Package 'ggRandomForests'

November 3, 2026

Type Package

Title Visually Exploring Random Forests

Version 4.0.0

Date 2026-11-02

License MIT + file LICENSE

Encoding UTF-8

Language en-US

URL https://github.com/ehrlinger/ggRandomForests,
https://ehrlinger.github.io/ggRandomForests/

BugReports https://github.com/ehrlinger/ggRandomForests/issues

Description Graphic elements for exploring Random Forests using 'randomForestSRC' and 'randomForest' for survival, regression, and classification forests, and 'varPro' for rule-based variable priority. Suggests 'randomForestRHF' for hazards with time-dependent covariates. Implements visualizations of the methods described in Breiman (2001) <doi:10.1023/A:1010933404324>, Ishwaran, Kogalur, Blackstone, and Lauer (2008) <doi:10.1214/08-AOAS169>, Lu and Ishwaran (2024) <doi:10.48550/arXiv.2409.09003>, and Ishwaran, Hsich, Kogalur, and Lee (2026) <doi:10.48550/arXiv.2608.21597>.

Depends R (>= 4.3.0)

Imports randomForestSRC (>= 3.4.0), randomForest, varPro (>= 3.3.0), igraph, survival, tidyr, dplyr, ggplot2, patchwork, stringr

Suggests bookdown, callr, covr, datasets, ggraph, kernelshap, knitr, lintr, magick, MASS, pkgdown, pkgload, quarto, ragg, randomForestRHF (>= 2.0.3), RColorBrewer, rmarkdown, testthat, vdiffr, withr

VignetteBuilder quarto

SystemRequirements Quarto command line tool (<https://github.com/quarto-dev/quarto-cli>)

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author John Ehrlinger [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0002-5340-5154>)

Maintainer John Ehrlinger <john.ehrlinger@gmail.com>

Repository CRAN

Date/Publication 2026-11-03 22:50:30 UTC

Contents

ggRandomForests-package 3
autoplot.gg 6
calc_auc 8
calc_roc.rfsrc 9
gg_ale_rfsrc 11
gg_auct 13
gg_beta_uvarpro 15
gg_beta_varpro 16
gg_brier 19
gg_error 21
gg_isopro 25
gg_ivarpro 28
gg_partial 31
gg_partial_rfsrc 33
gg_partial_varpro 36
gg_rfsrc.rfsrc 44
gg_rhf 46
gg_rhf_importance 48
gg_roc.rfsrc 50
gg_sdependent 52
gg_shap 53
gg_survival 54
gg_tune_rhf 57
gg_udependent 58
gg_variable 61
gg_varpro 63
gg_vimp 67
kaplan 71
nelson 72
plot.gg_ale_interaction 74
plot.gg_ale_rfsrc 75
plot.gg_auct 76
plot.gg_beta_uvarpro 77
plot.gg_beta_varpro 78
plot.gg_brier 79
plot.gg_error 80
plot.gg_isopro 84

plot.gg_ivarpro . . . 85
plot.gg_partial . . . 87
plot.gg_partial_rfsrc . . . 88
plot.gg_partial_varpro . . . 90
plot.gg_rfsrc . . . 95
plot.gg_rhf . . . 98
plot.gg_rhf_importance . . . 99
plot.gg_roc . . . 101
plot.gg_sdependent . . . 103
plot.gg_shap . . . 104
plot.gg_survival . . . 105
plot.gg_tune_rhf . . . 107
plot.gg_udependent . . . 108
plot.gg_variable . . . 109
plot.gg_varpro . . . 112
plot.gg_vimp . . . 114
print.gg_beta_uvarpro . . . 116
quantile_pts . . . 118
shap_beeswarm . . . 119
shap_dependence . . . 120
shap_importance . . . 121
summary.gg_beta_uvarpro . . . 121
surv_partial.rfsrc . . . 124
varpro_coverage . . . 126
varpro_feature_names . . . 129

Index **131**

`ggRandomForests-package`

ggRandomForests: Visually Exploring Random Forests

Description

`ggRandomForests` provides `ggplot2` (Wickham 2009) diagnostic and exploration figures for fits from `randomForestSRC::rfsrc()` (`rfsrc` class; **randomForestSRC** >= 3.4.0), `randomForestRHF::rhf()` (`rhf` class; **randomForestRHF** >= 2.0.0), `varPro::varpro()` (`varpro` class; **varPro** >= 3.1.0), or `randomForest`.

`randomForestSRC` gives a unified treatment of Breiman's (2001) random forests across data settings: regression and classification forests when the response is numeric or categorical, survival and competing-risk forests (Ishwaran et al. 2008) for right-censored data.

Details

The package is built on one decision: keep the data step and the figure step apart. A `gg_*` function pulls a tidy data object out of the forest; its `plot()` method turns that object into a figure. Two things follow.

The data object stands on its own. It carries everything its plot needs, so you can save it, inspect it, or come back to it later without keeping the original forest (which can be large) in memory.

You are never locked into the default figure. Each `plot()` method returns a single plottable object: a `ggplot` you extend with `+`, or a `patchwork` composite for the multi-panel methods. Add layers, swap scales, apply a theme, or ignore the default entirely and build the figure from the tidy data yourself. Every `gg_*` object also carries `print()` and `summary()` methods: `print()` shows a short header rather than dumping every row, and `summary()` returns a diagnostics object.

Forest diagnostics

- `gg_rfsrc`: predicted versus observed values.
- `gg_error`: OOB error against the number of trees.
- `gg_vimp`: variable importance ranking (Ishwaran et al. 2010).
- `gg_variable`: marginal variable dependence.
- `gg_roc`: ROC curves for classification forests (see also `calc_roc` and `calc_auc`).
- `gg_survival`: Kaplan-Meier / Nelson-Aalen estimates.
- `gg_brier`: time-resolved Brier score and CRPS for survival forests.

Partial dependence

- `gg_partial`: tidies the output of `randomForestSRC::plot.variable(partial = TRUE)`.
- `gg_partial_rfsrc`: computes partial dependence from the fitted forest directly, via `partial.rfsrc`.

SHAP explanations

- `gg_shap`: SHAP values for regression and classification forests, wrapping `kernelshap`, with `shap_importance`, `shap_beeswarm` and `shap_dependence` figures.

varPro rule-based variable selection

- `gg_varpro`: variable importance from a `varpro` fit.
- `gg_partial_varpro` (alias `gg_partialpro`): partial effects on an interpretable scale.
- `gg_beta_varpro`: per-region lasso coefficients.
- `gg_isopro`: isolation-forest outlier scores.
- `gg_ivarpro`: individual (per-observation) importance.

Unsupervised varPro

- `gg_udependent`: variable dependency graph.
- `gg_beta_uvarpro`: entropy-based variable ranking.
- `gg_sdependent`: signal-variable detection.

Random Hazard Forests These functions are for time-to-event data whose predictors can change during follow-up. They show how event risk, time-varying discrimination, and variable priority change over time, and let you inspect a saved tree-size tuning search.

- `gg_rhf`: per-case event-risk curves over time.
- `gg_auct`: AUC curves over follow-up time.
- `gg_rhf_importance`: variable priority by time window.
- `gg_tune_rhf`: saved tree-size tuning paths by OOB risk or OOB iAUC.

See `vignette("rhf", package = "ggRandomForests")` for a longitudinal workflow that uses all four RHF families.

varPro is a required dependency (`Imports`), so the varPro families are always available. **randomForestRHF** and **kernelshap** are in `Suggests`: their functions check for the package and fail with a clear message when it is not installed.

References

Breiman, L. (2001). Random forests, Machine Learning, 45:5-32.

Ishwaran H, Kogalur U (2026). Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC). R package version 3.6.2. https://cran.r-project.org/package=randomForestSRC

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R. R News 7(2), 25–31.

Ishwaran H., Kogalur U.B., Blackstone E.H. and Lauer M.S. (2008). Random survival forests. Ann. Appl. Statist. 2(3), 841–860.

Ishwaran, H., U. B. Kogalur, E. Z. Gorodeski, A. J. Minn, and M. S. Lauer (2010). High-dimensional variable selection for survival data. J. Amer. Statist. Assoc. 105, 205-217.

Ishwaran, H. (2007). Variable importance in binary regression trees and forests. Electronic J. Statist., 1, 519-537.

Lu M, Ishwaran H (2024). Model-independent variable selection via the rule-based variable priority framework. arXiv:2409.09003. https://arxiv.org/abs/2409.09003

Ishwaran H, Kogalur U (2026). Model-Independent Variable Selection via the Rule-Based Variable Priority. R package version 3.2.0. https://CRAN.R-project.org/package=varPro

Ishwaran H, Hsich EM, Kogalur UB, Lee DKK (2026). Random Hazard Forests. arXiv:2608.21597. doi:10.48550/arXiv.2608.21597.

Ishwaran H, Kogalur UB (2026). *randomForestRHF: Random Hazard Forests*. R package version 2.0.3. https://CRAN.R-project.org/package=randomForestRHF

Wickham, H. ggplot2: elegant graphics for data analysis. Springer New York, 2009.

`autoplot.gg` `autoplot` *methods for* **ggRandomForests** *data objects*

Description

These let you call `ggplot2::autoplot()` on any `gg_*` object **ggRandomForests** returns. Each is a thin wrapper around the matching `plot.gg_*()` S3 method, and `...` passes straight through, so every argument those plot methods take is still available here.

Usage

```
## S3 method for class 'gg_error'
autoplot(object, ...)

## S3 method for class 'gg_vimp'
autoplot(object, ...)

## S3 method for class 'gg_rfsrc'
autoplot(object, ...)

## S3 method for class 'gg_variable'
autoplot(object, ...)

## S3 method for class 'gg_partial'
autoplot(object, ...)

## S3 method for class 'gg_partial_rfsrc'
autoplot(object, ...)

## S3 method for class 'gg_partialpro'
autoplot(object, ...)

## S3 method for class 'gg_partial_varpro'
autoplot(object, ...)

## S3 method for class 'gg_roc'
autoplot(object, ...)

## S3 method for class 'gg_survival'
autoplot(object, ...)

## S3 method for class 'gg_brier'
autoplot(object, ...)

## S3 method for class 'gg_rhf'
autoplot(object, ...)
```

```
## S3 method for class 'gg_tune_rhf'
autoplot(object, ...)

## S3 method for class 'gg_auct'
autoplot(object, ...)

## S3 method for class 'gg_rhf_importance'
autoplot(object, ...)

## S3 method for class 'gg_varpro'
autoplot(object, ...)

## S3 method for class 'gg_udependent'
autoplot(object, ...)

## S3 method for class 'gg_isopro'
autoplot(object, ...)

## S3 method for class 'gg_shap'
autoplot(object, ...)

## S3 method for class 'gg_ale_rfsrc'
autoplot(object, ...)

## S3 method for class 'gg_ale_interaction'
autoplot(object, ...)
```

Arguments

`object`	A `gg_*` data object (see Details).
`...`	Additional arguments passed to the underlying `plot.gg_*()` method.

Details

The following `gg_*` classes are supported:

`gg_error` OOB error vs. number of trees

`gg_vimp` Variable importance ranking

`gg_rfsrc` Predicted vs. observed values

`gg_variable` Marginal dependence

`gg_partial` Partial dependence (via `plot.variable`)

`gg_partial_rfsrc` Partial dependence (via `partial.rfsrc`)

`gg_partial_varpro` Partial dependence (via **varPro**)

`gg_partialpro` Partial dependence via **varPro** (deprecated alias)

`gg_varpro` Variable importance from **varPro**

`gg_roc` ROC curve

gg_survival Survival / cumulative hazard curves

gg_brier Time-resolved Brier score and CRPS

gg_rhf Random Hazard Forest hazard curves

gg_tune_rhf Random Hazard Forest tuning path

gg_auct Random Hazard Forest time-varying AUC

gg_rhf_importance Random Hazard Forest variable priority

gg_shap SHAP explanations

Value

A ggplot object.

Examples

```
library(ggplot2)
set.seed(42)
rf <- randomForestSRC::rfsrc(Ozone ~ ., data = na.omit(airquality),
                             ntree = 50, importance = TRUE,
                             tree.err = TRUE)
autoplot(gg_error(rf))
autoplot(gg_vimp(rf))
```

---

calc_auc *Area Under the ROC Curve calculator*

Description

Area Under the ROC Curve calculator

Usage

```
calc_auc(x)
```

Arguments

x	gg_roc object

Details

calc_auc uses the trapezoidal rule to calculate the area under the ROC curve.

This is a helper function for the gg_roc functions.

Value

AUC. 50% is random guessing, higher is better.

See Also

calc_roc gg_roc

plot.gg_roc

Examples

```
##
## Taken from the gg_roc example
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris, ntree = 100)

gg_dta <- gg_roc(rfsrc_iris, which_outcome = 1)

calc_auc(gg_dta)

gg_dta <- gg_roc(rfsrc_iris, which_outcome = 2)

calc_auc(gg_dta)

## randomForest tests
rf_iris <- randomForest::randomForest(Species ~ ., data = iris)
gg_dta <- gg_roc(rfsrc_iris, which_outcome = 2)

calc_auc(gg_dta)
```

---

calc_roc.rfsrc *Receiver Operator Characteristic calculator*

Description

Receiver Operator Characteristic calculator

Usage

```
## S3 method for class 'rfsrc'
calc_roc(object, dta, which_outcome = "all", oob = TRUE, ...)

calc_roc(object, dta, which_outcome = "all", oob = TRUE, ...)
```

Arguments

object	A fitted rfsrc, predict.rfsrc, or randomForest classification object containing predicted class probabilities.
dta	A factor (or coercible to factor) of the true observed class labels, one per observation. Typically object$yvar for rfsrc or object$y for randomForest.

which_outcome	Integer index of the class for which the ROC curve is computed (e.g. 1 for the first class, 2 for the second). Use "all", or its numeric spelling 0, to request all classes. The randomForest method returns a macro-averaged one-vs-rest curve; the rfsrc method warns and falls back to class 1 (see #72).
oob	Logical; if TRUE (default for rfsrc) use OOB predicted probabilities. Forced to FALSE for randomForest objects.
...	Extra arguments passed to helper functions (currently unused).

