text{NULL})
\end{align*}
\]

\[
\begin{align*}
\text{fitted_node}(\text{node}, \text{data}, \text{vmatch} = 1: \text{ncol}(\text{data}), \\
\text{obs} = 1: \text{nrow}(\text{data}), \text{perm} = \text{NULL})
\end{align*}
\]

\[
\begin{align*}
\text{id_node}(\text{node})
\end{align*}
\]

\[
\begin{align*}
\text{split_node}(\text{node})
\end{align*}
\]

\[
\begin{align*}
\text{surrogates_node}(\text{node})
\end{align*}
\]

\[
\begin{align*}
\text{kids_node}(\text{node})
\end{align*}
\]

\[
\begin{align*}
\text{info_node}(\text{node})
\end{align*}
\]

\[
\begin{align*}
\text{formatinfo_node}(\text{node}, \text{FUN} = \text{NULL}, \text{default} = ",", \text{prefix} = \text{NULL}, \ldots)
\end{align*}
\]

**Arguments**

\[
\begin{align*}
\text{id} & \quad \text{integer, a unique identifier for a node.} \\
\text{split} & \quad \text{an object of class partysplit.} \\
\text{kids} & \quad \text{a list of partynode objects.} \\
\text{surrogates} & \quad \text{a list of partysplit objects.} \\
\text{info} & \quad \text{additional information.} \\
\text{node} & \quad \text{an object of class partynode.} \\
\text{data} & \quad \text{a list or data.frame.} \\
\text{vmatch} & \quad \text{a permutation of the variable numbers in data.} \\
\text{obs} & \quad \text{a logical or integer vector indicating a subset of the observations in data.} \\
\text{perm} & \quad \text{a vector of integers specifying the variables to be permuted prior before splitting (i.e., for computing permutation variable importances). The default NULL doesn’t alter the data.} \\
\text{FUN} & \quad \text{function for formatting the info, for default see below.} \\
\text{default} & \quad \text{a character used if the info in node is NULL.} \\
\text{prefix} & \quad \text{an optional prefix to be added to the returned character.} \\
\ldots & \quad \text{further arguments passed to capture.output.}
\end{align*}
\]
Details

A node represents both inner and terminal nodes in a tree structure. Each node has a unique identifier id. A node consisting only of such an identifier (and possibly additional information in info) is a terminal node.

Inner nodes consist of a primary split (an object of class partysplit) and at least two kids (daughter nodes). Kid nodes are objects of class partynode itself, so the tree structure is defined recursively. In addition, a list of partysplit objects offering surrogate splits can be supplied. Like partysplit objects, partynode objects aren't connected to the actual data.

Function kidids_node() determines how the observations in data[obs,] are partitioned into the kid nodes and returns the number of the list element in list kids each observations belongs to (and not it’s identifier). This is done by evaluating split (and possibly all surrogate splits) on data using kidids_split.

Function fitted_node() performs all splits recursively and returns the identifier id of the terminal node each observation in data[obs,] belongs to. Arguments vmatch, obs and perm are passed to kidids_split.

Function formatinfo_node() extracts the the info from node and formats it to a character vector using the following strategy: If is.null(info), the default is returned. Otherwise, FUN is applied for formatting. The default function uses as.character for atomic objects and applies capture.output to print(info) for other objects. Optionally, a prefix can be added to the computed character string.

All other functions are accessor functions for extracting information from objects of class partynode.

Value

The constructor partynode() returns an object of class partynode:

id a unique integer identifier for a node.

split an object of class partysplit.

kids a list of partynode objects.

surrogates a list of partysplit objects.

info additional information.

kidids_split() returns an integer vector describing the partition of the observations into kid nodes by their position in list kids.

fitted_node() returns the node identifiers (id) of the terminal nodes each observation belongs to.

