Package 'lorax'

July 16, 2026

Title Speak for the Trees

Version 0.1.0

Description Extracts decision rules from tree- and rule-based models fitted in 'R'. Rules are expressed as logical predicates that identify paths to terminal nodes, making model behavior more transparent and interpretable. Provides conversion methods to 'partykit' party objects for a wide range of model types. The 'partykit' infrastructure is described in Hothorn and Zeileis (2015) <https://jmlr.org/papers/v16/hothorn15a.html>.

License MIT + file LICENSE

URL https://github.com/tidymodels/lorax

BugReports https://github.com/tidymodels/lorax/issues

Depends R (>= 4.3.0)

Imports cli, dplyr, generics, partykit, purrr, rlang (>= 1.1.0), tibble

Suggests aorsf, C50, Cubist, dbarts, grf, knitr, lightgbm, modeldata, palmerpenguins, randomForest, ranger, rpart, spelling, testthat (>= 3.0.0), tidyr, xgboost

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Encoding UTF-8

Language en-US

RoxygenNote 8.0.0

NeedsCompilation no

Author Max Kuhn [aut, cre] (ORCID: <https://orcid.org/0000-0003-2402-136X>), Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Max Kuhn <max@posit.co>

Repository CRAN

Date/Publication 2026-07-16 12:50:25 UTC

Contents

`active_predictors.C5.0`

Extract the active features from a tree

Description

If a tree does not use a predictor in the training set in any of its splits it is functionally independent of the prediction function. This generic returns a data frame containing character vector of predictor names that were used in at least one split.

Usage

```
## S3 method for class 'C5.0'
active_predictors(x, tree = 1L, ...)

## S3 method for class 'ObliqueForest'
active_predictors(x, tree = 1L, ...)

## S3 method for class 'bart'
active_predictors(x, tree = 1L, ...)

## S3 method for class 'cforest'
```

```
active_predictors(x, tree = 1L, ...)

## S3 method for class 'cubist'
active_predictors(x, ...)

active_predictors(x, ...)

## S3 method for class 'grf'
active_predictors(x, tree = 1L, ...)

## S3 method for class 'lgb.Booster'
active_predictors(x, tree = 1L, ...)

## S3 method for class 'party'
active_predictors(x, ...)

## S3 method for class 'randomForest'
active_predictors(x, tree = 1L, ...)

## S3 method for class 'ranger'
active_predictors(x, tree = 1L, ...)

## S3 method for class 'rpart'
active_predictors(x, ...)

## S3 method for class 'xgb.Booster'
active_predictors(x, tree = 1L, nthread = NULL, ...)
```

Arguments

x	A object
tree	Integer vector specifying which trees to extract active predictors from. Default is 1L for the first tree. Values must be between 1 and the number of trees in the forest.
...	Other arguments passed to methods
nthread	Integer number of threads to use when reading the tree structure out of an **xgboost** model. The default (NULL) inherits the nthread the booster was trained with.

Value

A tibble with list column active_predictors containing a character vector of predictors.

Examples

```
if (rlang::is_installed(c("rpart", "palmerpenguins"))) {
  data(penguins, package = "palmerpenguins")
  penguins <- na.omit(penguins)
```

```
  # Fit a tree
  tree <- rpart::rpart(species ~ ., data = penguins)
  tree

  # Extract active predictors
  active_predictors(tree)

  # Only primary splits are included - competing and surrogate splits
  # are excluded since they don't affect predictions
}

# C5.0 single tree
if (rlang::is_installed(c("C50", "palmerpenguins"))) {
  data(penguins, package = "palmerpenguins")
  penguins <- na.omit(penguins)

  # Tree-based model
  c5_tree <- C50::C5.0(species ~ ., data = penguins)
  active_predictors(c5_tree)

  # Boosted model - extract from multiple trials
  c5_boost <- C50::C5.0(species ~ ., data = penguins, trials = 5)
  active_predictors(c5_boost, tree = 1:3)

  # Rule-based model
  c5_rules <- C50::C5.0(species ~ ., data = penguins, rules = TRUE)
  active_predictors(c5_rules)
}
```

`as.party.bart` *Convert BART model to party object*

Description

Convert a single tree from a BART (Bayesian Additive Regression Trees) model to a party object for use with **partykit** visualization and analysis tools.

Usage

```
## S3 method for class 'bart'
as.party(obj, tree = 1L, chain = 1L, data, ...)
```

Arguments

`obj`	A `bart` object from the **dbarts** package fitted with `keeptrees = TRUE`.
`tree`	Integer specifying which tree to convert (1-based indexing, default is 1). BART models contain `n.trees` trees in the ensemble.

`chain`	Integer specifying which MCMC chain to extract from (1-based indexing, default is 1). Only relevant for models fitted with multiple chains.
`data`	data.frame containing the **original untransformed** training data with original response values (required). BART internally transforms data (creating dummy variables for factors and converting responses to 0/1). You must provide the original data frame that includes both the predictor variables and the response variable in their original formats (e.g., factors for classification).
`...`	Not currently used.

Details

Important note on data transformation:

BART internally transforms the training data in ways that make it unsuitable for display in party objects. Specifically, BART creates dummy variables for factor predictors and converts factor responses to 0/1 numeric values. To get correct terminal node statistics, bar charts, and other visualizations, you **must** provide the original untransformed data (including the response variable) via the `data` parameter.

BART tree storage format:

The **dbarts** package stores trees in depth-first traversal order in a data.frame accessible via `obj$fit$getTrees()`. Each row represents one node:

- `var`: 1-based variable index for split, or -1 for terminal nodes
- `value`: threshold for internal nodes, prediction for terminal nodes
- `tree`: 1-based tree number
- `chain`: chain number (if multiple chains)
- `sample`: MCMC sample number

Depth-first traversal order:

- Nodes stored as: parent, left subtree (complete), right subtree (complete)
- Example: root at row 1, left child at row 2, right child after left subtree
- Must track row consumption to determine subtree boundaries

Node indexing:

- User-facing `tree` and `chain` parameters use 1-based indexing (R convention)
- Variable indices in `var` column are 1-based (match `obj$varNames`)
- Value -1 in `var` indicates terminal node

Split encoding:

- Left child: feature < threshold
- Right child: feature >= threshold
- partykit split created with `right = TRUE` (right interval closed)

Variable names:

- Available in `obj$fit$data@x` column names or `obj$varNames`
- `var` column provides 1-based index into these names

The party object will use 1-based node IDs and variable indices as required by **partykit**.

Value

A constparty object from the **partykit** package.