Details

For a randomForestSRC prediction and the actual response value, calculate the specificity (1-False Positive Rate) and sensitivity (True Positive Rate) of a predictor.

This is a helper function for the gg_roc functions, and not intended for use by the end user.

Value

A gg_roc data.frame with columns sens (sensitivity), spec (specificity), and pct (the probability threshold), with one row per unique prediction value. Suitable for passing to calc_auc or plot.gg_roc.

See Also

calc_auc gg_roc

plot.gg_roc

Examples

```
## Taken from the gg_roc example
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris, ntree = 100)

gg_dta <- calc_roc(rfsrc_iris, rfsrc_iris$yvar,
  which_outcome = 1, oob = TRUE
)
gg_dta <- calc_roc(rfsrc_iris, rfsrc_iris$yvar,
  which_outcome = 1, oob = FALSE
)

rf_iris <- randomForest::randomForest(Species ~ ., data = iris)
# randomForest stores the response in $y (rfsrc uses $yvar); pass the
# original training factor so calc_roc has the class labels.
gg_dta <- calc_roc(rf_iris, iris$Species,
  which_outcome = 1
)
gg_dta <- calc_roc(rf_iris, iris$Species,
  which_outcome = 2
)
```

gg_ale_rfsrc *Accumulated Local Effects (ALE) data from an rfsrc model*

Description

A partial dependence curve (gg_partial_rfsrc) marginalizes the forest's prediction by averaging over the joint distribution of the other predictors, a computation that is misleading when predictors are correlated, because it evaluates the forest at combinations of predictor values that never occur together in the data. Accumulated Local Effects (Apley and Zhu, 2020) avoid this by only ever perturbing a predictor within small local neighborhoods of its own observed values, then accumulating those local effects into a global curve.

Usage

```
gg_ale_rfsrc(
  rf_model,
  xvar.names,
  xvar2.name = NULL,
  newx = NULL,
  cat_limit = 10,
  n_eval = 25,
  which.class = 1
)
```

Arguments

rf_model	A fitted rfsrc object. Regression and classification forests only; survival is not yet implemented (see gg_shap, which has the same limitation).
xvar.names	Character vector of predictor names to compute ALE for. When xvar2.name is supplied, this must name exactly one predictor.
xvar2.name	Optional single character name of a second predictor. When supplied, second-order (interaction) ALE is computed for the pair xvar.names x xvar2.name instead of first-order ALE. Both predictors must be continuous in this version (see cat_limit).
newx	Optional data.frame of predictor values to evaluate ALE at. Defaults to the training data stored in rf_model$xvar. All column names must match rf_model$xvar.names.
cat_limit	Variables with fewer than cat_limit unique values in newx are treated as categorical; all others are continuous. Defaults to 10.
n_eval	Number of quantile bins used for a continuous predictor's ALE grid (first-order) or per axis (second-order). Defaults to 25.
which.class	For classification forests, the class (integer column index into the predicted-probability matrix) whose ALE is computed. Defaults to 1.

Details

For a continuous predictor, `gg_ale_rfsrc` bins the observed values into `n_eval` quantile-based intervals. Within each bin, every observation's predictor value is replaced first with the bin's lower edge and then with its upper edge (all other predictors held at that observation's own values), and the average change in prediction is the bin's local effect. These local effects are accumulated (cumulatively summed) across bins and centered to have a weighted mean of zero, giving a curve that is directly comparable to `gg_partial_rfsrc`'s output but immune to extrapolation into implausible predictor combinations.

For a categorical predictor, the model's own factor level order is used as the "grid": the local effect for the step from level k to level $k+1$ is averaged over the observations at both levels, each predicted at level $k+1$ and at level k, so an observation at an interior level serves two steps. Relevel the predictor before fitting the forest to control this ordering.

Supplying `xvar2.name` switches to second-order (interaction) ALE between `xvar.names` and `xvar2.name`, isolating the part of their joint effect that is not explained by either variable's own main effect. It is the ALE analogue of an interaction term, computed on a 2-D grid of bins using the same local-perturbation idea. Following Apley and Zhu, each variable's main effect is estimated from the accumulated surface by accumulating its local differences along that variable's axis, weighted by the observations in each cell, and subtracted; the surface is then centred to a weighted mean of zero. A cell with no observations takes the local effect of the nearest cell that has some. A purely additive forest returns an all-zero surface.

Value

For first-order ALE (`xvar2.name = NULL`), a named list with two elements, classed `"gg_ale_rfsrc"`:

continuous A `data.frame` with columns `x` (the bin edges, numeric), `yhat` (the centered ALE value at each edge), and `name` (variable name), for all continuous predictors.

categorical The same columns but `x` kept as a `factor` in the model's level order, for low-cardinality predictors.

For second-order ALE (`xvar2.name` supplied), a `data.frame` classed `"gg_ale_interaction"` with columns `x` (grid values of `xvar.names`), `y` (grid values of `xvar2.name`), `ale` (the interaction surface value), `name1`, and `name2`.

References

Apley, D. W. and Zhu, J. (2020). Visualizing the effects of predictor variables in black box supervised learning models. Journal of the Royal Statistical Society Series B, 82(4), 1059-1086.

See Also

`gg_partial_rfsrc`, `plot.gg_ale_rfsrc`

Examples

```
## ------------------------------------------------------------
## regression, first-order ALE
## ------------------------------------------------------------
airq.obj <- randomForestSRC::rfsrc(Ozone ~ ., data = na.omit(airquality),
```

```
                                    ntree = 100)
ale_dta <- gg_ale_rfsrc(airq.obj, xvar.names = c("Wind", "Temp"))
plot(ale_dta)


## ------------------------------------------------------------
## second-order (interaction) ALE between two continuous predictors
## ------------------------------------------------------------
ale_int <- gg_ale_rfsrc(airq.obj, xvar.names = "Wind",
                        xvar2.name = "Temp", n_eval = 15)
plot(ale_int)
```

`gg_auct` *Tidy time-varying AUC from a Random Hazard Forest*

Description

Extracts the time-dependent AUC curve from `randomForestRHF::auct.rhf()` into a tidy long data frame, one row per time point, with bootstrap confidence bounds when available and the integrated AUC (iAUC) summary attached as an attribute.

Usage

```
gg_auct(object, ...)

## S3 method for class 'auct.rhf'
gg_auct(object, ...)

## S3 method for class 'rhf'
gg_auct(
  object,
  marker = c("chf", "haz"),
  auct_fit = NULL,
  method = c("cumulative", "incident"),
  ...
)
```

Arguments

`object`	A fitted `rhf` object from **randomForestRHF**.
`...`	Further arguments passed to `randomForestRHF::auct.rhf()`, for example `bootstrap.rep` to request confidence bounds, or `riskset` for the incident definition. Ignored when `auct_fit` is supplied.
`marker`	Risk marker for the AUC: `"chf"` (cumulative hazard, default) or `"haz"` (hazard). Not used when `auct_fit` is supplied, though the value is still validated.

auct_fit — Optional precomputed randomForestRHF::auct.rhf() result (class "auct.rhf") for the same object. NULL (default) computes it. Supply it to reuse an expensive bootstrap run.

method — Which time-dependent AUC definition to compute, passed to randomForestRHF::auct.rhf(). "cumulative" (default) ranks accumulated risk through a horizon; "incident" ranks local failures within the risk set at each time. See the note below on choosing between them. Not used when auct_fit is supplied, though the value is still validated.

Value

A data.frame of class c("gg_auct", "data.frame") with columns time, auc, se, lower, upper, marker (CI columns NA when no bootstrap), an iauc attribute (a list with uno, std, uno.se, std.se, conf.level), a method attribute ("cumulative" or "incident", taken from the auct.rhf fit, which is what plot.gg_auct() names on its y axis), and a provenance attribute derived from object (source, family, ntree, n).

Note

The two definitions answer different questions rather than better and worse versions of the same one. Cumulative/dynamic AUC ranks accumulated risk through a horizon, comparing subjects who have failed by that horizon against subjects still event-free at it. Incident/dynamic AUC ranks local failures within the risk set at each time. Pick the one that matches the question you are asking, and read the two curves as separate estimands rather than as a check on each other.

References

Ishwaran H, Hsich EM, Kogalur UB, Lee DKK (2026). Random Hazard Forests. arXiv:2608.21597. doi:10.48550/arXiv.2608.21597.

Ishwaran H, Kogalur UB (2026). *randomForestRHF: Random Hazard Forests*. R package version 2.0.3. https://CRAN.R-project.org/package=randomForestRHF.

See Also

plot.gg_auct(), randomForestRHF::auct.rhf()

Examples

```
if (requireNamespace("randomForestRHF", quietly = TRUE)) {
  data(pbc, package = "randomForestSRC")
  d <- randomForestRHF::convert.counting(
    survival::Surv(days, status) ~ ., na.omit(pbc))
  o <- randomForestRHF::rhf("Surv(id, start, stop, event) ~ .", d, ntree = 30)
  plot(gg_auct(o, marker = "chf"))
}
```

gg_beta_uvarpro *Per-variable lasso-beta importance from an unsupervised varPro fit*

Description

Tidy wrapper around varPro::get.beta.entropy() for a uvarpro object. Where gg_beta_varpro() refines the *supervised* release-rule contrast, gg_beta_uvarpro() does the unsupervised analogue: uvarpro() builds entropy regions with no response, and get.beta.entropy() fits a cross-validated lasso within each region to ask how strongly every other variable explains the released variable. Averaging the absolute lasso coefficients per variable gives one number per variable: an unsupervised, lasso-flavoured importance.

Usage

```
gg_beta_uvarpro(object, ..., cutoff = NULL, beta_fit = NULL)
```

Arguments

object	A uvarpro object from varPro::uvarpro().
...	Forwarded to varPro::get.beta.entropy() when beta_fit = NULL (e.g. pre.filter, second.stage, use.cv). Ignored, with a warning, when beta_fit is supplied.
cutoff	Selection threshold on beta_mean. NULL (default) uses mean(beta_mean); a scalar sets it explicitly. Variables at or above the cutoff are flagged selected.
beta_fit	Optional pre-computed varPro::get.beta.entropy() matrix for object. When supplied, must be a numeric matrix with column names (the variables); ... is then ignored.

Details

get.beta.entropy(o) returns a (released-variable x variable) numeric matrix of absolute lasso coefficients. The column mean (na.rm = TRUE) is the per-variable importance reported here, matching the canonical sort(colMeans(beta, na.rm = TRUE), decreasing = TRUE) idiom in the varPro::uvarpro() help ("iowa housing - illustrates lasso importance").

Because get.beta.entropy() is expensive (a cross-validated glmnet per region), the beta_fit argument accepts a pre-computed matrix so you can iterate on the cutoff without re-fitting. The pairing mirrors the beta_fit argument of gg_beta_varpro().

Value

A gg_beta_uvarpro object (a data.frame), one row per variable, most-important first, with columns:

variable factor; levels reversed so the most-important variable lands at the top after coord_flip() (the gg_vimp convention).

beta_mean mean(|lasso beta|) over the released regions (colMeans(beta, na.rm = TRUE)).

n_released number of regions contributing a non-NA coefficient for the variable.

selected logical; beta_mean >= cutoff.

The provenance attribute records source, family ("unsupv"), cutoff, n_var, n_released_regions, and precomputed.

See Also

gg_beta_varpro() (supervised analogue), gg_udependent(), varPro::get.beta.entropy(), varPro::uvarpro().

Examples

```
if (requireNamespace("varPro", quietly = TRUE)) {
  set.seed(1)
  o <- varPro::uvarpro(mtcars, ntree = 50)
  gg <- gg_beta_uvarpro(o)
  plot(gg)
}
```

---

gg_beta_varpro *Per-variable lasso-beta importance from a varPro fit*

Description

Tidy wrapper around varPro::beta.varpro() for the regression or classification family. Aggregates the per-rule lasso coefficient $\hat{\beta}$ by variable into the mean absolute value $\mathrm{mean}(|\hat{\beta}|)$ and flags variables above a scalar cutoff. Optional beta_fit argument lets callers compute the expensive beta.varpro() step once and reuse the result.

Usage

```
gg_beta_varpro(object, ..., cutoff = NULL, beta_fit = NULL, which_class = NULL)
```

Arguments

object	A varpro fit from varPro::varpro() (regression or classification family).
...	Forwarded to varPro::beta.varpro() when beta_fit = NULL; ignored otherwise (with a warning). Documented forwardables: use.cv, use.1se, nfolds, maxit, thresh, max.rules.tree, max.tree.
cutoff	Selection threshold on beta_mean. NULL (default) means mean(beta_mean) across released variables. Numeric scalar otherwise.
beta_fit	Optional pre-computed varPro::beta.varpro() result for the same object. NULL (default) means the wrapper runs beta.varpro() itself. When supplied, must be a varpro-class object whose $results has columns tree / branch / variable / n.oob / imp.
which_class	For a classification fit, name of a single response level to subset on. NULL (default) returns all classes (binary fits resolve to the *last* factor level, the positive-class convention used by glm and gg_roc). Ignored with a warning on regression fits.

Value

A `data.frame` of class `c("gg_beta_varpro", "data.frame")`. For a regression fit: one row per released variable, sorted by `beta_mean` descending. For a classification fit: long-format with an extra `class` column, one row per (variable, class) pair; `variable` is a factor whose levels run in ascending `mean(|sum-of-class-beta|)`, most important last, so every facet / panel shares the same row order with the top variable at the top. `which_class` (or the binary default last-factor-level) collapses the output to a single class.

What this is doing

Think of the varPro release-rule mechanism as asking: "given a region of the feature space that the forest carved out, what changes when I remove the constraint on this one variable and let observations leave?" The standard importance answer (from `gg_varpro()`) measures that change as a z-scored contrast between local estimators: no synthetic data, no permutation. `beta.varpro()` asks the same question with a different ruler: for each rule (a tree-branch pair), it fits a one-predictor lasso regression of the response on the released variable's values, restricted to the OOB observations inside the rule's region. The wrapper aggregates those per-rule coefficients into one number per variable.

The key distinction from `gg_vimp()`, which measures Breiman-Cutler permutation importance by perturbing a variable's values and watching OOB error climb, is that neither `gg_varpro()` nor `gg_beta_varpro()` touches the data synthetically: all contrasts are between real subsets defined by the forest's rules.