References


Examples

data("iris", package = "datasets")

## a stump defined by a binary split in Sepal.Length
stump <- partynode(id = 1L,
    split = partysplit(which(names(iris) == "Sepal.Length"),
        breaks = 5),
    kids = lapply(2:3, partynode))

## textual representation
print(stump, data = iris)

## list element number and node id of the two terminal nodes
table(kidids_node(stump, iris),
    fitted_node(stump, data = iris))

## assign terminal nodes with probability 0.5
## to observations with missing 'Sepal.Length'
iris_NA <- iris
iris_NA[sample(1:nrow(iris), 50), "Sepal.Length"] <- NA
table(fitted_node(stump, data = iris_NA,
    obs = !complete.cases(iris_NA)))

## a stump defined by a primary split in 'Sepal.Length'
## and a surrogate split in 'Sepal.Width' which
## determines terminal nodes for observations with
## missing 'Sepal.Length'
stump <- partynode(id = 1L,
    split = partysplit(which(names(iris) == "Sepal.Length"),
        breaks = 5),
    kids = lapply(2:3, partynode),
    surrogates = list(partysplit(
        which(names(iris) == "Sepal.Width"), breaks = 3)))
f <- fitted_node(stump, data = iris_NA,
    obs = !complete.cases(iris_NA))
tapply(iris_NA$Sepal.Width[!complete.cases(iris_NA)], f, range)

partynode-methods

Methods for computing on partynode objects.

Description

Methods for computing on partynode objects.

Usage

is.partynode(x)
as.partynode(x, ...)
## S3 method for class 'partynode'
as.partynode(x, from = NULL, recursive = TRUE, ...)
## S3 method for class 'list'
as.partynode(x, ...)
## S3 method for class 'partynode'
as.list(x, ...)  
## S3 method for class 'partynode'
length(x)  
## S3 method for class 'partynode'
x[i, ...]  
## S3 method for class 'partynode'
x[[i, ...]]
is.terminal(x, ...)  
## S3 method for class 'partynode'
is.terminal(x, ...)
## S3 method for class 'partynode'
depth(x, root = FALSE, ...)
width(x, ...)
## S3 method for class 'partynode'
width(x, ...)
## S3 method for class 'partynode'
print(x, data = NULL, names = NULL, 
inner_panel = function(node) "", 
terminal_panel = function(node) " *", 
prefix = "", first = TRUE, digits = getOption("digits") - 2, 
...)
## S3 method for class 'partynode'
nodeprune(x, ids, ...)

Arguments

x an object of class partynode or list.
from an integer giving the identifier of the root node.
recursive a logical, if FALSE, only the id of the root node is checked against from. If TRUE, the ids of all nodes are checked.
i an integer specifying the kid to extract.
root a logical. Should the root count be counted in depth?
data an optional data.frame.
names a vector of names for nodes.
terminal_panel a panel function for printing terminal nodes.
inner_panel a panel function for printing inner nodes.
prefix lines start with this symbol.
first a logical.
digits number of digits to be printed.
ids a vector of node ids to be pruned-off.
... additional arguments.
Details

`is.partynode` checks if the argument is a valid `partynode` object. `is.terminal` is TRUE for terminal nodes and FALSE for inner nodes. The subset methods return the `partynode` object corresponding to the ith kid.

The `as.partynode` and `as.list` methods can be used to convert flat list structures into recursive `partynode` objects and vice versa. `as.partynode` applied to `partynode` objects renumbers the recursive nodes starting with root node identifier from.

`length` gives the number of kid nodes of the root node, `depth` the depth of the tree and `width` the number of terminal nodes.

Examples

```r
## a tree as flat list structure
nodelist <- list(
  # root node
  list(id = 1L, split = partysplit(varid = 4L, breaks = 1.9),
       kids = 2:3),
  # V4 <= 1.9, terminal node
  list(id = 2L),
  # V4 > 1.9
  list(id = 3L, split = partysplit(varid = 1L, breaks = 1.7),
       kids = c(4L, 7L)),
  # V1 <= 1.7
  list(id = 4L, split = partysplit(varid = 4L, breaks = 4.8),
       kids = 5:6),
  # V4 <= 4.8, terminal node
  list(id = 5L),
  # V4 > 4.8, terminal node
  list(id = 6L),
  # V1 > 1.7, terminal node
  list(id = 7L)
)

## convert to a recursive structure
node <- as.partynode(nodelist)

## print raw recursive structure without data
print(node)

## print tree along with the associated iris data
data("iris", package = "datasets")
print(node, data = iris)

## print subtree
print(node[2], data = iris)

## print subtree, with root node number one
print(as.partynode(node[2], from = 1), data = iris)

## number of kids in root node
length(node)
```
## depth of tree
`depth(node)`

## number of terminal nodes
`width(node)`

## convert back to flat structure
`as.list(node)`

---

### partysplit

**Binary and Multiway Splits**

**Description**

A class for representing multiway splits and functions for computing on splits.