Examples

```
if (rlang::is_installed(c("dbarts", "palmerpenguins"))) {
  # Classification example
  data(penguins, package = "palmerpenguins")
  penguins <- na.omit(penguins)

  # Prepare data with response column
  train_data <- penguins[, c("bill_length_mm", "bill_depth_mm",
                             "flipper_length_mm", "body_mass_g", "species")]

  set.seed(2847)
  fit <- dbarts::bart(
    x.train = train_data[, 1:4],
    y.train = train_data$species,
    keeptrees = TRUE,
    verbose = FALSE,
    ntree = 2
  )

  # Convert first tree - data parameter is required
  # Response will be preserved in original format (e.g., factor for
  # classification)
  party_tree <- as.party(fit, tree = 1L, chain = 1L, data = train_data)
  print(party_tree)
  plot(party_tree)

  # Regression example
  data(mtcars)
  set.seed(5193)
  fit_reg <- dbarts::bart(
    x.train = mtcars[, -1],
    y.train = mtcars$mpg,
    keeptrees = TRUE,
    verbose = FALSE,
    ntree = 2
  )
  party_tree_reg <- as.party(fit_reg, tree = 1L, chain = 1L, data = mtcars)
  print(party_tree_reg)
}
```

---

as.party.C5.0 *Convert C5.0 model to party object*

Description

Convert a single tree from a C5.0 decision tree or boosted model to a party object for use with partykit visualization and analysis tools.

Usage

```
## S3 method for class 'C5.0'
as.party(obj, tree = 1L, data = NULL, ...)
```

Arguments

`obj`	A `C5.0` object from the **C50** package.
`tree`	Integer specifying which tree to convert (1-based indexing, default is 1). For single tree models, use `tree = 1`. For boosted models with `trials > 1`, this selects which boosting iteration to extract.
`data`	Data.frame containing the training data, including both predictors and response variable. Required for proper party object creation with fitted values and node summaries.
`...`	Not currently used.

Details

C5.0 tree storage format:

The **C50** package stores trees in a custom text format in `obj$tree`. This format uses indented lines with key-value pairs:

- `type="2"`: Internal node with split
- `type="0"`: Terminal/leaf node
- `att="VariableName"`: Attribute/variable to split on
- `forks="n"`: Number of branches (2+ for numeric, can be 4+ for categorical)
- `cut="threshold"`: Numeric threshold for split
- `class="ClassName"`: Predicted class
- `freq="n1,n2,n3"`: Frequency of each class at node

Boosting and trials:

- Single tree models (`trials = 1`): Only `tree = 1` is valid
- Boosted models (`trials > 1`): Multiple sequential trees available
- The `tree` parameter maps to trial/iteration number
- Each boosting trial produces one tree

Tree structure:

- Trees stored as sequential lines in pre-order (parent, then children)
- No indentation used - hierarchy determined by fork counts
- Numeric ternary splits: <= threshold, missing, > threshold
- Categorical multiway splits: one branch per level group

Split encoding:

- Numeric splits: typically binary (<=, >) or ternary (<=, missing, >)
- Ternary numeric splits are simplified to binary by omitting the missing branch
- Categorical splits: can have 2+ branches, one for each level group
- Multiway categorical splits are preserved in the party object

Variable names:

- `obj$predictors` contains ordered list of predictor variable names
- `att` attribute in tree text references these by name
- Map to 1-based indices for **partykit**

Important limitations:

- **C50**'s text format is complex and may vary by version
- Ternary numeric splits simplified to binary (missing value branch omitted)
- Rule-based models (`obj$rules != ""`) not supported
- Some terminal nodes may have n=0 (empty branches where no observations fall)

Value

A `party` object from the **partykit** package.

Examples

```
if (rlang::is_installed(c("C50", "palmerpenguins"))) {
  data(penguins, package = "palmerpenguins")
  penguins <- na.omit(penguins)

  # Single tree model
  set.seed(2847)
  c5_tree <- C50::C5.0(species ~ ., data = penguins)
  party_tree <- as.party(c5_tree, tree = 1L, data = penguins)
  print(party_tree)
  plot(party_tree)

  # Boosted model with multiple trials
  set.seed(5193)
  c5_boost <- C50::C5.0(species ~ ., data = penguins, trials = 3)
  # Extract first boosting iteration
  party_tree1 <- as.party(c5_boost, tree = 1L, data = penguins)
  # Extract third boosting iteration
  party_tree3 <- as.party(c5_boost, tree = 3L, data = penguins)
}
```

`as.party.grf` *Convert grf model to party object*

Description

Convert a single tree from a **grf** (generalized random forests) model to a party object for use with **partykit** visualization and analysis tools.

Usage

```
## S3 method for class 'grf'
as.party(obj, tree = 1L, data = NULL, ...)
```

Arguments

`obj`	A grf object (e.g., `regression_forest`, `causal_forest`) from the grf package.
`tree`	Integer specifying which tree to convert (1-based indexing, default is 1). Must be between 1 and the number of trees in the forest.
`data`	Optional data.frame containing the training data. If NULL, will attempt to extract from the grf object (`obj$X.orig`), or create a placeholder data.frame. Providing data enables full party functionality.
`...`	Not currently used.

Details

GRF tree storage format:

The **grf** package stores trees in a nested list structure, typically accessed via `grf::get_tree(obj, tree)`. Each tree is represented as nested lists:

- `is_leaf`: Logical, TRUE for terminal nodes
- `split_variable`: 0-based index of variable to split on (internal nodes)
- `split_value`: Numeric threshold for split (internal nodes)
- `left_child`: Nested list for left subtree (internal nodes)
- `right_child`: Nested list for right subtree (internal nodes)
- Leaf nodes contain prediction information

Node indexing:

- Internally, **grf** uses 0-based variable indices
- User-facing `tree` parameter uses 1-based indexing (R convention)
- Trees use 0-based indexing internally but we access with 1-based tree number

Split encoding:

- For numeric variables: left child when feature < threshold, right child when feature >= threshold
- **partykit** split created with `right = TRUE` (right interval closed)

Tree structure:

- **grf** provides nested list structure (not flattened)
- This is the most direct representation for recursive conversion
- Each node is a list with `is_leaf` flag and split info

The party object will use 1-based node IDs and variable indices as required by **partykit**.

Value

A `party` object from the **partykit** package.

