Package ‘partykit’

September 3, 2013

Title A Toolkit for Recursive Partytioning

Date 2013-09-03

Version 0.1-6

Author Torsten Hothorn [aut, cre], Achim Zeileis [aut]

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

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. (It will also be the basis for a re-implementation of the party package. Currently, only a re-implementation of ctree() is contained in the package.)

Depends R (>= 2.5.0), graphics, stats, grid

Imports survival, RWeka (>= 0.4-19)

Suggests XML, pmml, mlbench, rJava, rpart, mvtnorm, Formula, TH.data

LazyData yes

License GPL-2

NeedsCompilation yes

Repository CRAN

Date/Publication 2013-09-03 20:37:16

R topics documented:

`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, weights, subset, na.action = na.pass,
    control = ctree_control(...), ...)
```

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.
`na.action`	a function which indicates what should happen when the data contain missing value.
`control`	a list with control parameters, see `ctree_control`.
`...`	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. Although the new code was already extensively tested, it is not yet as mature as the old code. If you notice differences in the structure/predictions of the resulting trees, please contact the package maintainers. See also below for some remarks about the internals of the different implementations and how they should be merged in future releases.

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 repeate steps 1) and 2).

The implementation utilizes a unified framework for conditional inference, or permutation tests, developed by Strasser and Weber (1999). The stop criterion in step 1) is either based on multiplicity adjusted p-values (`testtype = "Bonferroni"` 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 and no form of pruning or cross-validation or whatsoever is needed. The selection of the input variable to split in is based on the univariate p-values avoiding a variable selection bias towards input variables with many possible cutpoints.

Predictions can be computed using `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 `newdata = NULL`.

For a general description of the methodology see Hothorn, Hornik and Zeileis (2006) and Hothorn, Hornik, van de Wiel and Zeileis (2006).

Implementation details and roadmap: As pointed above, the function `ctree` is a reimplementation of `ctree`. Not only the R code changed but also the underlying C code which at the moment does not support the `xtrafo` and `ytrafo` arguments due to efficiency considerations. The roadmap for future releases is the following: (1) Make **party** depend on **partykit**. (2) Merge the R code into a single `ctree` function but keeping the two underlying C functions separate. (3) The new interface will always return an object of class `party` but call the new and more efficient C code only if `xtrafo` was not used (while `ytrafo` should be integrated into the new C code).

Value

An object of class `party`.

References

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

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

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

Examples

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

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

### multivariate responses
airq2 <- ctree(Ozone + Temp ~ ., data = airq)
airq2
plot(airq2)
```

ctree_control *Control for Conditional Inference Trees*

Description

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

Usage

```
ctree_control(teststat = c("quad", "max"),
    testtype = c("Bonferroni", "Univariate", "Teststatistic"),
    mincriterion = 0.95, minsplit = 20L, minbucket = 7L,
    minprob = 0.01, stump = FALSE, maxsurrogate = 0L, mtry = Inf,
    maxdepth = Inf, multiway = FALSE, splittry = 2L)
```

Arguments

`teststat` a character specifying the type of the test statistic to be applied.

`testtype` a character specifying how to compute the distribution of the test statistic.

`mincriterion` the value of the test statistic or 1 - p-value that must be exceeded in order to implement a split.

`minsplit` the minimum sum of weights in a node in order to be considered for splitting.

`minbucket` the minimum sum of weights in a terminal node.

`minprob` proportion of observations needed to establish a terminal node.

`stump` a logical determining whether a stump (a tree with three nodes only) is to be computed.

`maxsurrogate` number of surrogate splits to evaluate. Note the currently only surrogate splits in ordered covariables are implemented.

`mtry` number of input variables randomly sampled as candidates at each node for random forest like algorithms. The default `mtry = Inf` means that no random selection takes place.

`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.

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`.

Value

A list.

model.frame.rpart *Model Frame Method for rpart*

Description

A model.frame method for rpart objects.

Usage

```
## S3 method for class 'rpart'
model.frame(formula, ...)
```

Arguments

`formula`	an object of class `rpart`.
`...`	additional arguments.