**Usage**

```r
partysplit(varid, breaks = NULL, index = NULL, right = TRUE, prob = NULL, info = NULL)
kidids_split(split, data, vmatch = 1:length(data), obs = NULL)
character_split(split, data = NULL, digits = getOption("digits") - 2)
varid_split(split)
breaks_split(split)
index_split(split)
right_split(split)
prob_split(split)
info_split(split)
```

**Arguments**

- **varid**: an integer specifying the variable to split in, i.e., a column number in `data`.
- **breaks**: a numeric vector of split points.
- **index**: an integer vector containing a contiguous sequence from one to the number of kid nodes. May contain `NA`s.
- **right**: a logical, indicating if the intervals defined by `breaks` should be closed on the right (and open on the left) or vice versa.
- **prob**: a numeric vector representing a probability distribution over kid nodes.
- **info**: additional information.
- **split**: an object of class `partysplit`.
- **data**: a `list` or `data.frame`.
- **vmatch**: a permutation of the variable numbers in `data`.
- **obs**: a logical or integer vector indicating a subset of the observations in `data`.
- **digits**: minimal number of significant digits.
Details

A split is basically a function that maps data, more specifically a partitioning variable, to a set of integers indicating the kid nodes to send observations to. Objects of class partysplit describe such a function and can be set-up via the `partysplit()` constructor. The variables are available in a list or data.frame (here called data) and varid specifies the partitioning variable, i.e., the variable or list element to split in. The constructor `partysplit()` doesn’t have access to the actual data, i.e., doesn’t estimate splits.

`kidids_split(split, data)` actually partitions the data `data[obs, varid_split(split)]` and assigns an integer (giving the kid node number) to each observation. If `vmatch` is given, the variable `vmatch[varid_split(split)]` is used.

`character_split()` returns a character representation of its split argument. The remaining functions defined here are accessor functions for `partysplit` objects.

The numeric vector `breaks` defines how the range of the partitioning variable (after coercing to a numeric via `as.numeric`) is divided into intervals (like in `cut`) and may be `NULL`. These intervals are represented by the numbers one to `length(breaks) + 1`.

`index` assigns these `length(breaks) + 1` intervals to one of at least two kid nodes. Thus, `index` is a vector of integers where each element corresponds to one element in a list `kids` containing `partynode` objects, see `partynode` for details. The vector `index` may contain `NA`s, in that case, the corresponding values of the splitting variable are treated as missings (for example factor levels that are not present in the learning sample). Either `breaks` or `index` must be given. When `breaks` is `NULL`, it is assumed that the partitioning variable itself has storage mode integer (e.g., is a `factor`).

`prob` defines a probability distribution over all kid nodes which is used for random splitting when a deterministic split isn’t possible (due to missing values, for example).

`info` takes arbitrary user-specified information.

Value

The constructor `partysplit()` returns an object of class `partysplit`:

- `varid` an integer specifying the variable to split in, i.e., a column number in `data`,
- `breaks` a numeric vector of split points,
- `index` an integer vector containing a contiguous sequence from one to the number of kid nodes,
- `right` a logical, indicating if the intervals defined by `breaks` should be closed on the right (and open on the left) or vice versa
- `prob` a numeric vector representing a probability distribution over kid nodes,
- `info` additional information.

`kidids_split()` returns an integer vector describing the partition of the observations into kid nodes.

`character_split()` gives a character representation of the split and the remaining functions return the corresponding slots of `partysplit` objects.

References

See Also

cut

Examples

data("iris", package = "datasets")

## binary split in numeric variable 'Sepal.Length'
s15 <- partysplit(which(names(iris) == "Sepal.Length"),
  breaks = 5)
character_split(s15, data = iris)
table(kidids_split(s15, data = iris), iris$Sepal.Length <= 5)

## multiway split in numeric variable 'Sepal.Width',
## higher values go to the first kid, smallest values
to the last kid
sw23 <- partysplit(which(names(iris) == "Sepal.Width"),
  breaks = c(3, 3.5), index = 3:1)
character_split(sw23, data = iris)
table(kidids_split(sw23, data = iris),
  cut(iris$Sepal.Width, breaks = c(-Inf, 2, 3, Inf)))

## binary split in factor 'Species'
sp <- partysplit(which(names(iris) == "Species"),
  index = c(1L, 1L, 2L))
character_split(sp, data = iris)
table(kidids_split(sp, data = iris), iris$Species)