Examples

```
if (rlang::is_installed(c("grf", "palmerpenguins"))) {
  data(penguins, package = "palmerpenguins")
  penguins <- na.omit(penguins)

  # Regression forest
  set.seed(2847)
  rf <- grf::regression_forest(
    X = penguins[, c("bill_length_mm", "bill_depth_mm",
                     "flipper_length_mm", "body_mass_g")],
    Y = penguins$bill_length_mm,
    num.trees = 3,
    num.threads = 1
  )

  # Convert first tree
  party_tree <- as.party(rf, tree = 1L, data = penguins)
  print(party_tree)
  plot(party_tree)

  # Can also work with other grf forest types
  set.seed(5193)
  cf <- grf::causal_forest(
    X = penguins[, c("bill_length_mm", "bill_depth_mm",
                     "flipper_length_mm", "body_mass_g")],
    Y = penguins$bill_length_mm,
    W = rbinom(nrow(penguins), 1, 0.5),
    num.trees = 3,
    num.threads = 1
  )
  party_tree2 <- as.party(cf, tree = 1L, data = penguins)
}
```

`as.party.lgb.Booster` *Convert lgb.Booster model to party object*

Description

Convert a single tree from a **lightgbm** boosted tree model to a party object for use with **partykit** visualization and analysis tools.

Usage

```
## S3 method for class 'lgb.Booster'
as.party(obj, tree = 1L, data, ...)
```

Arguments

`obj`	An `lgb.Booster` object from the **lightgbm** package.
`tree`	Integer specifying which tree to convert (1-based indexing, default is 1). For multiclass models with `num_class` classes and `nrounds` boosting rounds, there are `num_class * nrounds` total trees.
`data`	data.frame containing the training data **with the response variable included** (required). LightGBM models do not store the original training data or response values. You must provide the original data frame that includes both the predictor variables and the response variable.
`...`	Not currently used.

Details

Important note on data:

lightgbm models do not store the original training data or response values. You **must** provide the original data frame (including the response variable) via the `data` parameter for correct terminal node statistics, bar charts, and other visualizations.

LightGBM tree storage format:

lightgbm stores trees in a tabular format accessible via `lightgbm::lgb.model.dt.tree()`. Each tree is represented as rows in a table:

- `tree_index`: 0-based tree index
- `split_index`: 0-based node ID for internal nodes (NA for leaves)
- `leaf_index`: 0-based node ID for leaf nodes (NA for internal)
- `split_feature`: Feature name (character) for splits
- `threshold`: Numeric threshold for splits
- `decision_type`: Split type ("<=", "==", etc.)
- `left_child`: 0-based node ID of left child
- `right_child`: 0-based node ID of right child
- `leaf_value`: Prediction value for leaf nodes
- `node_parent`: 0-based parent node ID
- `depth`: Depth of node in tree

Node indexing:

- Internally, **lightgbm** uses 0-based tree and node indices

- User-facing `tree` parameter uses 1-based indexing (R convention)
- When `tree=1` is requested, we filter to `tree_index==0` internally
- Internal nodes use `split_index`, leaf nodes use `leaf_index`

Split encoding:

- `decision_type "<="`: left child when feature <= threshold
- right child when feature > threshold
- **partykit** split created with `right = FALSE` (left interval closed)

Child node references:

- Internal nodes have explicit `left_child` and `right_child` IDs
- These reference either `split_index` (internal) or `leaf_index` (leaf)
- Need to look up child in appropriate column based on node type

Variable names:

- `split_feature` column contains actual feature names or `"Column_N"` defaults
- Must map to column positions in data.frame

The party object will use 1-based node IDs and variable indices as required by **partykit**.

Value

A `constparty` object from the **partykit** package.

Examples

```
if (rlang::is_installed("lightgbm")) {
  # Binary classification example
  data(agaricus.train, package = "lightgbm")

  # Prepare data with response column
  train_data <- as.data.frame(as.matrix(agaricus.train$data))
  train_data$label <- agaricus.train$label

  dtrain <- lightgbm::lgb.Dataset(
    agaricus.train$data,
    label = agaricus.train$label
  )

  set.seed(7264)
  bst <- lightgbm::lgb.train(
    params = list(objective = "binary", max_depth = 3, num_threads = 1L),
    data = dtrain,
    nrounds = 3,
    verbose = -1
  )

  # Convert first tree - data parameter is required
  party_tree <- as.party(bst, tree = 1L, data = train_data)
  print(party_tree)
```

```
  plot(party_tree)

  # Regression example
  data(mtcars)
  reg_data <- mtcars
  dtrain_reg <- lightgbm::lgb.Dataset(as.matrix(mtcars[, -1]), label = mtcars$mpg)

  set.seed(6381)
  bst_reg <- lightgbm::lgb.train(
    params = list(
      objective = "regression", max_depth = 3, min_data_in_leaf = 1,
      num_threads = 1L
    ),
    data = dtrain_reg,
    nrounds = 3,
    verbose = -1
  )

  party_tree_reg <- as.party(bst_reg, tree = 1L, data = reg_data)
  print(party_tree_reg)
}
```

`as.party.randomForest` *Convert randomForest model to party object*

Description

Convert a single tree from a **randomForest** model to a party object for use with **partykit** visualization and analysis tools.

Usage

```
## S3 method for class 'randomForest'
as.party(obj, tree = 1L, data = NULL, ...)
```

Arguments

`obj`	A `randomForest` object from the **randomForest** package.
`tree`	Integer specifying which tree to convert (1-based indexing, default is 1). Must be between 1 and the number of trees in the forest.
`data`	Optional data.frame containing the training data. If NULL, a placeholder data.frame will be created with correct variable names but no observations. Providing data enables full party functionality including predictions.
`...`	Not currently used.

Details

randomForest tree storage format:

The **randomForest** package stores trees in `obj$forest` as parallel matrices:

- `leftDaughter[i, tree]`: 1-based row index of left child (0 = no child)
- `rightDaughter[i, tree]`: 1-based row index of right child (0 = no child)
- `bestvar[i, tree]`: 1-based variable index for split (0 for terminal)
- `xbestsplit[i, tree]`: threshold value for split
- `nodestatus[i, tree]`: node status (-1 = terminal, -3 = internal)
- `nodepred[i, tree]`: prediction at node (for regression) or class (classification)

Node indexing:

- **randomForest** uses 1-based row indices for nodes (root is row 1)
- Value 0 in leftDaughter/rightDaughter indicates no child
- User-facing `tree` parameter uses 1-based indexing (R convention)

Split encoding:

- For numeric variables: left child when feature <= threshold, right child when feature > threshold
- Note: **randomForest** uses <= for left (different from ranger's <)
- partykit split created with `right = FALSE` to match this

Terminal node identification:

- `nodestatus == -1` indicates terminal node
- Alternatively: `bestvar == 0` or both `daughters == 0`

The party object will use 1-based node IDs and variable indices as required by partykit.

Value

A `party` object from the **partykit** package.