What `imp` actually is (pedantic, because the column name is misleading)

The `imp` column on `beta.varpro()`'s `$results` is **not** a variable-importance score in the conventional sense. It is a regularised regression coefficient. Specifically:

- Per rule, `glmnet` fits a one-predictor lasso of the response on the released variable inside the rule's OOB region. `use.cv = TRUE` selects λ by 10-fold CV (default `nfolds = 10`); `use.1se = TRUE` (default) picks `lambda.1se`. `use.cv = FALSE` uses the full λ path.
- `imp` is the **absolute** fitted coefficient $|\hat{\beta}|$ at the chosen λ. **There is no direction here**: `varPro::beta.varpro()` wraps every coefficient it returns in `abs()`, so the sign is discarded upstream and never reaches us. Read `imp` as strength of local association, not its direction. (For a signed local estimator, see `gg_ivarpro()`, where the sign does survive.) **Magnitude depends on the predictor's units** (raw `x`, no standardization); a predictor in millimetres has a smaller $|\hat{\beta}|$ than the same predictor in metres.
- Lasso shrinkage can drive $\hat{\beta}$ to **exactly zero**. Those zeros are data, not missingness, and are kept in the aggregation. Convergence failures land as `NA_real_` and are dropped.
- The per-variable aggregate is `beta_mean`, $\mathrm{mean}(|\hat{\beta}|)$, across the rules where this variable was released. It is **not** a permutation importance, **not** a split-strength importance, and **not** directly comparable on the same numeric axis to `gg_varpro()`'s z-scores. Disagreement with `gg_varpro` is often diagnostic, not a bug.

In code form: `imp_r` is the glmnet coefficient $\hat{\beta}$ fit on `y ~ x_v` restricted to rule r, with λ chosen by `use.cv` / `use.1se`.

What's in the output

One row per released variable. Columns:

- `variable`: predictor name.
- `beta_mean`: mean of $|\hat{\beta}|$ across that variable's rules.
- `n_rules`: count of rules contributing (zero-beta rules included; only `NA` failures excluded).
- `selected`: `beta_mean >= cutoff`.

Provenance attribute carries `source`, `family`, `ntree`, `cutoff`, `cutoff_default`, `use.cv`, `n_rules_total`, `n_rules_nonzero`, `precomputed`, and `xvar.names`.

What you use this for

Picking variables when local effects matter more than aggregate split-strength contribution. Compare side-by-side with `gg_varpro()`: a variable that scores high here but low in `gg_varpro` is one whose local linear effect inside many rules is real even though its release-rule contrast is modest.

Caching

`beta.varpro()` is the expensive call (per-rule `glmnet` / `cv.glmnet`, often minutes on real data). Compute it once and reuse:

```
v <- varPro::varpro(mpg ~ ., data = mtcars, ntree = 200)
b <- varPro::beta.varpro(v, use.cv = TRUE)        # expensive, once
gg_a <- gg_beta_varpro(v, beta_fit = b)            # cheap
gg_b <- gg_beta_varpro(v, beta_fit = b, cutoff = 0.5)
```

Provenance carries `precomputed = TRUE` when `beta_fit` was supplied.

Classification

For a varpro classification fit (`object$family == "class"`, binary or multi-class), the returned frame is long-format with an extra `class` column: one row per (variable, class) pair. The `beta_mean` column aggregates the **per-class lasso coefficient** $\hat{\beta}$ stored in `beta.varpro()`'s `imp.<k>` columns (one per class level). Same pedantic-beta semantics as regression, applied independently to each class.

Binary default: `which_class = NULL` resolves to the *last* factor level of the response, the positive-class convention used by `glm` and `gg_roc`. For a 30-day-mortality outcome with levels `c("no", "yes")`, that means the wrapper shows you `"yes"` (the event) by default.

Multi-class default: `which_class = NULL` returns all K classes; the plot method renders `facet_wrap(~ class)` with one cutoff line per facet.

`which_class = "<name>"` filters to a single class regardless of K. Errors if the name isn't in the response levels.

Per-class cutoffs: `cutoff = NULL` resolves to each class's `mean(beta_mean)`. A scalar broadcasts. A named numeric vector overrides per class; missing names fall back to that class's mean.

Example (30-day mortality, binary):

```
fit <- varPro::varpro(event_30d ~ ., data = clinical, ntree = 200)
gg  <- gg_beta_varpro(fit)   # default: "yes" panel
plot(gg)
```

Reproducibility

Byte-for-byte agreement between cached (`beta_fit = b`) and uncached (`beta_fit = NULL`) outputs requires that `b` was computed by `beta.varpro(object, ...)` on the same `object`; `set.seed()` alone is not sufficient, because `beta.varpro`'s internal `cv.glmnet` fits can pick slightly different folds across separate calls. Reuse `beta_fit` when reproducibility matters.

Note

Multivariate regression (`regr+`) and survival fits are not supported. The unsupported-family path reports the family it received.

See Also

`gg_varpro()`, `gg_vimp()`, `plot.gg_beta_varpro()`, `varPro::beta.varpro()`.

Examples

```
if (requireNamespace("varPro", quietly = TRUE)) {
  set.seed(1)
  v <- varPro::varpro(mpg ~ ., data = mtcars, ntree = 50)
  b <- varPro::beta.varpro(v)
  gg <- gg_beta_varpro(v, beta_fit = b)
  plot(gg)
}
```

---

gg_brier *Brier score and CRPS for survival forests*

Description

The Brier score asks a familiar question of any probabilistic forecast: how far did the predicted probability sit from what actually happened? For a survival forest the forecast is the predicted survival probability at a given moment, and the "what happened" is whether the subject was still alive at that moment. The score is computed at every event time, so you get a curve rather than a single number, and lower is better everywhere. A perfectly calibrated forest that predicts `0` for every subject who died and `1` for every subject who survived would score `0`; a forest that predicts `0.5` for everyone scores roughly `0.25` regardless of the true outcome. That is the "uninformative" reference, not a ceiling; a forest can score worse than it.

Usage

```
gg_brier(object, ...)
```

Arguments

object	A fitted rfsrc survival forest (object$family == "surv").
...	Currently unused; accepted for S3 dispatch compatibility.

Details

This function extracts the time-resolved Brier score for a survival forest grown with randomForestSRC, both overall and broken down by mortality-risk quartile (lowest-risk to highest-risk subjects). It also returns, in the crps column, the running continuous ranked probability score (CRPS): the Brier score integrated over time and divided by elapsed time, so each value summarizes calibration up to that point on the time axis.

Because subjects are right-censored, a plain Brier score is biased: censored subjects contribute no outcome information yet still inflate the denominator. The score here uses inverse-probability-of-censoring weighting (IPCW), which up-weights uncensored observations to compensate. The censoring distribution is estimated either by Kaplan-Meier (cens.model = "km", the default) or by a separate censoring forest (cens.model = "rfsrc") when the censoring mechanism is itself covariate-dependent.

Internally, this wraps get.brier.survival and rebuilds the quartile decomposition and running CRPS from the returned brier.matx and mort components, following the approach in the internal plot.survival of **randomForestSRC**.

Value

A gg_brier data.frame with columns

time event time grid (object$time.interest).

brier overall Brier score at each time.

bs.q25, bs.q50, bs.q75, bs.q100 Brier score within each mortality-risk quartile (lowest to highest risk).

bs.lower, bs.upper 15th and 85th percentile of per-subject Brier contributions at each time. Used by plot.gg_brier(by_quartile = TRUE) to draw an envelope around the overall curve.

crps running CRPS (overall) at each time, normalized by elapsed time.

crps.q25, crps.q50, crps.q75, crps.q100 running CRPS within each mortality-risk quartile.

crps.lower, crps.upper running CRPS of the 15th / 85th per-subject Brier percentile, normalized by elapsed time.

The integrated CRPS is attached as attr(., "crps_integrated"). It is get.brier.survival()$crps, the raw area under the Brier curve, not normalized by time, so it is in the units of the time axis and grows with follow-up. The time-normalized value, get.brier.survival()$crps.std (the raw integral divided by max(.$time)), is attached as attr(., "crps_std") and reads on the Brier scale. The last value of the crps column instead divides by the time elapsed since the first event time.

Note

Brier score / CRPS is randomForestSRC survival-only; there is no randomForest method.

References

Graf E., Schmoor C., Sauerbrei W., Schumacher M. (1999). Assessment and comparison of prognostic classification schemes for survival data. Statistics in Medicine, 18(17-18):2529-2545.

Gerds T.A., Schumacher M. (2006). Consistent estimation of the expected Brier score in general survival models with right-censored event times. Biometrical Journal, 48(6):1029-1040.

See Also

plot.gg_brier, get.brier.survival, gg_error

Examples

```
library(survival)   # Surv() must be on the search path for rfsrc()
data(pbc, package = "randomForestSRC")
rfsrc_pbc <- randomForestSRC::rfsrc(
  Surv(days, status) ~ ., data = pbc, nsplit = 10, ntree = 100
)
gg_dta <- gg_brier(rfsrc_pbc)
plot(gg_dta)
plot(gg_dta, type = "crps")
plot(gg_dta, envelope = TRUE)   # overall line + 15-85% envelope

# Multi-model comparison: stack gg_brier outputs and plot with ggplot2.
rf2 <- randomForestSRC::rfsrc(
  Surv(days, status) ~ ., data = pbc, nsplit = 10, mtry = 4, ntree = 100
)
compare_dta <- dplyr::bind_rows(
  dplyr::mutate(gg_brier(rfsrc_pbc), model = "default"),
  dplyr::mutate(gg_brier(rf2),       model = "mtry=4")
)
ggplot2::ggplot(compare_dta,
  ggplot2::aes(x = time, y = brier, color = model)) +
  ggplot2::geom_line()
```

---

gg_error *Random forest error trajectory data object*

Description

Extract the cumulative out-of-bag (OOB) or in-bag training error rate from randomForestSRC and randomForest fits as a function of the number of grown trees.

Usage

```
gg_error(object, ...)
```

Arguments

object	A fitted rfsrc or randomForest object.
...	Optional arguments passed to the methods. Set training = TRUE to append the in-bag error trajectory when supported.

Details

You have to ask rfsrc for the trajectory. How many trees you get a curve over is decided by randomForestSRC::rfsrc(), not here. Its block.size argument controls how often the error is recorded, and it defaults to NULL unless you request importance; NULL records the error at the *final tree only*. So a default fit gives gg_error() a single point, not a curve, and tree.err = TRUE on its own does not change that. Grow the forest with block.size = 1 for an error recorded at every tree, or a larger block.size for every *n*th. If a plot.gg_error() figure comes back as one dot, this is why.

For randomForestSRC objects the function reshapes the rfsrc$err.rate matrix, drops the trees where rfsrc recorded nothing, and annotates the rest with the tree index required by plot.gg_error. When supplied a randomForest object, the method inspects either the $mse or $err.rate component and, when training = TRUE is requested, reconstructs the original training set via the model call to compute an in-bag error curve using per-tree predictions. Training curves are only available when the forest was stored (keep.forest = TRUE) and the original data can be recovered.

Value

A gg_error data.frame containing at least the cumulative OOB error columns and an ntree counter. When training = TRUE is honored an additional train column is included.

References

Breiman L. (2001). Random forests, Machine Learning, 45:5-32.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, Rnews, 7(2):25-31.

Ishwaran H, Kogalur U (2026). Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC). R package version 3.6.2. https://cran.r-project.org/package=randomForestSRC

See Also

plot.gg_error, gg_vimp, gg_variable, rfsrc, randomForest, plot.rfsrc

Examples

```
## Examples from RFSRC package...
## ------------------------------------------------------------
## classification example
## ------------------------------------------------------------
## ------------- iris data
## You can build a randomForest
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris,
  ntree = 100, tree.err = TRUE, block.size = 1)

# Get a data.frame containing error rates
```

```
gg_dta <- gg_error(rfsrc_iris)

# Plot the gg_error object
plot(gg_dta)

## RandomForest example
rf_iris <- randomForest::randomForest(Species ~ .,
  data = iris,
  tree.err = TRUE
)
gg_dta <- gg_error(rf_iris)
plot(gg_dta)

gg_dta <- gg_error(rf_iris, training = TRUE)
plot(gg_dta)
## ------------------------------------------------------------
## Regression example
## ------------------------------------------------------------

## ------------- airq data
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ .,
  data = airquality, ntree = 100,
  na.action = "na.impute", tree.err = TRUE, block.size = 1
)

# Get a data.frame containing error rates
gg_dta <- gg_error(rfsrc_airq)

# Plot the gg_error object
plot(gg_dta)


## ------------- Boston data
if (requireNamespace("MASS", quietly = TRUE)) {
  data(Boston, package = "MASS")
  Boston$chas <- as.logical(Boston$chas)
  rfsrc_boston <- randomForestSRC::rfsrc(medv ~ .,
    data = Boston,
    ntree = 100,
    forest = TRUE,
    importance = TRUE,
    tree.err = TRUE,
    save.memory = TRUE
  )

  # Get a data.frame containing error rates
  gg_dta <- gg_error(rfsrc_boston)

  # Plot the gg_error object
  plot(gg_dta)
}
```

```
## ------------- mtcars data
rfsrc_mtcars <- randomForestSRC::rfsrc(mpg ~ ., data = mtcars,
  ntree = 100, tree.err = TRUE, block.size = 1)

# Get a data.frame containing error rates
gg_dta<- gg_error(rfsrc_mtcars)

# Plot the gg_error object
plot(gg_dta)


## ------------------------------------------------------------
## Survival example
## ------------------------------------------------------------
## ------------- veteran data
## randomized trial of two treatment regimens for lung cancer
data(veteran, package = "randomForestSRC")
rfsrc_veteran <- randomForestSRC::rfsrc(Surv(time, status) ~ ., data = veteran,
                      ntree = 100, tree.err = TRUE, block.size = 1)

gg_dta <- gg_error(rfsrc_veteran)
plot(gg_dta)

## ------------- pbc data
data(pbc, package = "randomForestSRC", envir = environment())
# Recode: binary columns to logical, low-cardinality columns to factors.
for (ind in seq_len(dim(pbc)[2])) {
  if (!is.factor(pbc[, ind])) {
    if (length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 2) {
      if (sum(range(pbc[, ind], na.rm = TRUE) == c(0, 1)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
    }
  } else {
    if (length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 2) {
      if (sum(sort(unique(pbc[, ind])) == c(0, 1)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
      if (sum(sort(unique(pbc[, ind])) == c(FALSE, TRUE)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
    }
  }
  if (!is.logical(pbc[, ind]) &&
    length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 5) {
    pbc[, ind] <- factor(pbc[, ind])
  }
}
# Convert age from days to years
pbc$age <- pbc$age / 364.24
pbc$years <- pbc$days / 364.24
pbc <- pbc[, -which(colnames(pbc) == "days")]
pbc$treatment <- as.numeric(pbc$treatment)
```

```
pbc$treatment[which(pbc$treatment == 1)] <- "DPCA"
pbc$treatment[which(pbc$treatment == 2)] <- "placebo"
pbc$treatment <- factor(pbc$treatment)
# Split into a training set (assigned treatment) and a test set (not assigned)
dta_train <- pbc[-which(is.na(pbc$treatment)), ]
pbc_test <- pbc[which(is.na(pbc$treatment)), ]

#========
# build the forest:
rfsrc_pbc <- randomForestSRC::rfsrc(
  Surv(years, status) ~ .,
 dta_train,
 ntree = 100,
 nsplit = 10,
 na.action = "na.impute",
 tree.err = TRUE,
 forest = TRUE,
 importance = TRUE,
 save.memory = TRUE
)


gg_dta <- gg_error(rfsrc_pbc)
plot(gg_dta)
```

`gg_isopro` *Tidy data from a varPro isolation-forest fit*

Description

Pulls per-observation anomaly scores out of a `isopro` fit so you can plot them, sort them, or write them to disk without having to know the internal shape of the fit.