## multiway split in factor 'Species'
sp <- partysplit(which(names(iris) == "Species"), index = 1:3)
character_split(sp, data = iris)
table(kidids_split(sp, data = iris), iris$Species)

## multiway split in numeric variable 'Sepal.Width'
sp <- partysplit(which(names(iris) == "Sepal.Width"),
  breaks = quantile(iris$Sepal.Width))
character_split(sp, data = iris)

---

prune.modelparty  Post-Prune modelparty Objects

Description

Post-pruning of modelparty objects based on information criteria like AIC, BIC, or related user-defined criteria.

Usage

prune.modelparty(tree, type = "AIC", ...)

---
Arguments

- `tree`: object of class `modelparty`.
- `type`: pruning type. Can be "AIC", "BIC" or a user-defined function (details below).
- `...`: additional arguments.

Details

In `mob`-based model trees, pre-pruning based on p-values is used by default and often no post-pruning is necessary in such trees. However, if pre-pruning is switched off (by using a large `alpha`) or does not suffice (e.g., possibly in large samples) the `prune` method can be used for subsequent post-pruning based on information criteria.

The function `prune.modelparty` can be called directly but it is also registered as a method for the generic `prune` function from the `rpart` package. Thus, if `rpart` is attached, `prune(tree, type = "AIC", ...)` also works (see examples below).

To customize the post-pruning strategy, `type` can be set to a function(`objfun, df, nobs`) which either returns `TRUE` to signal that a current node can be pruned or `FALSE`. All supplied arguments are of length two: `objfun` is the sum of objective function values in the current node and its child nodes, respectively. `df` is the degrees of freedom in the current node and its child nodes, respectively. `nobs` is vector with the number of observations in the current node and the total number of observations in the dataset, respectively.

For "AIC" and "BIC" type is transformed so that AIC or BIC are computed. However, this assumes that the `objfun` used in `tree` is actually the negative log-likelihood. The degrees of freedom assumed for a split can be set via the `dfsplit` argument in `mob_control` when computing the tree or manipulated later by changing the value of `tree$info$control$dfsplit`.

Value

An object of class `modelparty` where the associated tree is either the same as the original or smaller.

See Also

- `prune`, `lmtree`, `glmtree`, `mob`

Examples

```r
set.seed(29)
n <- 1000
d <- data.frame(
  x = runif(n),
  z = runif(n),
  z_noise = factor(sample(1:3, size = n, replace = TRUE))
)
d$y <- rnorm(n, mean = d$x * c(-1, 1)[(d$z > 0.7) + 1], sd = 3)

## glm versus lm / logLik versus sum of squared residuals
fmla <- y ~ x | z + z_noise
lm_big <- lmtree(formula = fmla, data = d, maxdepth = 3, alpha = 1)
glm_big <- glmtree(formula = fmla, data = d, maxdepth = 3, alpha = 1)
```
AIC(lm_big)
AIC(glm_big)

## load rpart for prune() generic
## (otherwise: use prune.modelparty directly)
if (require("rpart")) {

## pruning
lm_aic <- prune(lm_big, type = "AIC")
lm_bic <- prune(lm_big, type = "BIC")

width(lm_big)
width(lm_aic)
width(lm_bic)

glm_aic <- prune(glm_big, type = "AIC")
glm_bic <- prune(glm_big, type = "BIC")

width(glm_big)
width(glm_aic)
width(glm_bic)
}

---

### varimp

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
| Standard and conditional variable importance for ‘cforest’, following the permutation principle of the ‘mean decrease in accuracy’ importance in ‘randomForest’.

#### Usage

```r
## S3 method for class 'constparty'
varimp(object, nperm = 1L,
      risk = c("loglik", "misclassification"), conditions = NULL, mincriterion = 0, 
      ...)  

## S3 method for class 'cforest'
varimp(object, nperm = 1L, 
       OOB = TRUE, risk = c("loglik", "misclassification"),
       conditional = FALSE, threshold = .2, applyfun = NULL, 
       cores = NULL, ...)
```

#### Arguments

- `object` an object as returned by `cforest`. 

---

### Variable Importance

---

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
| Standard and conditional variable importance for ‘cforest’, following the permutation principle of the ‘mean decrease in accuracy’ importance in ‘randomForest’.