Examples

```
if (rlang::is_installed(c("randomForest", "palmerpenguins"))) {
  # Classification example
  data(penguins, package = "palmerpenguins")
  penguins <- na.omit(penguins)

  set.seed(2847)
  rf <- randomForest::randomForest(species ~ ., data = penguins, ntree = 3)

  # Convert first tree
  party_tree <- as.party(rf, tree = 1L, data = penguins)
  print(party_tree)
  plot(party_tree)

  # Predictions from party object
  predict(party_tree, newdata = penguins[1:5, ])
```

```
  # Regression example
  data(mtcars)
  set.seed(5193)
  rf_reg <- randomForest::randomForest(mpg ~ ., data = mtcars, ntree = 3)
  party_tree_reg <- as.party(rf_reg, tree = 1L, data = mtcars)
  print(party_tree_reg)
}
```

`as.party.ranger` *Convert ranger model to party object*

Description

Convert a single tree from a **ranger** random forest model to a party object for use with **partykit** visualization and analysis tools.

Usage

```
## S3 method for class 'ranger'
as.party(obj, tree = 1L, data = NULL, ...)
```

Arguments

`obj`	A `ranger` object from the **ranger** package.
`tree`	Integer specifying which tree to convert (1-based indexing, default is 1). Must be between 1 and the number of trees in the forest.
`data`	Data.frame containing the training data, including both predictors and response variable. Required for proper party object creation with fitted values and node summaries.
`...`	Not currently used.

Details

Ranger tree storage format:

The **ranger** package stores trees in `obj$forest` with parallel vectors:

- `split.varIDs[[tree]]`: 0-based variable indices for splits
- `split.values[[tree]]`: threshold values for splits
- `child.nodeIDs[[tree]]`: matrix with 2 columns (left, right child IDs)
- `is.ordered[[tree]]`: whether split variable is ordered (for categoricals)
- All node IDs are 0-based (root = 0)

Node indexing:

- Internally, **ranger** uses 0-based node indices (root is node 0)
- User-facing `tree` parameter uses 1-based indexing (R convention)

- Leaf nodes have `split.varIDs` entry of `NA` or large sentinel value

Split encoding:

- For numeric variables: left child when feature < threshold, right child when feature >= threshold
- **partykit** split created with `right = TRUE` (right interval closed)

Child node references:

- `child.nodeIDs` is a matrix with 2 columns: `left_child`, `right_child`
- Value 0 indicates no child (terminal node)
- Both children 0 means current node is terminal

The party object will use 1-based node IDs and variable indices as required by **partykit**.

Value

A `party` object from the **partykit** package.

Examples

```
if (rlang::is_installed(c("ranger", "palmerpenguins"))) {
  # Classification example
  data(penguins, package = "palmerpenguins")
  penguins <- na.omit(penguins)

  set.seed(2847)
  rf <- ranger::ranger(
    species ~ ., data = penguins, num.trees = 3, num.threads = 1
  )

  # Convert first tree
  party_tree <- as.party(rf, tree = 1L, data = penguins)
  print(party_tree)
  plot(party_tree)

  # Predictions from party object
  predict(party_tree, newdata = penguins[1:5, ])

  # Regression example
  data(mtcars)
  set.seed(5193)
  rf_reg <- ranger::ranger(
    mpg ~ ., data = mtcars, num.trees = 3, num.threads = 1
  )
  party_tree_reg <- as.party(rf_reg, tree = 1L, data = mtcars)
  print(party_tree_reg)
}
```

`as.party.xgb.Booster` *Convert xgb.Booster model to party object*

Description

Convert a single tree from an **xgboost** boosted tree model to a party object for use with **partykit** visualization and analysis tools.

Usage

```
## S3 method for class 'xgb.Booster'
as.party(obj, tree = 1L, data, nthread = NULL, ...)
```

Arguments

`obj`	An `xgb.Booster` object from the **xgboost** package.
`tree`	Integer specifying which tree to convert (1-based indexing, default is 1). For multiclass models with `num_class` classes and `nrounds` boosting rounds, there are `num_class * nrounds` total trees.
`data`	data.frame containing the training data **with the response variable included** (required). XGBoost models do not store the original training data or response values. You must provide the original data frame that includes both the predictor variables and the response variable.
`nthread`	Integer number of threads to use when reading the tree structure out of the model. The default (`NULL`) inherits the `nthread` the booster was trained with.
`...`	Not currently used.

Details

Important note on data:

XGBoost models do not store the original training data or response values. You **must** provide the original data frame (including the response variable) via the `data` parameter for correct terminal node statistics, bar charts, and other visualizations.

XGBoost tree storage format:

xgboost stores trees in a tabular format accessible via `xgboost::xgb.model.dt.tree()`. Each tree is represented as rows in a table:

- `Tree`: 0-based tree index (e.g., 0, 1, 2, ...)
- `Node`: 0-based node ID within tree (e.g., "0-0", "0-1" for tree 0)
- `Feature`: Feature name (character) or "Leaf" for terminal nodes
- `Split`: Numeric threshold for splits (NA for leaves)
- `Yes`: 0-based node ID of yes branch (feature < threshold)
- `No`: 0-based node ID of no branch (feature >= threshold)
- `Missing`: 0-based node ID for missing values

- `Quality`: Prediction value for leaf nodes, gain for internal nodes

Node indexing:

- Internally, **xgboost** uses 0-based tree and node indices
- User-facing `tree` parameter uses 1-based indexing (R convention)
- When `tree=1` is requested, we filter to `Tree==0` internally

Split encoding:

- Yes branch: feature < threshold (left child)
- No branch: feature >= threshold (right child)
- **partykit** split created with `right = TRUE` (right interval closed)

Child node references:

- `Yes` column: node ID for left child (< condition)
- `No` column: node ID for right child (>= condition)
- Leaf nodes have `Feature == "Leaf"`

Variable names:

- `Feature` column contains actual feature names (not indices)
- Must map to column positions in data.frame
- If numeric indices used (`f0`, `f1`, ...), map to data columns

The party object will use 1-based node IDs and variable indices as required by **partykit**.

Value

A `constparty` object from the **partykit** package.