Usage

```
gg_isopro(object, ..., newdata = NULL)
```

Arguments

`object`	An `isopro` fit returned by `isopro`.
`...`	Currently unused. Present before `newdata` so that `newdata` is only matched by name, preserving backward compatibility with callers of the PR #94 signature `gg_isopro(object, ...)`.
`newdata`	Optional `data.frame` of new observations to score against the fit. Must be passed by name. When `NULL` (default) the extractor returns the in-sample tidy frame from the fit's stored `$case.depth` and `$howbad`. When supplied, each row is scored via `predict.isopro` and the same tidy shape is returned for the test data.

Value

A `data.frame` of class `c("gg_isopro", "data.frame")`, one row per observation. Columns:

obs Integer; observation index `1..n`, in the same order as the rows of the data passed to `isopro`.

case.depth Numeric; mean isolation depth across the forest. Lower means the observation was isolated quickly, so more anomalous.

howbad Numeric in `[0, 1]`; the `case.depth` values pushed through their own empirical CDF and flipped so higher means more anomalous. This is the score the plot method draws by default.

A `provenance` attribute records `source = "varPro::isopro"`, the observation count `n`, and the number of trees `ntree`.

What isopro is doing

An isolation forest (Liu, Ting and Zhou 2008) is a random forest grown on very small subsamples of the data and asked to split until each observation lands in its own terminal node. The intuition is geometric: a typical observation sits in the dense middle of the feature cloud and takes many splits to isolate, while an unusual observation sits out near an edge and gets cut off after only a few. So **the depth at which an observation is isolated is a proxy for how typical it is**: shallow depth means anomalous, deep depth means ordinary. Average a single observation's depth across many trees and the noise washes out, leaving a stable per-observation rank.

`isopro` supports three flavours of isolation forest, which differ in how the splits are chosen:

`"rnd"` The original Liu/Ting/Zhou method: each tree node picks a variable at random and a split point uniformly at random in the variable's range. Fast, no model, surprisingly effective.

`"unsupv"` Unsupervised splitting from `randomForestSRC`: at each split a random subset of the features serves as pseudo-responses (`ytry` of them, about the square root of the number of features by default) and the split is chosen to separate those pseudo-responses. More structured than `"rnd"`; sometimes more accurate, especially when the anomalies follow a coherent direction.

`"auto"` An auto-encoder formulation that grows a multivariate forest predicting each feature from the others. Most expressive, slowest, best suited to low-dimensional data.

No method is universally best. The varPro authors recommend trying at least two and comparing the score distributions; the plot method here colors per-method curves automatically when you stack the results.

What's in the output

The fit gives back two parallel per-observation vectors: `case.depth` is the raw mean isolation depth (units of "splits", lower = more anomalous) and `howbad` is the same information on a `[0, 1]` scale, the empirical CDF of `case.depth`. On the fit those two run the same way: `varPro::isopro()`'s `howbad` is *lower* = more anomalous, like the depth it came from.

We flip it. `gg_isopro()` returns `1 - object$howbad`, so the column you get is **higher = more anomalous** and reads the way a score should. That is our transformation, not the fit's, and it means `gg$howbad` is `1 - fit$howbad` rather than a copy of it, worth knowing if you compare the two side by side. Both columns are kept so you can plot in either space and have the raw depth on hand for diagnostics; `howbad` is the canonical score and is what the plot method uses by default.

What you use this for

This is screening, not inference. Reach for it when you want to:

- flag observations that may be data-entry errors, out-of-range measurements, or distinct sub-populations before fitting a primary model;
- check whether a held-out cohort sits inside the training distribution before scoring with a model trained elsewhere;
- give the analyst a ranked list of "look at these first" cases for a manual review;
- score a held-out cohort or a fresh batch of incoming data against a fitted model and compare the test scores to the training distribution.

The score is a *rank*, not a probability of being an outlier: two observations with `howbad = 0.92` are both unusual, not "92% likely to be anomalous". Pick a cutoff by looking at where the elbow rises; `plot.gg_isopro` can annotate either a score (`threshold`) or a top-percent (`top_n_pct`) for you.

Scoring new data

Pass a `data.frame` as `newdata` and the extractor calls `predict.isopro` twice: once with `quantiles = FALSE` to get the raw mean case depth per row, and once with `quantiles = TRUE` to get the per-row quantile of that depth against the training-data depth distribution.

varPro's `predict.isopro` returns quantiles where *smaller is more anomalous*, which is the opposite polarity of the wrapper's `howbad` (where *higher* is more anomalous). The wrapper exposes both conventions so nothing is hidden:

- `case.depth` carries varPro's native polarity, *lower* = *more anomalous*. This is the unmodified output of `predict(object, newdata, quantiles = FALSE)`. Use it to cross-reference against raw varPro output.
- `howbad` is the flipped, wrapper-convention version. The relationship is `howbad = 1 - predict(object, newdata, quantiles = TRUE)`.

To overlay training and test scores in one plot, bind the two extractor calls with a `method` label column (the same column `plot.gg_isopro` uses to color rnd / unsupv / auto comparisons):

```
gg_train <- gg_isopro(fit)
gg_test  <- gg_isopro(fit, newdata = test_df)
gg_both  <- rbind(cbind(gg_train, method = "train"),
                  cbind(gg_test,  method = "test"))
class(gg_both) <- c("gg_isopro", "data.frame")
plot(gg_both)
```

Comparing methods

To compare methods (`"rnd"`, `"unsupv"`, `"auto"`), call `gg_isopro` on each fit and `dplyr::bind_rows()` the results with a `method` label column. The plot method auto-detects `method` and colors the curves.

References

Liu, F. T., Ting, K. M., and Zhou, Z. H. (2008). Isolation Forest. *Eighth IEEE International Conference on Data Mining*, 413-422.

Ishwaran H, Kogalur U (2026). *varPro: Model-Independent Variable Selection via the Rule-Based Variable Priority*. R package version 3.2.0. https://CRAN.R-project.org/package=varPro

See Also

plot.gg_isopro, isopro

Examples

```
if (requireNamespace("varPro", quietly = TRUE)) {
  set.seed(1)
  fit <- varPro::isopro(data = iris[, 1:4], method = "rnd",
                        sampsize = 32, ntree = 50)
  gg <- gg_isopro(fit)
  plot(gg)
}
```

---

gg_ivarpro *Individual (local) variable importance from a varPro fit*

Description

Tidy wrapper around varPro::ivarpro() for the regression or classification family. Returns one row per (observation, variable) pair where the local-importance cell is non-NA; classification adds a `class` column. `which_obs` collapses to a per-observation profile; `which_class` collapses to a single class. Optional `ivarpro_fit` argument lets callers cache the expensive `ivarpro()` call.

Usage

```
gg_ivarpro(
  object,
  ...,
  which_obs = NULL,
  which_class = NULL,
  cutoff = NULL,
  ivarpro_fit = NULL
)
```

Arguments

object	A varpro fit from varPro::varpro() (regression or classification family).
...	Forwarded to varPro::ivarpro() when ivarpro_fit = NULL; ignored otherwise (with a warning). Documented forwardables: adaptive, cut, cut.max, ncut, nmin, nmax, noise.na, max.rules.tree, max.tree, use.loo, use.abs, scale.
which_obs	Optional integer scalar; 1-based row index into the training data. NULL (default) returns the aggregate view.
which_class	Optional response level name. NULL default on a binary classification fit resolves to the last factor level (positive-class convention), or to the one level that has a profile when the fit was not mirrored. Ignored with a warning on regression fits.
cutoff	Selection threshold on |local_imp|. NULL (default) resolves to the per-class mean(|local_imp|) (or per-frame mean for regression). A numeric scalar broadcasts. A named numeric vector (names a subset of class levels) overrides per class with fallback to the per-class mean for missing names.
ivarpro_fit	Optional pre-computed varPro::ivarpro() result for the same object. Shape-validated.

Value

A data.frame of class c("gg_ivarpro", "data.frame"). Regression: columns obs / variable / local_imp / selected. Classification: long-format with an extra class column. variable is a factor whose levels run in ascending mean(|local_imp|) across all rows, most important last (the unified ranking axis shared across facets / panels).

What this is doing

The varPro framework builds importance from release rules: for a given rule region, it compares a local estimator inside that region to what the estimator becomes after the constraint on the tested variable is removed ("released"). That contrast is summed over many rules and trees to get a global z-score: the quantity gg_varpro() shows. What ivarpro() adds is a per-observation view of the same mechanism.

Concretely: ivarpro() walks the forest's rules and, for each (observation, variable) pair, computes a scaled per-rule contribution to predicting that observation. Per-rule LOO removes the observation from its own rule before scoring, so the contribution is not inflated by the observation having helped define the region. Per-region scaling (scale = "local", default) standardizes the contribution by the rule's local response standard deviation so values are comparable across rules of different size. Aggregating those per-rule scores into one number per (obs, variable) pair gives the local_imp cell.

No permutation, no synthetic data: the contrast is always between real subsets of the observed data, defined by the forest's own rules. This is the same no-synthetic-features property that distinguishes gg_varpro() from gg_vimp()'s Breiman-Cutler permutation importance.

What local_imp actually is (pedantic)

local_imp[i, v] is the **scaled aggregated rule contribution** of variable v to predicting observation i, NOT a permutation importance and NOT a SHAP value. **Sign carries direction** of the

local response shift inside the rule's region. **Magnitude is on the response scale** when `scale = "global"`, or unit-free when `scale = "local"` (the default). The matrix is **heavily sparse**: an observation contributes only to rules that retain it as OOB; on real data, per-variable NA fractions of 50-95% are common. Comparison with `gg_varpro()` (aggregate split-strength) and `gg_beta_varpro()` (per-rule lasso beta) is diagnostic: a variable that's important globally but has low per-observation contribution for a specific case is interesting; the inverse (high local but low global) flags a regime-specific signal.

What's in the output

Long-format tidy frame. Regression has columns `obs`, `variable`, `local_imp`, `selected`. Classification adds a `class` column (factor in response-level order). `variable` is a factor whose levels run in ascending `mean(|local_imp|)` across all rows, so the most important variable is the last level and plots at the top; for classification that aggregate is across all (obs, class) so every facet / panel shows variables in the same row order. NA cells are filtered out; the source matrix is sparse, and the tidy frame only carries the cells where local importance is defined.

Provenance attribute carries `source`, `family`, `ntree`, `cutoff` (named numeric vector: length 1 named `"regr"` for regression, length K named with class levels for classification), `cutoff_default`, `use.loo`, `scale`, `n_train`, `n_obs`, `n_var`, `precomputed`, `xvar.names`, `class_levels` (classification only), `which_obs`, `which_class`.

What you use this for

Per-observation interpretation ("which variables drive *this* prediction?"), variable-selection diagnostics via the aggregate distribution view, and side-by-side comparison against `gg_varpro()` / `gg_beta_varpro()` to spot variables that matter locally but not globally (or vice versa).

Caching

`ivarpro()` is **the most expensive call in varPro** (per-rule leave-one-out + per-region scaling, often minutes on real data). Compute it once and reuse:

```
v   <- varPro::varpro(medv ~ ., data = Boston, ntree = 200)
iv  <- varPro::ivarpro(v, scale = "local")              # expensive, once
gg_aggregate <- gg_ivarpro(v, ivarpro_fit = iv)          # cheap
gg_case1     <- gg_ivarpro(v, ivarpro_fit = iv, which_obs = 1L)
```

Provenance carries `precomputed = TRUE` when `ivarpro_fit` was supplied.

Classification

For a classification fit, `ivarpro()` returns a list of K matrices (one per class) for multi-class, or a flat data.frame for binary. The binary frame explains one class, named by its `target` attribute and by default the first factor level. Because the two class probabilities sum to one, the wrapper derives the other class exactly, as the negated local slopes (or the same values under `use.abs = TRUE`), so both levels are available. A `y.external` profile is not a class probability and is not mirrored: it is kept as a single class, under its target's name when that is a level and under the last level otherwise. The wrapper stacks per-class frames into a long-format frame with a `class` column. `which_class = NULL` returns all classes (binary defaults to the last factor level, the positive-class convention used by

glm and gg_roc); which_class = "<name>" filters to a single class. cutoff polymorphism mirrors gg_beta_varpro(): NULL is per-class mean(|local_impl|), a scalar broadcasts, a named numeric vector overrides per class with fallback to that class's mean.