#### Usage

```r
## S3 method for class 'constparty'
varimp(object, nperm = 1L,
      risk = c("loglik", "misclassification"), conditions = NULL, mincriterion = 0, 
      ...)  

## S3 method for class 'cforest'
varimp(object, nperm = 1L, 
       OOB = TRUE, risk = c("loglik", "misclassification"),
       conditional = FALSE, threshold = .2, applyfun = NULL, 
       cores = NULL, ...)
```

#### Arguments

- `object` an object as returned by `cforest`. 

---
varimp

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 value of the test statistic or 1 - p-value of the association between the variable of interest and a covariate that must be exceeded inorder to include the covariate in the conditioning scheme for the variable of interest (only relevant if conditional = TRUE).

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).

risk a character determining the risk to be evaluated.

conditions a list of conditions.

applyfun an optional lapply-style function with arguments function(X, FUN, ...). It is used for computing the variable importances for each tree. The default is to use the basic lapply function unless the cores argument is specified (see below). Extra care is needed to ensure correct seeds are used in the parallel runs (RNGkind("L'Ecuyer-CMRG") for example).

cores numeric. If set to an integer the applyfun is set to mclapply with the desired number of cores.

... additional arguments, not used.

Details

NEEDS UPDATE

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.

Value

A vector of ‘mean decrease in accuracy’ importance scores.
References


Examples

```r
set.seed(290875)
data("readingSkills", package = "party")
readingSkills.cf <- cforest(score ~ ., data = readingSkills, mtry = 2, ntree = 50)

# standard importance
varimp(readingSkills.cf)

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

---

**WeatherPlay**  
*Weather Conditions and Playing a Game*

Description

Artificial data set concerning the conditions suitable for playing some unspecified game.

Usage

data("WeatherPlay")

Format

A data frame containing 14 observations on 5 variables.

- **outlook** factor.
- **temperature** numeric.
- **humidity** numeric.
- **windy** factor.
- **play** factor.
Source
Table 1.3 in Witten and Frank (2011).

References

See Also
party, partynode, partsplit

Examples
```r
## load weather data
data("WeatherPlay", package = "partykit")
WeatherPlay

## construct simple tree
pn <- partynode(1L,
split = partsplit(1L, index = 1:3),
kids = list(
    partynode(2L,
        split = partsplit(3L, breaks = 75),
        kids = list(
            partynode(3L, info = "yes"),
            partynode(4L, info = "no")),
        partynode(5L, info = "yes"),
        partynode(6L,
        split = partsplit(4L, index = 1:2),
        kids = list(
            partynode(7L, info = "yes"),
            partynode(8L, info = "no"))))))