Examples

```
if (rlang::is_installed("xgboost")) {
  data(agaricus.train, package = "xgboost")

  # Binary classification example, on a small subset for a fast example.
  rows <- seq_len(200)
  train_data <- as.data.frame(as.matrix(agaricus.train$data[rows, ]))
  train_data$label <- agaricus.train$label[rows]

  dtrain <- xgboost::xgb.DMatrix(
    agaricus.train$data[rows, ],
    label = agaricus.train$label[rows],
    nthread = 1
  )

  set.seed(3691)
  bst <- xgboost::xgb.train(
    data = dtrain,
    nrounds = 3,
    verbose = 0,
    params = xgboost::xgb.params(
```

```
    max_depth = 3,
    objective = "binary:logistic",
    nthread = 1
  )
)

# Convert first tree - data parameter is required
party_tree <- as.party(bst, tree = 1L, data = train_data)
print(party_tree)
plot(party_tree)

# Regression example
data(mtcars)
reg_data <- mtcars
dtrain_reg <- xgboost::xgb.DMatrix(
  as.matrix(mtcars[, -1]),
  label = mtcars$mpg,
  nthread = 1
)

set.seed(9158)
bst_reg <- xgboost::xgb.train(
  data = dtrain_reg,
  nrounds = 3,
  verbose = 0,
  params = xgboost::xgb.params(
    max_depth = 3,
    objective = "reg:squarederror",
    nthread = 1
  )
)

party_tree_reg <- as.party(bst_reg, tree = 1L, data = reg_data)
print(party_tree_reg)
}
```

---

combine_rule_elements *Combine multiple R expressions into a single composite expression*

Description

This function takes a list of R expressions and combines them using a logical operator to create a single composite expression. It is useful for building complete rule paths by combining individual split conditions from tree-based models.

Usage

```
combine_rule_elements(exprs, operator = "&")
```

Arguments

exprs	A list of R expressions to combine. Each element must be a language object (expression or symbol). The list can be empty (returns TRUE), contain a single expression (returns unchanged), or multiple expressions (combines with operator).
operator	A character string specifying the logical operator to use: • "&" (default): combines expressions with AND logic. • "|": combines expressions with OR logic.

Value

An R expression object that combines all input expressions. Returns TRUE for empty list, the single expression for length-1 list, or a nested expression for multiple elements.

Examples

```
# Basic AND combination
expr1 <- rlang::expr(x > 5)
expr2 <- rlang::expr(y < 10)
combine_rule_elements(list(expr1, expr2))

# OR operator
combine_rule_elements(list(expr1, expr2), operator = "|")

# Integration with rect_split_to_expr()
split1 <- list(column = "age", value = 30, operator = ">=")
split2 <- list(column = "income", value = 50000, operator = ">")
exprs <- list(
  rect_split_to_expr(split1),
  rect_split_to_expr(split2)
)
rule <- combine_rule_elements(exprs)

# Evaluate with data
test_data <- data.frame(age = 35, income = 60000)
eval(rule, test_data)

# Edge cases
combine_rule_elements(list()) # returns TRUE
combine_rule_elements(list(rlang::expr(x > 0))) # returns x > 0
```

---

extract_rules.bart *Extract rules from a BART model*

Description

Extract interpretable decision rules from a single tree in a BART (Bayesian Additive Regression Trees) model. Each terminal node (leaf) becomes one rule representing the path from root to that leaf.

Usage

```
## S3 method for class 'bart'
extract_rules(x, tree = 1L, chain = 1L, ...)
```

Arguments

x	A bart object from the **dbarts** package fitted with keeptrees = TRUE.
tree	Integer specifying which tree to extract rules from. Uses 1-based indexing (default is 1L). BART models contain n.trees trees in the ensemble.
chain	Integer specifying which MCMC chain to extract from. Uses 1-based indexing (default is 1L). Only relevant for models fitted with multiple chains.
...	Not currently used.

Details

The BART model must be fitted with keeptrees = TRUE to enable tree extraction. This function uses 1-based indexing for the tree parameter and output id column (R convention).

Split conditions in BART follow the pattern: left child when feature < threshold, right child when feature >= threshold. Rules are combinations of these conditions using AND logic.

Value

A tibble with class c("rule_set_bart", "rule_set") and columns:

- tree: integer, the tree number (matches input parameter).
- rules: list of R expressions, one per terminal node.
- id: integer, terminal node ID (1-based).

Examples

```
if (rlang::is_installed(c("dbarts", "palmerpenguins"))) {
  # Classification example
  data(penguins, package = "palmerpenguins")
  penguins <- na.omit(penguins)

  train_data <- penguins[, c("bill_length_mm", "bill_depth_mm",
                             "flipper_length_mm", "body_mass_g", "species")]

  set.seed(2847)
  fit <- dbarts::bart(
    x.train = train_data[, 1:4],
    y.train = train_data$species,
    keeptrees = TRUE,
    verbose = FALSE,
    ntree = 2
  )

  # Extract rules from first tree
  rules <- extract_rules(fit, tree = 1L)
```

```
  # View as text
  rule_text(rules$rules[[1]])

  # Regression example
  data(mtcars)
  set.seed(5193)
  fit_reg <- dbarts::bart(
    x.train = mtcars[, -1],
    y.train = mtcars$mpg,
    keeptrees = TRUE,
    verbose = FALSE,
    ntree = 2
  )
  rules_reg <- extract_rules(fit_reg, tree = 1L)
}
```

`extract_rules.C5.0` *Extract an expression that defines a path to a terminal node*

Description

A rule is a logical expression of predictor variables that reflects which data are contained in or sent to a terminal node in a tree-based model. Rules can take any form but, for most trees, they are simple statements such as `x < 1.2`, `y == "red"`, or `z %in% c("blue", "green")`.

Usage

```
## S3 method for class 'C5.0'
extract_rules(x, tree = 1L, ...)

## S3 method for class 'cforest'
extract_rules(x, tree = 1L, ...)

extract_rules(x, ...)

## S3 method for class 'grf'
extract_rules(x, tree = 1L, ...)

## S3 method for class 'randomForest'
extract_rules(x, tree = 1L, data = NULL, ...)

## S3 method for class 'ranger'
extract_rules(x, tree = 1L, data = NULL, ...)
```

Arguments

x	A object
tree	Integer vector specifying which trees to extract rules from. Default is 1L for the first tree. Values must be between 1 and the number of trees in the forest (x$num.trees).
...	Other arguments passed to methods
data	Data.frame containing the training data. Required for ranger models to properly extract rules with fitted values and node summaries.

Value

A data frame with column rules (an R expression) and id (an identifier).

Examples

```
fit <- partykit::ctree(Species ~ ., data = iris)
extract_rules(fit)
```

---

extract_rules.cubist *Extract rules from a Cubist model*

Description

Extracts rule conditions from a Cubist regression model as R expressions. Each rule consists of conditions that define when a linear model applies.

Usage

```
## S3 method for class 'cubist'
extract_rules(x, committee = 1L, ...)
```

Arguments

x	A cubist object from the Cubist package.
committee	An integer vector specifying which committee(s) to extract rules from. Defaults to 1L (first committee). Values must be between 1 and the total number of committees in the model.
...	Not used.