Reproducibility

Byte-for-byte agreement between cached (ivarpro_fit = iv) and uncached (ivarpro_fit = NULL) outputs requires reusing the same ivarpro() result. set.seed() alone is not sufficient because per-rule LOO subsampling can drift across separate calls. Reuse ivarpro_fit when reproducibility matters.

Note

Multivariate regression (regr+) and survival fits are not supported. The unsupported-family path reports the family it received.

See Also

gg_varpro(), gg_vimp(), gg_beta_varpro(), varPro::ivarpro().

Examples

```
if (requireNamespace("varPro", quietly = TRUE) &&
    requireNamespace("MASS", quietly = TRUE)) {
  set.seed(1)
  v <- varPro::varpro(medv ~ ., data = MASS::Boston, ntree = 50)
  iv <- varPro::ivarpro(v)
  gg <- gg_ivarpro(v, ivarpro_fit = iv)
  plot(gg)
}
```

---

gg_partial *Split partial dependence data into continuous or categorical datasets*

Description

A partial dependence curve answers a what-if question about a forest: hold every other predictor at its observed value, sweep one of them across its range, and watch how the ensemble prediction moves. Marginalized over the joint distribution of the other variables, the resulting curve isolates the average effect of the swept predictor alone.

Usage

```
gg_partial(part_dta, nvars = NULL, cat_limit = 10, model = NULL)
```

Arguments

part_dta	partial plot data from randomForestSRC::plot.variable
nvars	how many of the partial plot variables to calculate
cat_limit	Categorical features are built when there are fewer than cat_limit unique feature values.
model	a label name applied to all features. Useful when combining multiple partial plot objects in figures.

Details

gg_partial handles the bookkeeping step after you've already called randomForestSRC::plot.variable(partial = TRUE): it takes the list that function returns and separates the variables into two tidy data frames, one for continuous predictors (plotted as lines) and one for categorical predictors (plotted as box plots). The split is controlled by cat_limit: variables with more unique x-values than this threshold are treated as continuous; all others are categorical.

If you'd rather skip the plot.variable step and pass the fitted forest directly, see gg_partial_rfsrc, which calls partial.rfsrc for you.

Value

A named list with two elements:

continuous data.frame with columns x, yhat, name (and optionally model) for continuous variables

categorical data.frame with the same columns but with x as a factor, for low-cardinality / categorical variables. Unlike continuous, it is not averaged: plot.variable() returns one prediction per training observation per level, and plot.gg_partial draws their spread as box plots. Average yhat within x for the partial dependence value.

Note

Partial-dependence extraction is randomForestSRC-only; there is no randomForest method (the randomForest package provides no comparable partial-dependence interface).

For survival forests, randomForestSRC::plot.variable defaults to surv.type = "mort", so yhat is *mortality* (the expected number of events) and not a survival probability. It is therefore not on $[0, 1]$ and is not directly comparable with the survival probabilities returned by gg_variable. For a comparable quantity, ask for it explicitly: randomForestSRC::plot.variable(rf, partial = TRUE, surv.type = "surv"). The label describing the plotted quantity is recorded on the returned object as attr(x, "ylabel") and is used as the y-axis title by plot.gg_partial.

gg_partial_rfsrc defaults to partial.type = "surv" and so reports survival probabilities. The two entry points therefore report different quantities by default.

See Also

gg_partial_rfsrc gg_partialpro

Examples

```
## Build a small regression forest on the airquality dataset
set.seed(42)
airq <- na.omit(airquality)
rf <- randomForestSRC::rfsrc(Ozone ~ ., data = airq, ntree = 50)

## Compute partial dependence via plot.variable (show.plots = FALSE to
## suppress the base-graphics output, we only want the data)
pv <- randomForestSRC::plot.variable(rf, partial = TRUE,
                                     show.plots = FALSE)

## Split into continuous and categorical data frames
result <- gg_partial(pv)
head(result$continuous)

## Label this model for later comparison with a second forest
result_labeled <- gg_partial(pv, model = "airq_model")
unique(result_labeled$continuous$model)
```

`gg_partial_rfsrc` *Partial dependence data from an rfsrc model*

Description

A partial dependence curve marginalizes the forest's prediction over all other predictors: for each evaluation point of the target variable, the forest scores every training observation with that value substituted in, then averages the result. What you get is the average effect of the target variable after "integrating out" the rest, a curve that would be flat if the variable carried no signal.

Usage

```
gg_partial_rfsrc(
  rf_model,
  xvar.names = NULL,
  xvar2.name = NULL,
  newx = NULL,
  partial.time = NULL,
  partial.type = c("surv", "chf", "mort"),
  cat_limit = 10,
  n_eval = 25
)
```

Arguments

`rf_model`	A fitted `rfsrc` object.
`xvar.names`	Character vector of predictor names for which partial dependence should be computed. Must be a subset of `rf_model$xvar.names`.

`xvar2.name` Optional single character name of a grouping variable in `newx`. When supplied, partial dependence is computed separately for each unique value of this column in `newx`, with the variable held at that value for every training observation, and a `grp` column is appended. For a continuous variable, set `newx[[xvar2.name]]` to a short grid first, since every distinct value costs one more `partial.rfsrc()` call for each variable in `xvar.names`. A factor keeps its level labels in `grp`, as a factor in the model's level order.

`newx` Optional `data.frame` that sets the evaluation grid, not the data being averaged over. Each of `xvar.names` is evaluated at the quantile grid (or unique levels) of its column in `newx`, and `xvar2.name` takes its values from the same place. The average is always taken over the training data held in `rf_model` (see `partial.rfsrc`), so overwriting any other column of `newx` has no effect on `yhat`; to hold a second variable fixed, pass it as `xvar2.name`. Defaults to `rf_model$xvar`. All column names must match `rf_model$xvar.names`.

`partial.time` Numeric vector of desired time points for survival forests (ignored for regression/classification). Values are automatically snapped to the nearest entry in `rf_model$time.interest`; see the **Survival forests** section below. When `NULL` (default), three quartile points of `time.interest` are used. Has no effect with `partial.type = "mort"`.

`partial.type` Character; type of predicted value for survival forests, passed through to `partial.rfsrc`. One of `"surv"` (default), `"chf"`, or `"mort"`. Ignored for non-survival forests. `partial.rfsrc()` requires a non-`NULL` value for survival families; supplying it here avoids a cryptic "argument is of length zero" error from the underlying C code.

`cat_limit` Variables with fewer than `cat_limit` unique values in `newx` are treated as categorical; all others are continuous. Defaults to 10.

`n_eval` Number of evaluation points for continuous variables. Instead of passing all observed values (which can be slow, especially for survival forests), continuous predictors are evaluated on a quantile grid of this many points. Categorical variables always use all unique levels. Defaults to 25.

Details

This function builds those curves for one or more predictors by calling `partial.rfsrc` and then tidy-stacking the results into separate data frames for continuous and categorical variables. Unlike `gg_partial` (which wraps `plot.variable`), you pass the fitted `rfsrc` object directly, with no intermediate `plot.variable` step.

For survival forests, the marginalized quantity depends on `partial.type`: survival probability (`"surv"`), cumulative hazard function (`"chf"`), or expected mortality (`"mort"`). You can request the curve at one or more time horizons via `partial.time`; the resulting data have a `time` column so the plot layers them as separate colored lines. Mortality is the exception: it is summed over every event time, so `partial.time` has no effect on it and the data carry no `time` column.

Value

A named list with two elements:

continuous A data.frame with columns x (numeric), yhat, name (variable name), and optionally grp (the level of xvar2.name) and time (survival forests with partial.type "surv" or "chf") for all continuous predictors.

categorical A data.frame with the same columns but x kept as a factor (levels in the model's level order, not alphabetical), for low-cardinality predictors. Unlike continuous, it is not averaged: there is one row per training observation per level (and per time point), so plot.gg_partial_rfsrc draws it as box plots. Average yhat within x for the partial dependence value.

Survival forests and partial.time

partial.rfsrc expects every value in partial.time to be an exact member of the model's time.interest vector, the unique observed event times stored in the fitted object. Pass an arbitrary time, even a plausible one such as c(1, 3) for a study measured in years, and you get a C-level prediction error from inside partial.rfsrc.

gg_partial_rfsrc takes care of this: every element of partial.time is silently snapped to its nearest time.interest value before the call. To target a specific follow-up horizon, find the closest grid point yourself and pass it explicitly:

```
ti  <- rf_model$time.interest
t1  <- ti[which.min(abs(ti - 1))]   # nearest to 1 year
pd  <- gg_partial_rfsrc(rf_model, xvar.names = "x", partial.time = t1)
```

The snap is silent, and it matters most before the first event: every request earlier than the first event time returns that first event's curve, so partials at, say, 10, 30 and 60 days come back identical in a sample whose first death is later. Check the time column of the result, and choose horizons after the first event.

Logical predictor columns

partial.rfsrc does not handle logical predictor columns correctly in survival forests (randomForestSRC <= 3.5.1). If your training data contains binary 0/1 columns, convert them to factor rather than logical before fitting the model.

Note

For survival forests this function defaults to partial.type = "surv", so yhat is a survival probability on $[0, 1]$. gg_partial wraps randomForestSRC::plot.variable instead, which defaults to surv.type = "mort" and therefore yields *mortality* (an expected event count). The two entry points consequently report different quantities by default; check the y-axis label, which each records on the returned object.

See Also

gg_partial, partial.rfsrc, get.partial.plot.data

Examples

```
## ------------------------------------------------------------
##
## regression
##
## ------------------------------------------------------------

airq.obj <- randomForestSRC::rfsrc(Ozone ~ ., data = airquality, ntree = 100)

## partial effect for wind
prt_dta <- gg_partial_rfsrc(airq.obj,
                      xvar.names = c("Wind"))
```

`gg_partial_varpro` *Partial dependence data from a varPro model*

Description

`varPro::partialpro` returns one list, with continuous and categorical predictors mixed together. This function splits that list into two tidy data frames, one for each kind, and resolves the y-axis label the plot method will use.

Deprecated. `gg_partialpro()` has been superseded by `gg_partial_varpro()` and is now a thin alias for it. It will be removed in a future release; use `gg_partial_varpro()` directly.

Usage

```
gg_partial_varpro(
  part_dta = NULL,
  object = NULL,
  scale = c("auto", "prob", "prob_typical", "odds", "logodds", "rmst", "surv",
    "mortality", "chf"),
  time = NULL,
  nvars = NULL,
  cat_limit = 10,
  model = NULL,
  ...
)

gg_partialpro(
  part_dta,
  object = NULL,
  scale = c("auto", "prob", "prob_typical", "odds", "logodds", "rmst", "surv",
    "mortality", "chf"),
  time = NULL,
  nvars = NULL,
  cat_limit = 10,
```

```
  model = NULL,
  ...
)
```

Arguments

part_dta	Partial plot data from varPro::partialpro. Each element must contain xvirtual, xorg, yhat.par, yhat.nonpar, and yhat.causal. Supply at least one of part_dta or object.
object	A fitted varpro object, the forest the partial data came from. When supplied it provides the provenance metadata, and when part_dta is NULL it is passed to varPro::partialpro(object) for you. Required when scale %in% c("surv","chf").
scale	Character; the y-axis scale. One of "auto" (default), the classification scales "prob" / "prob_typical" / "odds" / "logodds", or the survival scales "rmst" / "surv" / "mortality" / "chf". With "auto": classification fits resolve to "prob" (probability of the target class) and survival fits to "surv" (survival probability at a default horizon τ); see **Details**.
time	Numeric; the evaluation time point. Required when scale = "rmst" (the RMST horizon τ), where it now *drives* the partial computation through an RMST(τ) learner (see **Details**), not just the axis label. Optional when scale = "surv": it is the horizon τ of the $S(\tau)$ learner, read at the nearest value in object\$rf\$time.interest; if NULL, τ defaults to the median follow-up time, with a message. Optional when scale = "chf": if supplied it is snapped to the nearest value in object\$rf\$time.interest and used for both computation and axis labeling; if NULL, three quartile time points from time.interest are used (see gg_partial_rfsrc).
nvars	Integer; how many variables to keep, selected as the top nvars **by varPro importance** when object is supplied (falling back to the incoming part_dta list order otherwise). Defaults to every variable in part_dta. See **Details**.
cat_limit	Integer; a variable with length(xvirtual) <= cat_limit is treated as categorical. Default 10.
model	Character; a label tacked onto every row, handy when you are combining results from several models in one figure.
...	Forwarded to partialpro on the object-driven path (when part_dta is NULL). Use this to control which variables are computed (e.g. xvar.names or nvar) or to tune the isolation-forest UVT step (cut, nsmp). Without it, partialpro falls back to varPro::get.topvars(object), which can return few or no variables for some fits (yielding empty continuous/categorical frames). A name you pass in xvar.names that the fit cannot reach is dropped by partialpro, so you can ask for twelve variables and get ten; partialpro warns and names the missing ones. See **Details**. Ignored, with a warning, when part_dta is supplied. With scale = "chf" the work goes through gg_partial_rfsrc rather than partialpro, so xvar.names is honored but the partialpro-only arguments (cut, nsmp) do not apply and are ignored with a warning.

Details

Which variables you actually get: varpro screens twice before anything reaches partial dependence. The split-weight screen decides which predictors are worth guiding the trees with, and what

survives it lands in `object$xvar.names`; `varPro::get.topvars` then ranks a shorter list out of that. So the design matrix, the reachable set, and the default list are three different sizes; a fit on 45 predictors might carry 26 in `object$xvar.names` and 15 in `get.topvars`. `partialpro` can only reach the middle one. How much gets screened off depends on the data and the fit, so check rather than assume: `length(object$xvar.names)` against `ncol(object$x)` tells you where you stand.

This bites when you bring a variable list in from somewhere else, say the top names off an `rfsrc` VIMP ranking. `partialpro` intersects your `xvar.names` with what it can reach and keeps the overlap, so a request for twelve variables can come back with ten. It is the intermittent kind of trap: a top-10 list may come back whole while a top-12 list loses two. `partialpro` warns and names what it dropped, but only after the isolation-forest work is done. A quick `setdiff(my_names, object$xvar.names)` answers the same question before you spend the computation.