## couple with data
py <- party(pn, WeatherPlay)

## print/plot/predict
print(py)
plot(py)
predict(py, newdata = WeatherPlay)

## customize printing
print(py,
    terminal_panel = function(node) paste(": play=", info_node(node), sep = ""))
```
Index

* datasets
  HuntingSpiders, 17
  WeatherPlay, 56
* hplot
  panelfunctions, 31
  party-plot, 41
* misc
  ctree_control, 10
  mob_control, 25
* tree
  cforest, 2
  ctree, 7
  extree_data, 13
  extree_fit, 15
  glmtree, 16
  lmtree, 19
  mob, 22
  model_frame_rpart, 27
  nodeapply, 28
  nodeids, 30
  party, 34
  party-coercion, 37
  party-methods, 39
  party-predict, 43
  partynode, 45
  partynode-methods, 47
  partysplit, 50
  varimp, 54
  .party (party-methods), 39
  [.partynode (partynode-methods), 47
  [[.party (party-methods), 39
  [[[.partynode (partynode-methods), 47

as.constparty (party-coercion), 37
as.list.partynode (partynode-methods), 47
as.numeric, 51
as.party (party-coercion), 37
as.partynode (partynode-methods), 47
as.simpleparty (party-coercion), 37
breaks_split (partysplit), 50
capture.output, 45, 46
catL2BB, 27
cd_plot, 34
cforest, 2, 3, 5, 12
character_split (partysplit), 50
coeff, 23
coeff.modelparty (mob), 22
ctree, 5, 7, 12, 13, 34
ctree_control, 3, 5, 8, 10, 12, 55
cut, 51, 52
data.frame, 35, 36, 45, 50
data_party (party), 34
depth.party (party-methods), 39
depth.partynode (partynode-methods), 47
deviance, 23
deviance.modelparty (mob), 22
edge_simple, 43
eedge_simple (panelfunctions), 31
estfun, 23
extree_data, 13, 15
extree_fit, 15
factor, 51
fitted.modelparty (mob), 22
fitted_node (partynode), 45
formatinfo_node, 33
formatinfo_node (partynode), 45
formula.modelparty (mob), 22
GenzBretz, 11
get_paths (nodeids), 30
getCall.modelparty (mob), 22
getCall.party (party-methods), 39
gettree (cforest), 2
glm, 16
glm.control, 16
glm.fit, 16

58
INDEX

glmTree, 16, 20, 24, 53
grid.text, 33

HuntingSpiders, 17

id_node (partynode), 45
importance, 55
index_split (partysplit), 50
info_node (partynode), 45
info_split (partysplit), 50
is.constparty (party), 34
is.partynode (partynode-methods), 47
is.simpleparty (party), 34
is.terminal (partynode-methods), 47

J48, 38

kidids_node (partynode), 45
kidids_split, 46
kidids_split (partysplit), 50
kids_node (partynode), 45

lapply, 4, 12, 27, 55
length.party (party-methods), 39
length.partynode (partynode-methods), 47
list, 45, 50
lm.fit, 19
lm.wfit, 19
lmtree, 17, 19, 24, 53
logLik, 23
logLik.modelparty (mob), 22

mclapply, 4, 12, 27, 55
mob, 16, 17, 19, 20, 22, 25–27, 33, 53
mob_control, 16, 17, 19, 20, 22, 24, 25, 53
model.frame, 16, 19, 22, 28
model.frame.modelparty (mob), 22
model.frame.rpart, 27
modelparty, 16, 20
modelparty (mob), 22

names.party (party), 34
names<-party (party), 34
nobs.modelparty (mob), 22
node_barpplot, 43
node_barpplot (panelfunctions), 31
node_bivplot (panelfunctions), 31
node_boxplot, 43
node_boxplot (panelfunctions), 31
node_ecdf (panelfunctions), 31
node_inner, 42, 43
node_inner (partynode-methods), 31
node_mvar (partynode-methods), 31
node_party (party), 34
node_surv (partynode-methods), 31
node_terminal, 43
node_terminal (partynode-methods), 31
nodeapply, 28
nodeids, 30
nodeprune (party-methods), 39
nodeprune.partynode
(partynode-methods), 47

panelfunctions, 31
party, 8, 9, 24, 28, 30, 34, 38, 39, 43, 44, 57
party-coercion, 37
party-methods, 39
party-plot, 41
party-predict, 43
partynode, 28, 30, 35, 36, 40, 45, 51, 57
partynode-methods, 47
partysplit, 45, 46, 50, 57
plot.constparty (party-plot), 41
plot.glmtree (glmTree), 16
plot.lmtree (lmtree), 19
plot.modelparty (mob), 22
plot.party (party-plot), 41
plot.simpleparty (party-plot), 41
pmmlTreeModel (party-coercion), 37
pmvnorm, 27
predict, 5, 9, 44
predict.cforest (cforest), 2
predict.glmtree (glmTree), 16
predict.lmtree (lmtree), 19
predict.modelparty (mob), 22
predict.party (party-predict), 43
predict_party (party-predict), 43
print.constparty (party-methods), 39
print.glmtree (glmTree), 16
print.lmtree (lmtree), 19
print.modelparty (mob), 22
print.party (party-methods), 39
print.partynode (partynode-methods), 47
print.simpleparty (party-methods), 39
prob_split (partysplit), 50
prune, 53
prune.lmtree (prune.modelparty), 52
prune.modelparty, 52
randomForest, 5
refit.modelparty (mob), 22
residuals.modelparty (mob), 22
right_split (partysplit), 50
rpart, 28, 38
sctest.constparty (ctree), 7
sctest.modelparty (mob), 22
spine, 34
split_node (partynode), 45
summary.modelparty (mob), 22
surrogates_node (partynode), 45
survfit, 33
terms, 35, 36
treeresponse, 5
varid_split (partysplit), 50
varimp, 54
WeatherPlay, 56
weights.modelparty (mob), 22
Weka_tree, 38
where, 5
width (partynode-methods), 47
width.party (party-methods), 39