Details

Cubist models use committees (similar to boosting iterations) where each committee contains multiple rules. Each rule has:

- Conditions that determine when the rule applies (splits on predictors)
- A linear model that makes predictions when conditions are met

This function extracts the conditions as R expressions that can be evaluated on data. Rules with no conditions (applying to all data) return `TRUE`.

The expressions use standard R operators:

- Continuous splits: `>`, `<=`, etc.
- Categorical single value: `==`
- Categorical multiple values: `%in%`
- Missing values: `is.na()`

Value

A tibble with columns:

- `committee`: Integer committee number
- `id`: Integer rule number within the committee
- `rules`: List column containing R expressions for each rule's conditions

See Also

`rules::tidy.cubist()` for extracting rules as text strings

Examples

```
if (rlang::is_installed("Cubist")) {
library(Cubist)
library(lorax)

# Create sample data
set.seed(1)
n <- 100
p <- 5
X <- matrix(rnorm(n * p), n, p)
colnames(X) <- paste0("x", 1:p)
y <- X[, 1] + X[, 2]^2 + rnorm(n)

# Fit Cubist model with multiple committees
mod <- cubist(X, y, committees = 3)

# Extract rules from first committee
rules <- extract_rules(mod)
rules
```

```
# Extract from multiple committees
rules_all <- extract_rules(mod, committee = 1:3)

# Convert to readable text
rule_text(rules$rules[[1]])
}
```

`extract_rules.lgb.Booster`

Extract rules from an lgb.Booster model

Description

Extract interpretable decision rules from a single tree in a LightGBM boosted tree model. Each terminal node (leaf) becomes one rule representing the path from root to that leaf.

Usage

```
## S3 method for class 'lgb.Booster'
extract_rules(x, tree = 1L, ...)
```

Arguments

`x`	An `lgb.Booster` object from the **lightgbm** package.
`tree`	Integer specifying which tree to extract rules from. Uses 1-based indexing (default is `1L`). For multiclass models with `num_class` classes and `nrounds` boosting rounds, there are `num_class * nrounds` total trees.
`...`	Not currently used.

Details

lightgbm uses 0-based indexing internally, but this function uses 1-based indexing for the `tree` parameter and output `id` column (R convention).

Split conditions in **lightgbm** follow the pattern: left child when feature <= threshold, right child when feature > threshold. Rules are combinations of these conditions using AND logic.

Note: This function does not work with **lightgbm** models containing categorical features.

Value

A tibble with class `c("rule_set_lgb.Booster", "rule_set")` and columns:

- `tree`: integer, the tree number (matches input parameter).
- `rules`: list of R expressions, one per terminal node.
- `id`: integer, terminal node ID (1-based).

Examples

```
if (rlang::is_installed("lightgbm")) {
  # Binary classification
  data(agaricus.train, package = "lightgbm")
  dtrain <- lightgbm::lgb.Dataset(
    agaricus.train$data,
    label = agaricus.train$label
  )
  set.seed(2847)
  bst <- lightgbm::lgb.train(
    params = list(objective = "binary", max_depth = 3, num_threads = 1L),
    data = dtrain,
    nrounds = 3,
    verbose = -1
  )

  # Extract rules from first tree
  rules <- extract_rules(bst, tree = 1L)

  # View as text
  rule_text(rules$rules[[1]])

  # Regression example
  data(mtcars)
  dtrain_reg <- lightgbm::lgb.Dataset(as.matrix(mtcars[, -1]), label = mtcars$mpg)
  set.seed(5193)
  bst_reg <- lightgbm::lgb.train(
    params = list(
      objective = "regression", max_depth = 3, min_data_in_leaf = 1,
      num_threads = 1L
    ),
    data = dtrain_reg,
    nrounds = 3,
    verbose = -1
  )
  rules_reg <- extract_rules(bst_reg, tree = 1L)
}
```

`extract_rules.ObliqueForest` *Extract rules from an ObliqueForest model*

Description

Extracts the decision rules for terminal nodes in a specified tree from an aorsf ObliqueForest model. Each rule represents the path from the root node to a terminal node using oblique (linear combination) splits.

Usage

```
## S3 method for class 'ObliqueForest'
extract_rules(x, tree = 1L, ...)
```

Arguments

x	An ObliqueForest object from the **aorsf** package
tree	Integer specifying which tree to extract rules from (1-based). Default is 1L for the first tree. Must be between 1 and the number of trees in the forest (x$n_tree).
...	Other arguments passed to methods

Details

Tree and Node Indexing:

Both the tree parameter and the id column use **1-based indexing** for user convenience, matching R's standard indexing convention:

- tree = 1 extracts rules from the first tree
- id = 1 refers to the first terminal node

Internally, aorsf uses 0-based indexing (where node 0 is the root), but this is automatically converted to 1-based indexing in the output for consistency with R conventions.

Factor Variables and Reference Coding:

The **aorsf** package internally converts unordered factor variables using **reference coding** (also called dummy coding). For a factor with k levels, aorsf creates k-1 binary indicator variables, with the first level serving as the reference category:

- A factor color with levels ["red", "blue", "green"] creates indicators for blue and green only. When both indicators are 0, it represents red.
- The extracted rules automatically convert these back to factor comparisons: 2.1 * color_blue becomes 2.1 * (color == "blue").
- Rules can be evaluated directly on data with the original factor columns (no need to manually create indicator variables).
- Ordered factors are converted to a single integer variable representing the ordinal level, not to multiple indicators.

Reference coding prevents collinearity in the internal regression computations used to find optimal splits.

Predictor Scaling:

The **aorsf** package **always scales data during prediction**, regardless of the scale_x parameter setting. The coefficients stored in trees are for scaled data: (x - mean) / sd for numeric predictors.

To make rules work with unscaled input data, the extracted rules automatically **include the scaling transformation** in the expressions themselves. For example, instead of showing a pre-computed unscaled coefficient, rules show:

```
728.58 * ((flipper_length_mm - 200.97) / 14.02) > 400.23
```

This approach:

- Allows rules to be evaluated directly on original (unscaled) data
- Preserves full floating-point precision (avoids errors from pre-computing unscaled coefficients)
- Makes the scaling transformation explicit and transparent

Factor indicator variables are not scaled since they are binary 0/1 values.

Value

A tibble with columns:

- `tree`: integer, the tree number (1-based).
- `rules`: list of R expressions, one per terminal node.
- `id`: integer, the terminal node ID (1-based for user convenience).