For a complete view, fit with both screens off: `varPro::varpro(..., sparse = FALSE, split.weight = FALSE)`. `split.weight = FALSE` is the one that lifts the ceiling. It puts every predictor in `object$xvar.names`, so partial dependence can reach them all, and it leaves a strong variable's curve where it was. `sparse = FALSE` does the smaller thing, deepening `varPro::get.topvars` so the reported ranking shows its tail rather than the screened top. Both are **varPro**'s own arguments, and its defaults (both `TRUE`) go the other way, toward the screened set; keep the defaults when that sparser set is what you want. `nvar` is not the knob here; it only caps how much gets reported.

Which rows you actually get: varPro has no imputation. Every entry point opens by growing a one-node stump through `randomForestSRC::rfsrc` to settle the family and hand back cleaned data, and that call takes `rfsrc`'s default `na.action = "na.omit"`. Any case with a missing value, in a predictor or in the outcome, is deleted before the fit. What you are told about it depends on the **varPro** version. Before 3.2.2 the deletion is silent: no warning, no message, and `na.action = "na.impute"` passed to `varPro::varpro` lands in `...`, never reaches the stump, and is discarded without remark. From 3.2.2, `varpro` warns with the input, omitted and retained counts, records them in `object$model.info$observations`, and stops with an error on `na.action` (or any other argument it does not recognise) instead of ignoring it. The rows are deleted either way; 3.2.2 makes the deletion visible.

The loss compounds across predictors rather than adding up. At 5% missing per column, independently, retention is 0.95^p: 60% of rows at 10 predictors, 36% at 20, 8% at 50. On a wide clinical frame a little missingness everywhere can delete most of the cohort. Before 3.2.2 nothing in the fit records it: `object$rf$n` is the count *after* deletion and no original is kept, so neither you nor this package can recover the number from the object. On any version, check before you fit: `nrow(dta)` against `sum(complete.cases(dta))`.

Imputing first, without inventing outcomes: `varPro::roughfix` does mean and modal fill, `randomForestSRC::impute` does the forest-based job, and varPro's own help pages use the latter. Both fill **every** column they are handed, the outcome included, so where the outcome is itself missing they manufacture it, and the release rules are then fit partly to invented responses. Put the observed outcome back and drop the cases that never had one:

```
imp   <- randomForestSRC::impute(y ~ ., data = dta)
imp$y <- dta$y                    # restore the observed outcome
imp   <- imp[!is.na(imp$y), ]     # drop cases with no real outcome
```

That is von Hippel's impute-then-delete: keep the outcome in the imputation model, since leaving it out attenuates the associations varPro is looking for, then discard the cases whose outcome was imputed, which carry no information about the response. Only the deletion step depends on having

missing outcomes; the imputation matters whenever a *predictor* is missing. Two cautions. Imputing with the outcome stamps its signal into the filled predictor cells, which is right for a single fit but crosses fold boundaries in `varPro::cv.varpro`; impute inside the folds if the selection error has to mean anything. And a completed frame is one dataset, not many, so the curves here carry no uncertainty from the imputation; read them as conditional on it.

`nvars` **selects by importance, not by list position:** when `object` is supplied, the variables in `part_dta` are first reordered by their rank in `varPro::get.topvars(object)`, and `nvars` then keeps the top `nvars` of *that* ordering, not the first `nvars` elements as they arrived. `get.topvars()` typically ranks far fewer variables than `part_dta` contains; any variable it does not rank keeps its incoming order and is appended after the ranked block, so nothing is ever dropped from consideration, only reordered. The returned `continuous$name` / `categorical$name` column is a `factor` whose levels follow this same importance order (raw list order when no `object` was supplied), so facet panels and legend entries sort by importance rather than alphabetically.

Scale detection: with `scale = "auto"` and an `object` in hand, the scale resolves to `"surv"` for a survival forest, `"prob"` for a classification forest and `"generic"` for a regression forest. We do not read an RMST horizon τ off the `varpro` object. Before **varPro** 3.2.2 there is none to read; from 3.2.2 one is recorded only when `varpro` itself was fit with an `rmst` horizon. So RMST output requires you to pass `scale = "rmst", time = tau` explicitly.

RMST partial dependence (scale = "rmst"): `varPro::partialpro` has no time argument, so its default survival learner returns ensemble mortality at every horizon. Before **varPro** 3.2.2 a horizon passed through `...` is silently dropped, and multi-horizon plots built that way differ only by Monte-Carlo noise, not by τ; from 3.2.2 `partialpro` stops with an error instead. To get a genuine RMST(τ) curve, `scale = "rmst"` supplies `partialpro` a `learner` that returns $\mathrm{RMST}(\tau) = \int_0^\tau S(t)\,dt$ from the survival forest, so the curve actually depends on τ. This path **recomputes** from `object`, so it needs `object` (a survival fit) with `part_dta = NULL`; a precomputed `part_dta` can only be relabeled, and `gg_partial_varpro` warns when you try. A τ beyond the model's largest event time is truncated there (with a warning), since $S(t)$ cannot be extrapolated.

Classification scale (scale = "prob"/"odds"/"logodds"): `varPro::partialpro` returns classification effects as *log-odds* of the target class. `scale = "prob"` (the classification default) back-transforms to probability $P(Y = \mathrm{target})$, `"odds"` to the odds, and `"logodds"` keeps the raw scale. The back-transform is applied per observation *before* averaging, so the curve is the mean predicted probability, not the probability of the mean log-odds.

Restoring each subject's level (scale = "prob"): `partialpro` fits each subject's curve separately but returns every row at the cohort-mean intercept, keeping only the subject's own slope. On the log-odds scale that sets every subject to the average log-odds, and averaging per-subject probabilities then no longer gives the expected proportion. So for `"prob"`, `gg_partial_varpro` shifts each subject's curve to pass through that subject's own out-of-bag log-odds at its observed value of the variable, and then back-transforms and averages. The shape `partialpro` fitted is unchanged. This needs `object`, the classification fit, and is recorded as `anchored` in the provenance. Without `object`, or when `...` passes `partialpro` a custom `learner` or `newdata`, the curve is left as `partialpro` returned it (with a warning when `object` is missing). A precomputed `part_dta` is anchored whenever `object` is supplied, which assumes it came from `partialpro(object)` with its default learner; a `part_dta` built with your own learner should be passed with `scale = "logodds"` instead. A subject that was never out of bag takes its in-bag prediction as its anchor. Binary variables are never shifted, because `partialpro` already returns per-subject levels for them. The `causal` contrast is shown only on `"logodds"` (see `plot.gg_partial_varpro`).

Two probability scales, and why they disagree (scale = "prob" vs "prob_typical"): `partialpro`

returns a matrix of per-subject log-odds, one row per observation and one column per grid point. Collapsing it to a curve takes an average and a back-transform, and the *order* of those two steps is a modelling choice, not a detail:

$$\text{prob}(x) = \frac{1}{n}\sum_i \text{logit}^{-1}(z_i(x)) \qquad \text{prob_typical}(x) = \text{logit}^{-1}\left(\frac{1}{n}\sum_i z_i(x)\right)$$

`"prob"` (the classification default) transforms per observation and then averages, so the curve is the **mean predicted probability**, the expected proportion of this cohort, and the standard partial dependence quantity on a probability scale. `"prob_typical"` averages on the log-odds scale and then transforms once, giving the probability for a **subject sitting at the mean log-odds**.

These are different estimands and they do not agree. logit^{-1} is concave above zero and convex below it, so by Jensen's inequality `"prob"` is pulled toward 0.5 relative to `"prob_typical"`, at both ends of the curve. The gap widens with the spread of per-subject log-odds, and on a heterogeneous cohort it is not small. In a simulation where a second variable spreads the subjects' log-odds to an SD near 4, one point reads 0.13 under `"prob_typical"` and 0.35 under `"prob"`, against a true partial dependence of 0.38. Before the level restoration described above, `"prob"` read 0.14 there: with every subject at the mean log-odds, the two scales nearly coincide. `"prob_typical"` uses `partialpro`'s values as returned.

Which to report is a question about the claim, not about the code. If the sentence is "what fraction of these patients would wean", that is `"prob"`. If it is "what would we predict for a typical patient", that is `"prob_typical"`, while remembering that the mean-log-odds subject need not resemble anyone in the data. A figure captioned as a percentage of patients wants `"prob"`.

The distinction applies only to the `continuous` frame. The `categorical` frame keeps its values unaveraged so the plot method can draw boxplots, so there is no averaging order to choose and the two scales return the same numbers there.

Survival probability (scale = "surv"): `scale = "surv"` (the survival default) computes $S(\tau \mid x)$ through `partialpro` (the same UVT engine as mortality and RMST), bounded in $[0, 1]$. `partialpro` smooths each case's $S(\tau)$ with an unbounded polynomial, so where survival is near 1 the averaged curve can run slightly past it; the curve is clamped to $[0, 1]$ after averaging, and the categorical frame per value. The curve is still the average of the smoothed per-case $S(\tau)$, with only impossible values changed. When `time` is not supplied, τ defaults to the **median follow-up time** of the fit, a data-driven horizon that is always in the model's own time units, so it cannot be mis-specified the way a hand-typed τ can. The resolved τ is reported in a message and the axis label; pass `time = tau` to choose another. `scale = "mortality"` keeps the unbounded ensemble-mortality score as an explicit opt-in.

Ensemble mortality (scale = "mortality"): here the y-axis is *ensemble mortality*, the expected number of events a subject would see if they were exposed to the study-average cumulative hazard. It is the same quantity as the `rfsrc` `predicted` value for survival forests (Ishwaran, Kogalur, Blackstone & Lauer, 2008 doi:10.1214/08-AOAS169). This is an **unbounded relative-risk score**, *not* a survival probability and not $1 - S(t)$; don't read it as one. For a bounded, time-anchored survival summary, use `scale = "rmst", time = tau` (restricted mean survival time, in the time units of the outcome) or `scale = "surv"` / `"chf"`.

Arguments are documented on gg_partial_varpro; this alias shares its formals and forwards every argument unchanged.

Value

A named list of class "gg_partial_varpro" with elements:

continuous data.frame with columns variable, parametric, nonparametric, causal, name (and optionally model).

categorical data.frame with the same columns, one row per observation per category level.

A "provenance" attribute carries source, family, ntree, n, scale, rmst_tau, xvar.names, and path.

A gg_partial_varpro object (see gg_partial_varpro).

What partialpro is doing

A partial dependence curve answers the question, "if I hold a single variable at a grid of values and average out everything else, how does the model's prediction move?" That is the same question rfsrc partial dependence answers. What varPro::partialpro adds is two wrinkles that are worth understanding before you read the curves.

First, partialpro filters the partial grid through an isolation forest (Unlimited Virtual Twins, or UVT) so that unlikely combinations of the focal variable with the rest of the data are downweighted. The rfsrc version, by contrast, averages over the full marginal grid regardless of plausibility. So when a covariate is highly correlated with others, the two methods can disagree, and partialpro's curve is the one restricted to the data manifold.

That isolation forest is worth one note. partialpro grows it with varPro::isopro, whose method defaults to "unsupv", a forest with no outcome. randomForestSRC then passes a zero-length yvar.wt into its native code and decrements that pointer (entry.c:184), which is undefined behaviour and is reported by UBSAN builds. It is benign in practice (the pointer is formed but never dereferenced), and it is an upstream issue rather than one this package can fix (ggRandomForests is pure R); it is tracked as kogalur/randomForestSRC issue #477. Passing method = "rnd" through to partialpro avoids the unsupervised grow entirely, which is what this package's own tests and examples do.

Second, partialpro fits a local polynomial model to the predicted values rather than just plotting their mean. That gives three parallel curves per variable, stored as yhat.par, yhat.nonpar, and yhat.causal, which the plot method overlays so you can see whether a smooth parametric story and the raw forest predictions are telling you the same thing.

Interpretation of the y-axis depends on the outcome (per varPro::partialpro): response scale for regression, log-odds of the target class for classification, and either ensemble mortality (default) or RMST (if the original varpro call set rmst) for survival.

What's in the output

We split partialpro's mixed list into two tidy data frames so the plot method does not have to. A variable with more than cat_limit distinct grid points goes into $continuous, one row per grid point with the column means of yhat.par, yhat.nonpar, and yhat.causal stored as parametric, nonparametric, and causal. A variable at or below cat_limit goes into $categorical, one row per observation per category level, carrying the same three columns unaveraged so the plot method can draw boxplots. Path C (scale = "chf") takes a different route: we hand the underlying rfsrc forest to gg_partial_rfsrc so you get a cumulative-hazard curve on the usual rfsrc scale instead. scale = "surv" stays on the partialpro path, through an $S(\tau)$ learner (see **Details**).

What you use this for

- read the marginal shape of a relationship the varpro model found important: monotone, threshold, U-shape, flat;
- compare the three partialpro estimators on the same variable and flag the ones where parametric and nonparametric disagree, those are the candidates for closer inspection;
- report a survival partial dependence on the probability or cumulative-hazard scale (`scale = "surv"` or `"chf"`) rather than the unbounded mortality scale.

A varpro partial dependence curve is a description of the model, not a causal effect. The `causal` column is varpro's local estimator, not a structural causal claim about the data-generating process.

Factors, bands and small data

varPro one-hot encodes a factor, so a factor `stage` becomes the terms `stage1`, `stage2`, ..., and `partialpro()` looks terms up by those names (pass `xvar.names = "stage4"`, not `"stage"`). Each level is partialled on its own, every case set to 0 and to 1 on that level. `partialpro()` leaves the factor's other level columns as they were, so for a factor with three or more levels some of those cases hold two levels at once; this is upstream behaviour in `varPro::partialpro()`. Two-level factors, stored as a single 0/1 column, are not affected. To check a multi-level factor, use `gg_partial_rfsrc`, which sets the factor itself to each level.

The spread across cases is not a confidence interval. It shows how much cases differ under one fitted forest; varPro's is narrower still because `partialpro()` gives every case the cohort-mean intercept. To see how uncertain the curve is, refit on subsamples. The `varpro-small-data` vignette shows both.

References

von Hippel PT (2007). Regression with missing Ys: An improved strategy for analyzing multiply imputed data. *Sociological Methodology*, **37**(1), 83–117. doi:10.1111/j.14679531.2007.00180.x.