Details

A `model.frame` method for `rpart` objects.

Value

A model frame.

nodeapply *Apply Functions Over Nodes*

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 it's `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

```
    ## 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 ‘constparty’
    rpk <- as.party(rp)

    ## extract nodeids
    nodeids(rpk)
    unlist(nodeapply(node_party(rpk), ids = nodeids(rpk),
        FUN = id_node))
    unlist(nodeapply(rpk, ids = nodeids(rpk), FUN = id_node))

    ## but root nodes of party objects always have id = 1
    unlist(nodeapply(rpk, ids = nodeids(rpk), 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

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

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.
...	additional arguments.

Details

The identifiers of each node are extracted.

Value

A vector of node identifiers.

Examples

```
## 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)

## set up party object
data("iris")
tree <- party(node, data = iris,
    fitted = data.frame("(fitted)" =
                        fitted_node(node, data = iris),
                        check.names = FALSE))
tree

### ids of all nodes
nodeids(tree)

### ids of all terminal nodes
nodeids(tree, terminal = TRUE)

### ids of terminal nodes in subtree with root [3]
nodeids(tree, from = 3, terminal = TRUE)
```

`panelfunctions` *Panel-Generators 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, abbreviate = FALSE, fill = "white",
    gp = gpar())

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)

edge_simple(obj, digits = 3, abbreviate = FALSE)

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

node_barplot(obj, col = "black", fill = NULL, beside = NULL,
    ymax = NULL, ylines = NULL, widths = 1, gap = NULL,
    reverse = NULL, id = TRUE, gp = gpar())

node_surv(obj, col = "black", ylines = 2, id = TRUE, gp = gpar(), ...)
```

Arguments

`obj` an object of class `party`.

`digits` integer, used for formating numbers.

`abbreviate` logical indicating whether strings should be abbreviated.

`col` a color for points and lines.

`fill` a color to filling rectangles.

`id` logical. Should node IDs be plotted?

`just` justification of terminal panel viewport.

`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.

`width` width in boxplots.

`gap` gap between bars in a barplot (`node_barplot`).

`yscale` limits in y-direction

`ymax` upper limit in y-direction

`cex` 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.

`gp` graphical parameters.

`FUN` function for formatting the `info`, passed to `formatinfo_node`.

`...` 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` and `node_density` can be used to plot bar plots, box plots, histograms and estimated densities into the terminal nodes.

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

References

David Meyer, Achim Zeileis, and Kurt Hornik (2006). The Strucplot Framework: Visualizing Multi-Way Contingency Tables with vcd. *Journal of Statistical Software*, **17**(3). `http://www.jstatsoft.org/v17/i03/`

`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
data_party(party, id = 1L)
## 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`.

Examples

```
data("iris")

## 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))

party(stump, iris,
    fitted = data.frame("(fitted)" = fitted_node(stump, data = iris),
        check.names = FALSE), names = c("root", "left", "right"))
```

`party-coercion` *Coercion Functions*

Description

Functions coercing various objects to objects of class party.

Usage

```
as.party(obj, ...)
## S3 method for class 'rpart'
as.party(obj, ...)
## S3 method for class 'J48'
as.party(obj, ...)
## 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`, `J48`, `XMLnode` or objects inheriting from `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

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

    ## coerce to `constparty'
    as.party(rp)
```

party-methods *Methods for Party Objects*

Description

Methods for computing on party objects.

Usage

```
## 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, ...)
```

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`?

`...` additional arguments.

Details

`length` gives the number of nodes in the tree (in contrast to the `length` method for `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 `print` method generates a textual representation of the tree.