Examples

```
if (rlang::is_installed(c("aorsf", "palmerpenguins"))) {
  # Classification example
  penguins <- palmerpenguins::penguins[complete.cases(palmerpenguins::penguins), ]
  set.seed(2847)
  forest <- aorsf::orsf(
    species ~ ., data = penguins, n_tree = 3, n_thread = 1
  )

  # Extract rules from first tree (default)
  rules <- extract_rules(forest)

  # View rules as text
  rules$rules[[1]] |> rule_text(bullets = TRUE) |> cat("\n")

  # Extract rules from different tree
  rules3 <- extract_rules(forest, tree = 3L)

  # Regression example
  data(mtcars)
  set.seed(5193)
  forest_reg <- aorsf::orsf(
    mpg ~ ., data = mtcars, n_tree = 3, n_thread = 1
  )
  rules_reg <- extract_rules(forest_reg, tree = 1L)
}
```

extract_rules.party *Extract rules from a party object*

Description

Extract interpretable decision rules from a **partykit** `party` or `constparty` object. Each terminal node becomes one rule representing the path from root to that leaf.

Usage

```
## S3 method for class 'party'
extract_rules(x, ...)
```

Arguments

`x`	A `party` or `constparty` object from the **partykit** package.
`...`	Not currently used.

Value

A tibble with class `c("rule_set_party", "rule_set")` and columns:

- `id`: integer, the terminal node ID.
- `rules`: list of R expressions, one per terminal node.

Examples

```
fit <- partykit::ctree(Species ~ ., data = iris)
extract_rules(fit)
```

extract_rules.rpart *Extract rules from an rpart model*

Description

Extract interpretable decision rules from an **rpart** decision tree. Each terminal node becomes one rule representing the path from root to that leaf.

Usage

```
## S3 method for class 'rpart'
extract_rules(x, ...)
```

Arguments

x	An rpart object from the **rpart** package.
...	Not currently used.

Value

A tibble with class c("rule_set_rpart", "rule_set") and columns:

- id: integer, the terminal node ID.
- rules: list of R expressions, one per terminal node.

Examples

```
fit <- rpart::rpart(Species ~ ., data = iris)
extract_rules(fit)
```

---

extract_rules.xgb.Booster

Extract rules from an xgb.Booster model

Description

Extract interpretable decision rules from a single tree in an **xgboost** boosted tree model. Each terminal node (leaf) becomes one rule representing the path from root to that leaf.

Usage

```
## S3 method for class 'xgb.Booster'
extract_rules(x, tree = 1L, nthread = NULL, ...)
```

Arguments

x	An xgb.Booster object from the **xgboost** package.
tree	Integer specifying which tree to extract rules from. Uses 1-based indexing (default is 1L). For multiclass models with num_class classes and nrounds boosting rounds, there are num_class * nrounds total trees.
nthread	Integer number of threads to use when reading the tree structure out of the model. The default (NULL) inherits the nthread the booster was trained with.
...	Not currently used.

Details

xgboost uses 0-based indexing internally, but this function uses 1-based indexing for the `tree` parameter and output `id` column (R convention).

Split conditions in **xgboost** follow the pattern: Yes branch when feature < threshold, No branch when feature >= threshold. Rules are combinations of these conditions using AND logic.

Note: This function does not work with **xgboost** models containing categorical features or non-tree boosters (`gblinear`).

Value

A tibble with class `c("rule_set_xgb.Booster", "rule_set")` and columns:

- `tree`: integer, the tree number (matches input parameter).
- `rules`: list of R expressions, one per terminal node.
- `id`: integer, terminal node ID (1-based).

Examples

```
if (rlang::is_installed("xgboost")) {
  data(agaricus.train, package = "xgboost")

  # Binary classification on a small subset for a fast example.
  rows <- seq_len(200)
  set.seed(2847)
  bst <- xgboost::xgb.train(
    data = xgboost::xgb.DMatrix(
      agaricus.train$data[rows, ],
      label = agaricus.train$label[rows],
      nthread = 1
    ),
    nrounds = 3,
    params = xgboost::xgb.params(
      max_depth = 3,
      objective = "binary:logistic",
      nthread = 1
    )
  )

  # Extract rules from first tree
  rules <- extract_rules(bst, tree = 1L)

  # View as text
  rule_text(rules$rules[[1]])

  # Regression example
  data(mtcars)
  set.seed(8472)
  bst_reg <- xgboost::xgb.train(
    data = xgboost::xgb.DMatrix(
      as.matrix(mtcars[, -1]),
```

```
    label = mtcars$mpg,
    nthread = 1
  ),
  nrounds = 3,
  params = xgboost::xgb.params(
    max_depth = 3,
    objective = "reg:squarederror",
    nthread = 1
  )
 )
 rules_reg <- extract_rules(bst_reg, tree = 1L)
}
```

`obliq_split_to_expr` *Convert an oblique split to an R expression*

Description

This function converts an oblique split condition (linear combination) from a tree-based model into a valid R expression. Oblique splits use a weighted sum of multiple variables compared to a threshold.

Usage

```
obliq_split_to_expr(split)
```

Arguments

`split`	A named list with four required elements: • `columns`: character vector - variable names for the linear combination. • `values`: numeric vector - coefficients for each variable (same length as. `columns`) • `operator`: character string - one of: <, <=, >, >=, ==. • `threshold`: numeric scalar - the threshold value for comparison.

Value

An R expression object that can be evaluated. The expression represents: `values[1]*columns[1] + .. + values[n]*colum`

Examples

```
# Simple oblique split with two variables
obliq_split_to_expr(list(
  columns = c("x", "y"),
  values = c(2, 3),
  operator = ">",
  threshold = 10
```

```
))

# Oblique split with negative coefficients
obliq_split_to_expr(list(
  columns = c("age", "income"),
  values = c(1.5, -0.001),
  operator = "<=",
  threshold = 50
))

# Three-variable oblique split
obliq_split_to_expr(list(
  columns = c("x", "y", "z"),
  values = c(1, 2, -1),
  operator = ">=",
  threshold = 0
))

# Evaluate the expression
expr <- obliq_split_to_expr(list(
  columns = c("x", "y"),
  values = c(1, 1),
  operator = ">",
  threshold = 5
))
test_data <- data.frame(x = c(2, 3, 4), y = c(2, 3, 4))
test_data[eval(expr, test_data), ]
```

`rect_split_to_expr` *Convert a rectangular split to an R expression*

Description

This function converts a split condition from a tree-based model into an valid R expression. It is primarily used as a building block for `extract_rules()` to construct paths to terminal nodes.