Ishwaran H, Kogalur UB, Blackstone EH, Lauer MS (2008). Random survival forests. *The Annals of Applied Statistics*, **2**(3), 841–860. doi:10.1214/08AOAS169.

Ishwaran H, Blackstone EH (2025). Harnessing the power of virtual (digital) twins: Graphical causal tools for understanding patient and hospital differences. *Computational and Structural Biotechnology Journal*, **28**, 312.

See Also

`plot.gg_partial_varpro`, `gg_varpro`, `gg_vimp`, `gg_partialpro` (deprecated), `gg_partial_rfsrc`, `varpro_feature_names`

`gg_partial_varpro`

Examples

```
set.seed(42)
n_obs <- 30; n_pts <- 15
mock_data <- list(
  age = list(
    xvirtual    = seq(30, 80, length.out = n_pts),
```

```
    xorg        = sample(seq(30, 80, by = 5), n_obs, replace = TRUE),
    yhat.par    = matrix(rnorm(n_obs * n_pts), nrow = n_obs),
    yhat.nonpar = matrix(rnorm(n_obs * n_pts), nrow = n_obs),
    yhat.causal = matrix(rnorm(n_obs * n_pts), nrow = n_obs)
  ),
  sex = list(
    xvirtual    = c(0, 1),
    xorg        = sample(c(0, 1), n_obs, replace = TRUE),
    yhat.par    = matrix(rnorm(n_obs * 2), nrow = n_obs),
    yhat.nonpar = matrix(rnorm(n_obs * 2), nrow = n_obs),
    yhat.causal = matrix(rnorm(n_obs * 2), nrow = n_obs)
  )
)
## The two probability scales differ by the ORDER of averaging and
## back-transform, and disagree whenever subjects are heterogeneous.
## Mock data has no fit, so "prob" cannot restore subject levels and warns.
pa <- suppressWarnings(gg_partial_varpro(mock_data, scale = "prob"))
pt <- gg_partial_varpro(mock_data, scale = "prob_typical")
head(data.frame(prob = pa$continuous$parametric,
                prob_typical = pt$continuous$parametric))

result <- gg_partial_varpro(mock_data, scale = "logodds")
head(result$continuous)
head(result$categorical)


## The object-driven path: hand gg_partial_varpro() the varpro fit and let
## it call partialpro() for you.  This is where the reachability ceiling
## shows up (see Details).
set.seed(42)
vp <- varPro::varpro(mpg ~ ., data = mtcars, ntree = 50)

## Three different sizes.  partialpro() can only reach the second.
ncol(vp$x)                    # predictors in the data
length(vp$xvar.names)         # what the fit reaches
length(varPro::get.topvars(vp))   # the default when xvar.names is absent

## Say these came from an rfsrc VIMP ranking.  Check what the fit cannot
## reach before you spend the computation -- this is the habit worth having.
wanted <- c("wt", "hp", "qsec", "vs")
setdiff(wanted, vp$xvar.names)

## Ask anyway and partialpro() warns, naming what it dropped.
## (method = "rnd" is passed through to partialpro(); see the note on
## isolation-forest method in Details.)
pd <- gg_partial_varpro(object = vp, xvar.names = wanted,
                        method = "rnd")

## Refitting without the split-weight screen reaches every predictor.
vp_all <- varPro::varpro(mpg ~ ., data = mtcars, ntree = 50,
                         split.weight = FALSE)
length(vp_all$xvar.names)
setdiff(wanted, vp_all$xvar.names)   # empty; nothing to drop
```

`gg_rfsrc.rfsrc` *Predicted response data object*

Description

Every tree in a random forest makes its own prediction, and the forest's "ensemble" prediction is the average across all trees. The out-of-bag (OOB) variant averages only over the trees that did not include a given observation in their bootstrap sample, a built-in cross-validation estimate that requires no held-out test set. `gg_rfsrc` pulls those ensemble predictions out of the fitted forest and arranges them for plotting with `plot.gg_rfsrc`.

Usage

```
## S3 method for class 'rfsrc'
gg_rfsrc(object, oob = TRUE, by, ...)

gg_rfsrc(object, oob = TRUE, by, ...)
```

Arguments

`object`	A fitted `rfsrc` or `randomForest` object.
`oob`	Logical; if `TRUE` (default) return out-of-bag predictions. Set to `FALSE` to use full in-bag (training) predictions. Forced to `FALSE` automatically for `predict.rfsrc` objects, which carry no OOB estimates.
`by`	Optional stratifying variable. Either a character column name present in the training data, or a vector/factor of the same length as the training set. When supplied, a `group` column is added to the returned data and bootstrap CI bands (survival) are computed per group. Omit or leave missing to return an unstratified result.
`...`	Additional arguments controlling output for specific forest families: **surv_type** Character; one of `"surv"` (default), `"chf"`, or `"mortality"` for survival forests. **conf.int** Numeric coverage probability (e.g. `0.95`) to request bootstrap pointwise confidence bands for survival forests. Triggers wide-format output with `lower`, `upper`, `median`, and `mean` columns. **bs.sample** Integer; number of bootstrap resamples when `conf.int` is set. Defaults to the number of observations.

Details

The structure of the returned data depends on the forest family. For a regression forest you get a scatter of OOB predicted values against the observed response. For classification you get predicted class probabilities alongside the observed class label. For a survival forest you get the ensemble survival function (or cumulative hazard, or mortality, controlled by `surv_type`) at each unique event time (one curve per observation), which together trace the range of predicted risk in the cohort. Pass `conf.int` to add pointwise bootstrap confidence bands around the mean survival curve, or by to stratify all of the above by a predictor group.

For survival forests, use the `surv_type` argument (`"surv"`, `"chf"`, or `"mortality"`) to select the predicted quantity. Bootstrap confidence bands are requested by passing `conf.int` (e.g. `conf.int = 0.95`); the number of resamples is controlled by `bs.sample`.

Value

A `gg_rfsrc` object (a classed `data.frame`) whose structure depends on the forest family:

regression Columns `yhat` and the response name; optionally a `group` column when by is supplied.

classification One column per class with predicted probabilities; a `y` column with observed class labels; optionally `group`.

survival (no CI / grouping) Long-format with columns `variable` (event time), `value` (survival probability), `obs_id`, and `event`.

survival (with `conf.int` or by) Wide-format with pointwise bootstrap CI columns (`lower`, `upper`, `median`, `mean`) per time point; a `group` column when by is supplied.

The object carries class attributes for the forest family so that `plot.gg_rfsrc` dispatches correctly.

See Also

`plot.gg_rfsrc`, `rfsrc`, `gg_survival`

Examples

```
## ------------------------------------------------------------
## classification example (small, runs on CRAN)
## ------------------------------------------------------------
## -------- iris data
set.seed(42)
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris, ntree = 50)
gg_dta <- gg_rfsrc(rfsrc_iris)
plot(gg_dta)


## ------------------------------------------------------------
## Additional regression / survival examples are guarded with
## \donttest because the cumulative example time exceeds the
## 10-second CRAN budget. Run locally with `R CMD check --run-donttest`
## (or `devtools::check(run_dont_test = TRUE)`) to exercise them.
## ------------------------------------------------------------

## -------- air quality data (regression)
```

```
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ ., data = airquality,
                    na.action = "na.impute", ntree = 50)
plot(gg_rfsrc(rfsrc_airq))

## -------- Boston data (rfsrc + randomForest)
if (requireNamespace("MASS", quietly = TRUE)) {
  data(Boston, package = "MASS")
  Boston$chas <- as.logical(Boston$chas)
  rfsrc_boston <- randomForestSRC::rfsrc(medv ~ ., data = Boston, ntree = 50,
                       forest = TRUE, importance = TRUE,
                       tree.err = TRUE, save.memory = TRUE)
  plot(gg_rfsrc(rfsrc_boston))

  rf_boston <- randomForest::randomForest(medv ~ ., data = Boston,
                                          ntree = 50)
  plot(gg_rfsrc(rf_boston))
}

## -------- mtcars data
rfsrc_mtcars <- randomForestSRC::rfsrc(mpg ~ ., data = mtcars, ntree = 50)
plot(gg_rfsrc(rfsrc_mtcars))

## -------- veteran data (survival; with CI and group-by)
data(veteran, package = "randomForestSRC")
rfsrc_veteran <- randomForestSRC::rfsrc(Surv(time, status) ~ ., data = veteran,
                      ntree = 50)
plot(gg_rfsrc(rfsrc_veteran))
plot(gg_rfsrc(rfsrc_veteran, conf.int = .95))
plot(gg_rfsrc(rfsrc_veteran, by = "trt"))
```

`gg_rhf` *Tidy hazard and cumulative-hazard curves from a Random Hazard Forest*

Description

Extracts case-specific ensemble hazard and cumulative-hazard estimates from a fitted `randomForestRHF::rhf()` object into a tidy long data frame, one row per (case, time) pair on the forest's `time.interest` grid.

Usage

```
gg_rhf(object, ...)

## S3 method for class 'rhf'
gg_rhf(object, source = c("oob", "inbag"), ...)
```

Arguments

object	A fitted rhf object from **randomForestRHF**.
...	Not currently used.
source	Which ensemble estimate to extract: "oob" (default, out-of-bag) or "inbag". Falls back to the other when the requested one is absent (e.g. bootstrap = "none" has no OOB estimate).

Value

A data.frame of class c("gg_rhf", "data.frame") with columns id, time, hazard, chf, source, an integer ntime attribute (number of grid points), and a provenance attribute.

The frame always has ntime rows per case. Both hazard and chf may be NA, and they are masked on different rules. From **randomForestRHF** 2.0.0 the pointwise hazard is defined only where a grid point falls inside one of the case's supplied (start, stop] intervals, and is NA in gaps and after the final stop time. From 2.0.3 the cumulative hazard is NA after each case's final stop. **randomForestRHF** documents it as flat through an internal gap, but 2.1.0 returns NA from the start of a case's first gap to the end of its follow-up, including the observed interval after the gap, where the hazard is defined. So do not rely on chf for a case with a gap in its path. On a fit whose cases each have a single interval the two masks look the same; with time-dependent covariates they need not. Earlier versions carried chf across the whole grid. The values are passed through unchanged; use na.rm = TRUE when summarising either column, or see plot.gg_rhf(), which drops the masked cells before drawing.

References

Ishwaran H, Hsich EM, Kogalur UB, Lee DKK (2026). Random Hazard Forests. arXiv:2608.21597. doi:10.48550/arXiv.2608.21597.

Ishwaran H, Kogalur UB (2026). *randomForestRHF: Random Hazard Forests*. R package version 2.0.3. https://CRAN.R-project.org/package=randomForestRHF.

See Also

plot.gg_rhf(), randomForestRHF::rhf()

Examples

```
if (requireNamespace("randomForestRHF", quietly = TRUE)) {
  data(pbc, package = "randomForestSRC")
  d <- randomForestRHF::convert.counting(
    survival::Surv(days, status) ~ ., na.omit(pbc))
  o <- randomForestRHF::rhf("Surv(id, start, stop, event) ~ .", d, ntree = 30)
  gg <- gg_rhf(o)
  plot(gg, idx = c(1, 5, 10))
}
```

gg_rhf_importance *Tidy time-localized variable priority from a Random Hazard Forest*

Description

Extracts the variable-priority matrix from randomForestRHF::importance.rhf() into a tidy data frame, one row per variable and time window. The score measures how much the fitted integrated hazard changes when rules involving a variable are released.

Usage

```
gg_rhf_importance(object, ...)

## S3 method for class 'importance.rhf'
gg_rhf_importance(object, ...)

## S3 method for class 'rhf'
gg_rhf_importance(
  object,
  importance_fit = NULL,
  cache = NULL,
  time.index = NULL,
  ...
)
```

Arguments

object	A fitted rhf object from **randomForestRHF**.
...	Additional arguments passed to randomForestRHF::importance.rhf() when importance_fit is NULL.
importance_fit	Optional precomputed randomForestRHF::importance.rhf() result for the same object. Supply this object when you have already calculated variable priority. NULL (default) calculates it from object.
cache	Optional precomputed randomForestRHF::varpro.cache.rhf() result used when importance_fit is NULL.
time.index	Optional time-grid indices passed to randomForestRHF::importance.rhf() when importance_fit is NULL.

Details

Variable priority is time-localized. A large value means that releasing rules involving that variable changed the log integrated hazard more in that window. It is a ranking score, not a z-score, and this function does not apply a significance cutoff.

Calculating the upstream result can be expensive. For an analysis you will revisit, calculate importance_fit once and supply it here. The extractor accepts cache, time.index, and additional calculation arguments only when importance_fit is NULL.

Variables are ordered for plotting by their 90th percentile (`q90`) priority across windows. This changes the factor levels, but does not change the upstream row order or priority values.

Value

A `data.frame` of class `c("gg_rhf_importance", "data.frame")` with columns:

variable Variable name, ordered by `q90` priority for plotting.

time_window Upstream time-window label.

time Evaluation time at the end of the window.

time_index Index of `time` on the RHF time grid.

start, stop, midpoint Window boundaries and midpoint.

n_risk Number of observations at risk.

n_rules Number of rules contributing to the window.

priority RHF variable-priority score.

A `provenance` attribute records the source forest, upstream settings, whether `importance_fit` was supplied, and the installed **randomForestRHF** version.

References

Ishwaran H, Hsich EM, Kogalur UB, Lee DKK (2026). Random Hazard Forests. arXiv:2608.21597. doi:10.48550/arXiv.2608.21597.

Ishwaran H, Kogalur UB (2026). *randomForestRHF: Random Hazard Forests*. R package version 2.0.3. https://CRAN.R-project.org/package=randomForestRHF.

See Also

`plot.gg_rhf_importance()`, `randomForestRHF::importance.rhf()`

Examples

```
if (requireNamespace("randomForestRHF", quietly = TRUE)) {
  data(pbc, package = "randomForestSRC")
  d <- randomForestRHF::convert.counting(
    survival::Surv(days, status) ~ ., na.omit(pbc))
  o <- randomForestRHF::rhf(
    "Surv(id, start, stop, event) ~ .", d, ntree = 30)

  priority_fit <- randomForestRHF::importance.rhf(o)
  priority <- gg_rhf_importance(o, importance_fit = priority_fit)
  plot(priority)
}
```

gg_roc.rfsrc *ROC (Receiver Operating Characteristic) curve data from a classification forest.*

Description

A classifier does not hand you a class; it hands you a predicted probability, and you pick a threshold. Slide that threshold from 0 to 1 and the trade-off between catching the positives and crying wolf shifts the whole way. The ROC curve traces that trade-off. For one class of a classification rfsrc or randomForest forest, gg_roc walks every threshold and records sensitivity (the true positive rate) against specificity (1 minus the false positive rate).