Examples

```
## 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,
    fitted = data.frame("(fitted)" =
        fitted_node(node, data = iris),
        check.names = FALSE))
names(tree) <- paste("Node", nodeids(tree), sep = " ")

## number of kids in root node
```

```
length(tree)

## depth of tree
depth(tree)

## number of terminal nodes
width(tree)

## node number four
tree["Node 4"]
tree[["Node 4"]]
```

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(), ...)
## 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.

inner_panel an optional panel function of the form function(node) plotting the inner nodes. Alternatively, a panel generating function of class "grapcon_generator" that is called with arguments x and ip_args to set up a panel function.

ip_args a list of arguments passed to inner_panel if this is a "grapcon_generator" object.

edge_panel an optional panel function of the form function(split, ordered = FALSE, left = TRUE) plotting the edges. Alternatively, a panel generating function of class "grapcon_generator" that is called with arguments x and ip_args to set up a panel function.

ep_args a list of arguments passed to edge_panel if this is a "grapcon_generator" object.

drop_terminal a logical indicating whether all terminal nodes should be plotted at the bottom.

tnex a numeric value giving the terminal node extension in relation to the inner nodes.

newpage a logical indicating whether grid.newpage() should be called.

pop a logical whether the viewport tree should be popped before return.

gp graphical parameters.

digits number of digits to be printed.

... additional arguments passed to callies.

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).

References

David Meyer, Achim Zeileis, and Kurt Hornik (2006). The Strucplot Framework: Visualizing Multi-Way Contingency Tables with vcd. *Journal of Statistical Software*, **17**(3). http://www.jstatsoft.org/v17/i03/

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, ...)
predict_party(party, id, newdata = NULL, ...)
## Default S3 method:
predict_party(party, id, newdata = NULL, ...)
## S3 method for class 'constparty'
predict_party(party, id, newdata = NULL,
    type = c("response", "prob", "node"), 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.

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 a categorical response, "prob" returns the matrix of conditional class probabilities (simplify = TRUE) or a list with the conditional class probabilities for each observation (simplify = FALSE). "node" returns an integer vector of terminal node identifiers.

FUN — a function to compute summary statistics, i.e., constant predictions for each node with argument list (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).

... — 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

```
## 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),])

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

## median predictions
predict(pr, newdata = solder[c(3, 541, 640),],
        FUN = function(y, w = 1) median(y))
```

partynode *Inner and Terminal Nodes*

Description

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

Usage

```
partynode(id, split = NULL, kids = NULL, surrogates = NULL,
    info = NULL)
kidids_node(node, data, vmatch = 1:ncol(data),
    obs = NULL, perm = NULL)
fitted_node(node, data, vmatch = 1:ncol(data),
    obs = 1:nrow(data), perm = NULL)
id_node(node)
```

```
split_node(node)
surrogates_node(node)
kids_node(node)
info_node(node)
formatinfo_node(node, FUN = NULL, default = "", prefix = NULL, ...)
```

Arguments

`id` integer, a unique 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.

`node` an object of class `partynode`.

`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`.

`perm` 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.

`FUN` function for formatting the `info`, for default see below.

`default` a character used if the `info` in `node` is `NULL`.

`prefix` an optional prefix to be added to the returned character.

`...` further arguments passed to `capture.output`.

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.

Examples

```
data("iris")

## 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 Node 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, ...)
## 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, ...)
depth(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,
    ...)
```

Arguments

x	an object of class partynode or list.
from	an integer giving the identifier of the root node.
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.
...	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

```
## 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 tree
data("iris")
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

```
partysplit(varid, breaks = NULL, index = NULL, right = TRUE,
    prob = NULL, info = NULL)
kidids_split(split, data, vmatch = 1:ncol(data), obs = NULL,
    perm = 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 NAs.

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.

perm 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.

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. In case perm contains varid_split(split), the data are permuted using sample prior to partitioning.

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 NAs, 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.

See Also

`cut`

Examples

```
data("iris")

## binary split in numeric variable ‘Sepal.Length’
sl5 <- partysplit(which(names(iris) == "Sepal.Length"),
    breaks = 5)
character_split(sl5, data = iris)
table(kidids_split(sl5, 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$Sepal.Width))
character_split(sp, data = iris)
## predictions for permuted values of ‘Sepal.Width’
## correlation with actual data should be small
cor(kidids_split(sp, data = iris,
    perm = which(names(iris) == "Sepal.Width")),
    iris$Sepal.Width)
```

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

I. H. Witten and E. Frank (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd Edition, Morgan Kaufmann, San Francisco.

See Also

party, partynode, partysplit

Examples

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

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

## 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