Usage

```
rect_split_to_expr(split)
```

Arguments

`split`	A named list with three required elements: • `column`: character string - variable name for the split. • `value`: numeric, character, or character vector - the split. threshold/value(s) • `operator`: character string - one of: <, <=, >, >=, ==,. `%in%`

Value

An R expression object that can be evaluated.

Examples

```
# Numeric comparison
rect_split_to_expr(list(column = "age", value = 25, operator = "<"))

# Single character value uses ==
rect_split_to_expr(list(column = "color", value = "red", operator = "=="))

# Multiple character values use %in%
rect_split_to_expr(
  list(column = "color", value = c("red", "blue"), operator = "%in%")
)

# Evaluate the expression
expr <- rect_split_to_expr(list(column = "age", value = 30, operator = ">="))
test_data <- data.frame(age = c(20, 30, 40))
test_data[eval(expr, test_data), ]
```

---

rule_text *Convert a rule expression to a readable text format*

Description

This function formats R expressions representing rules from tree-based models as character strings. It provides options for formatting numeric values, displaying rules as bulleted lists, and controlling output width.

Usage

```
rule_text(
  expr,
  bullets = FALSE,
  digits = 4,
  max_width = Inf,
  key = NULL,
  max_group_nchar = Inf
)
```

Arguments

expr	An R expression to format. Typically created by rect_split_to_expr() or combine_rule_elements().

bullets — Logical indicating whether to break apart rule elements and display as a bulleted list. If TRUE, splits on & operators and creates a bulleted list with each condition on a new line. If FALSE (default), returns as a single line.

digits — Integer number of significant digits to use when formatting numeric values in the rule. Default is 4.

max_width — Maximum width for the output when bullets = FALSE. If the formatted rule exceeds this width, it will be truncated with .. appended. The .. is included in the width count. Default is Inf (no truncation).

key — Optional data frame or tibble with columns original and label (both character). When provided, variable names matching values in original are substituted with corresponding values in label in the printed output. The original column must not contain duplicates. If a variable name is not in key$original, it remains unchanged.

max_group_nchar — Maximum number of characters for value lists in %in% operations. When the total character count of values in a c(...) vector exceeds this limit, the values are replaced with {X values} where X is the count. Default is Inf (no abbreviation). Character count includes quotes and separators as they appear in deparsed output.

Value

A character string containing the formatted rule. When bullets = TRUE, conditions are separated by newlines with bullet markers.

Examples

```
# Simple numeric rule
rule1 <- rlang::expr(age >= 30)
rule_text(rule1)

# Multiple conditions
rule2 <- rlang::expr(age >= 30 & income > 50000)
rule_text(rule2)

# Bulleted format
cat(rule_text(rule2, bullets = TRUE), "\n")

# Control numeric precision
rule3 <- rlang::expr(x > 1.23456789)
rule_text(rule3, digits = 2)
rule_text(rule3, digits = 6)

# Truncate long rules
rule4 <- rlang::expr(very_long_variable_name > 100 & another_long_name < 50)
rule_text(rule4, max_width = 30)

# With label substitution
expr <- rlang::expr(pct_owed > 0.5 & amount < 1000)
key <- tibble::tibble(
```

```
  original = c("pct_owed", "amount"),
  label = c("percentage owed by customer", "total amount")
)
rule_text(expr, key = key)

# Integration with other helpers
split1 <- list(column = "age", value = 30.5, operator = ">=")
split2 <- list(column = "income", value = 50000, operator = ">")
split3 <- list(column = "city", value = c("NYC", "LA"), operator = "%in%")
rule <- combine_rule_elements(list(
  rect_split_to_expr(split1),
  rect_split_to_expr(split2),
  rect_split_to_expr(split3)
))
cat(rule_text(rule, bullets = TRUE), "\n")

# Abbreviate long value lists
split4 <- list(
  column = "county",
  value = c("adams", "benton", "chelan", "clallam"),
  operator = "%in%"
)
rule_long <- rect_split_to_expr(split4)
rule_text(rule_long) # Full list
rule_text(rule_long, max_group_nchar = 20) # Abbreviated
```

---

var_imp.ObliqueForest *Tree Importance Scores*

Description

Methods for computing variable importance scores via the model object using a common interface.

Usage

```
## S3 method for class 'ObliqueForest'
var_imp(object, complete = TRUE, ...)

## S3 method for class 'cforest'
var_imp(object, complete = TRUE, ...)

## S3 method for class 'grf'
var_imp(object, complete = TRUE, ...)

## S3 method for class 'lgb.Booster'
var_imp(object, complete = TRUE, feature_names = NULL, ...)

## S3 method for class 'party'
```

```
var_imp(object, complete = TRUE, ...)

## S3 method for class 'randomForest'
var_imp(object, complete = TRUE, type = NULL, ...)

## S3 method for class 'ranger'
var_imp(object, complete = TRUE, ...)

## S3 method for class 'rpart'
var_imp(object, complete = TRUE, ...)

## S3 method for class 'xgb.Booster'
var_imp(object, complete = TRUE, feature_names = NULL, nthread = NULL, ...)
```

Arguments

object	A model object.
complete	A logical to filling absent importance values with zeros.
...	Arguments passed to importance functions (if any).
feature_names	Character vector of feature names to include when complete = TRUE. For **xgboost** models, the model object does not store unused feature names, so this parameter allows you to specify the complete feature set. If NULL (default), only features that appear in at least one tree will be included.
type	Character string specifying which importance measure to extract. For classification forests, options are "gini" (default, uses MeanDecreaseGini), "accuracy" (uses MeanDecreaseAccuracy), or a class name. For regression forests, options are "mse" (default, uses IncNodePurity) or "permutation" (uses %IncMSE). If NULL, uses the default for the forest type.
nthread	Integer number of threads to use when reading the tree structure out of the model. The default (NULL) inherits the nthread the booster was trained with.

Details

Different engines compute importances differently:

- rpart::rpart(), xgboost::xgb.importance(), and lightgbm::lgb.importance() follow the change in the objective function (e.g., Gini, MSE, gain, ...) as the tree is constructed and reports the aggregate improvement in these statistics as importance.
- randomForest::importance() and ranger::ranger() produce standard permutation-based importance scores.
- grf::variable_importance() states that a "simple weighted sum of how many times feature i was split on at each depth in the forest" is used.

Keep in mind that, for rpart::rpart(), the importance calculation is affected by competing and surrogate splits. Consequently, there might be non-zero importances for predictors that were not used in any actual split in the tree. To make the splits and importances align, use the options maxcompete = 0 and maxsurrogate = 0.

Value

A tibble with columns `term` and `estimate`.

Examples

```
fit <- partykit::ctree(Species ~ ., data = iris)
var_imp(fit)
```

Index