Usage

```
## S3 method for class 'rfsrc'
gg_roc(object, which_outcome, oob = TRUE, per_class = FALSE, ...)

gg_roc(object, which_outcome, oob = TRUE, per_class = FALSE, ...)
```

Arguments

object	A classification rfsrc or randomForest object. Only forests with family == "class" (rfsrc) or type == "classification" (randomForest) are supported.
which_outcome	Index of the class to score: 1 or 2 for a binary forest, any valid class index for a multi-class one. A character level name is accepted by the randomForest method only; the rfsrc method requires an integer index. which_outcome = "all" or 0 behaves differently by engine: randomForest **method** Returns a macro-averaged one-vs-rest ROC computed over the per-class probabilities. rfsrc **method** Warns and falls back to class 1. The macro-average and per-class faceting for the rfsrc path are tracked separately under issue #72.
oob	Logical; if TRUE (default), build the curve from out-of-bag predicted probabilities, otherwise from full in-bag predictions. For randomForest, TRUE uses the out-of-bag vote probabilities in object$votes; FALSE uses in-bag predict(type = "prob").
per_class	Logical; if TRUE and the forest has more than two classes, return one ROC curve per class, each class scored against all the others. The result is a long-format data.frame with a class factor column and a named AUC vector attribute, ordered by descending AUC. Binary forests treat per_class = TRUE as a no-op. Honored by the randomForest method only.
...	Extra arguments (currently unused).

Value

A gg_roc data.frame, one row per unique prediction threshold, with columns:

sens Sensitivity (true positive rate) at the threshold.

spec Specificity (true negative rate) at the threshold.

pct The probability threshold used for that row.

Pass it to calc_auc for the area under the curve.

Defaults across entry points

plot(gg_roc(x)) and a direct plot.gg_roc(x) call on a raw multi-class forest do not draw the same figure: gg_roc(x) returns a single curve, while plot.gg_roc given a raw forest overlays one curve per class. Plain plot(x) on a forest reaches neither, as it dispatches to the forest's own method in randomForestSRC or randomForest. Pass which_outcome explicitly when the distinction matters; issue #72 tracks reconciling the two.

See Also

plot.gg_roc, calc_roc, calc_auc, rfsrc, randomForest

Examples

```
## ------------------------------------------------------------
## classification example
## ------------------------------------------------------------
## -------- iris data
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris, ntree = 100)

# ROC for setosa
gg_dta <- gg_roc(rfsrc_iris, which_outcome = 1)
plot(gg_dta)

# ROC for versicolor
gg_dta <- gg_roc(rfsrc_iris, which_outcome = 2)
plot(gg_dta)

# ROC for virginica
gg_dta <- gg_roc(rfsrc_iris, which_outcome = 3)
plot(gg_dta)

## -------- iris data
rf_iris <- randomForest::randomForest(Species ~ ., data = iris)

# ROC for setosa
gg_dta <- gg_roc(rf_iris, which_outcome = 1)
plot(gg_dta)

# ROC for versicolor
gg_dta <- gg_roc(rf_iris, which_outcome = 2)
plot(gg_dta)
```

```
# ROC for virginica
gg_dta <- gg_roc(rf_iris, which_outcome = 3)
plot(gg_dta)
```

---

gg_sdependent *Signal-variable detection from an unsupervised varPro fit*

Description

Tidy wrapper around varPro::sdependent() for a uvarpro object. Where gg_udependent() draws the cross-variable dependency *graph*, gg_sdependent() surfaces sdependent()'s *signal-variable detection*: a ranked table of the per-variable signal score and graph degree, with the variables flagged as "signal" (those whose dependency structure clears the detection threshold).

Usage

```
gg_sdependent(
  object,
  ...,
  threshold = 0.25,
  q.signal = 0.75,
  directed = TRUE,
  min.degree = NULL,
  beta_fit = NULL
)
```

Arguments

object	A uvarpro object from varPro::uvarpro().
...	Forwarded to varPro::get.beta.entropy() when beta_fit = NULL; ignored, with a warning, when beta_fit is supplied.
threshold, q.signal, directed, min.degree	Passed to varPro::sdependent() (defaults match gg_udependent()).
beta_fit	Optional precomputed varPro::get.beta.entropy() matrix.

Details

sdependent() runs on the varPro::get.beta.entropy() lasso-coefficient matrix and returns, with plot = FALSE, a list of imp.score (per-variable signal score), degree (node degree in the dependency graph), and signal.vars (the detected signal set). This wrapper tidies that into one row per candidate variable, ranked by imp.score. Because the entropy matrix is the expensive part, beta_fit accepts a precomputed varPro::get.beta.entropy() matrix (shared with gg_beta_uvarpro() and gg_udependent()).

Value

A gg_sdependent object (a data.frame), one row per candidate variable, most-signal first, with columns:

variable factor; levels reversed so the top variable lands at the top after coord_flip().

imp_score sdependent() per-variable signal score.

degree node degree in the dependency graph.

signal logical; variable is in sdependent()$signal.vars.

The provenance attribute records source, family ("unsupv"), threshold, q.signal, directed, n_signal, and n_var.

See Also

gg_udependent() (the dependency graph), gg_beta_uvarpro() (lasso importance), varPro::sdependent(), varPro::uvarpro().

Examples

```
if (requireNamespace("varPro", quietly = TRUE)) {
  set.seed(1)
  o <- varPro::uvarpro(mtcars, ntree = 50)
  gg <- gg_sdependent(o)
  plot(gg)
}
```

---

gg_shap *SHAP (Shapley additive explanations) data object*

Description

gg_shap computes SHAP values for a rfsrc or randomForest regression or classification forest by wrapping kernelshap, and reshapes them into a tidy data set with one row per (observation, variable).

Usage

```
gg_shap(object, newdata, bg_n = 50, which.class = 1, ...)
```

Arguments

object	A rfsrc or randomForest object (regression or classification).
newdata	Optional data.frame of predictor values to explain (same columns as the model's training predictors). When missing, the model's own training predictors are used.
bg_n	Size of the background/reference sample drawn from the training predictors and passed to kernelshap as bg_X. Larger values are more accurate but slower.
which.class	For classification forests, the class (integer column index into the predicted-probability matrix) whose predicted probability is explained. Defaults to 1.
...	Passed through to kernelshap (e.g. seed, exact, max_iter).

Value

A gg_shap object: a data.frame with columns id (observation index), vars (variable name, an ordered factor ranked by mean absolute SHAP), shap (the signed SHAP contribution), value (numeric feature value, NA for categorical features), and value_label (feature value as character). The background-sample mean prediction is stored in the "baseline" attribute.

See Also

kernelshap

Examples

```
if (requireNamespace("kernelshap", quietly = TRUE)) {
  rf <- randomForestSRC::rfsrc(Ozone ~ ., data = na.omit(airquality),
                               ntree = 50)
  gg_dta <- gg_shap(rf, bg_n = 20)
}
```

---

gg_survival *Nonparametric survival estimates.*

Description

Nonparametric survival estimates.

Usage

```
gg_survival(
  object = NULL,
  interval = NULL,
  censor = NULL,
  by = NULL,
```

```
  data = NULL,
  type = c("kaplan", "nelson"),
  ...
)

## S3 method for class 'rfsrc'
gg_survival(
  object,
  interval = NULL,
  censor = NULL,
  by = NULL,
  data = NULL,
  type = c("kaplan", "nelson"),
  ...
)

## Default S3 method:
gg_survival(
  object = NULL,
  interval = NULL,
  censor = NULL,
  by = NULL,
  data = NULL,
  type = c("kaplan", "nelson"),
  ...
)
```

Arguments

object	For the rfsrc method: a fitted rfsrc survival forest. For the default method: pass NULL (or omit) and supply interval, censor, and data instead.
interval	Character; name of the time-to-event column in data (default method only).
censor	Character; name of the event-indicator column in data (1 = event, 0 = censored; default method only).
by	Optional character; name of a grouping column for stratified estimates. For the rfsrc method, by must be a column in object$xvar.
data	A data.frame containing survival data (default method only).
type	One of "kaplan" (Kaplan-Meier, default) or "nelson" (Nelson-Aalen cumulative hazard). Default method only.
...	Additional arguments passed to kaplan or nelson.

Details

Placing the forest's ensemble survival curve over the marginal Kaplan-Meier baseline is a calibration check, not a test of signal. The ensemble curve averages the per-subject curves, so it should track the Kaplan-Meier estimate whether or not the predictors carry information; a forest grown on

pure noise tracks it about as closely as one grown on real predictors. A clear gap points to miscalibration, most often at late times when few subjects remain at risk. To see what the predictors carry, stratify with by and compare the groups. gg_survival computes the nonparametric baseline (the Kaplan-Meier or Nelson-Aalen estimate) so you can place it on the same canvas as the forest predictions from gg_rfsrc.

gg_survival is an S3 generic that dispatches on the class of its first argument:

rfsrc Extracts the outcome columns from the fitted forest's $yvar slot (time in column 1, event indicator in column 2) and delegates to kaplan. Use by to stratify on a predictor from $xvar: you get one Kaplan-Meier curve per group, ready to compare against the forest's group-specific ensemble curves.

default Accepts raw survival columns directly via interval, censor, and data. Delegates to kaplan (the default) or nelson depending on type.

Value

A gg_survival data.frame with columns time, surv, cum_haz, lower, upper, n.risk, and optionally groups when by is supplied.

Note

Survival estimation is randomForestSRC-only; randomForest has no survival forest, so no randomForest method exists.

See Also

kaplan nelson

plot.gg_survival

Examples

```
## -------- pbc data (default method, raw data columns)
data(pbc, package = "randomForestSRC")
pbc$time <- pbc$days / 364.25

gg_dta <- gg_survival(interval = "time", censor = "status", data = pbc)
plot(gg_dta, error = "none")

# Stratified
gg_dta <- gg_survival(
  interval = "time", censor = "status",
  data = pbc, by = "treatment"
)
plot(gg_dta)
```

gg_tune_rhf *Tidy a Random Hazard Forest tuning path*

Description

Extracts an already calculated tree-size tuning path from `randomForestRHF::tune.treesize.rhf()` into a data frame for inspection and plotting. The expensive step is upstream tuning. Calculate and retain that result once, then supply it to `gg_tune_rhf()` when you need the saved search path or its plot. `gg_tune_rhf()` only prepares that path; it never tunes a forest.

Usage

```
gg_tune_rhf(tune_fit, ...)

## S3 method for class 'tune.treesize.rhf'
gg_tune_rhf(tune_fit, ...)
```

Arguments

`tune_fit`	An object inheriting from `tune.treesize.rhf`, typically returned by `randomForestRHF::tune.treesiz` `randomForestRHF::tune.rhf()`, or `randomForestRHF::tune.iAUC.rhf()`.
`...`	Additional arguments reserved for methods.

Details

The returned path preserves the row order in `tune_fit$path`. Its columns are `treesize` (evaluated forest size), `metric` (`"OOB risk"` or `"OOB iAUC"`), `value` (the observed metric), `se` (the supplied bootstrap iAUC standard error, or `NA_real_`), and `selected` (whether that size is the upstream `best.size`). Upstream tuning minimizes OOB risk or maximizes OOB iAUC.

Provenance is stored in the `provenance` attribute: `best_size` is the selected tree size; `best_err` is OOB risk for risk tuning and `1 - iAUC` for iAUC tuning; `perf` identifies the criterion; `method` is the upstream search method; `bounds` gives its tree-size range; `n_evaluations` counts the evaluated sizes; and `randomForestRHF_version` records the installed upstream package version. The optional fitted forest is not copied into the tidy result.

Value

A `data.frame` with class `c("gg_tune_rhf", "data.frame")` and columns `treesize`, `metric`, `value`, `se`, and `selected`. The `provenance` attribute contains the upstream settings described in `gg_tune_rhf`.

References

Ishwaran H, Hsich EM, Kogalur UB, Lee DKK (2026). Random Hazard Forests. arXiv:2608.21597. doi:10.48550/arXiv.2608.21597.

Ishwaran H, Kogalur UB (2026). *randomForestRHF: Random Hazard Forests*. R package version 2.0.3. https://CRAN.R-project.org/package=randomForestRHF.

See Also

plot(), randomForestRHF::tune.treesize.rhf()

Examples

```
if (requireNamespace("randomForestRHF", quietly = TRUE)) {
  ## Calculate this expensive result once and retain it for reuse.
  simulated <- randomForestRHF::hazard.simulation(1, n = 100, nrecords = 3)
  tune_fit <- randomForestRHF::tune.iAUC.rhf(
    "Surv(id, start, stop, event) ~ .",
    simulated$dta,
    ntree = 12L,
    lower = 2L,
    upper = 5L,
    verbose = FALSE,
    forest = FALSE
  )
  tuning <- gg_tune_rhf(tune_fit)
  plot(tuning)
}
```

---

gg_udependent *Variable dependency graph from a uvarpro model*

Description

A uvarpro fit records how strongly each variable depends on the others. This function pulls those cross-variable dependency scores from the fit with get.beta.entropy and sdependent, and returns them as a tidy list that plot.gg_udependent can draw as a network.

Usage

```
gg_udependent(
  object,
  threshold = 0.25,
  q.signal = 0.75,
  directed = TRUE,
  min.degree = NULL,
  ...,
  pre.filter = TRUE,
  beta.mat = NULL
)
```

Arguments

object	A fitted uvarpro object (required).
threshold	Numeric; the positive dependency threshold passed on to sdependent(). An edge $i \to j$ is drawn when I[i, j] >= threshold. Default 0.25.
q.signal	Quantile threshold (0–1) for picking out the signal variables; passed on to sdependent(). Default 0.75.
directed	Logical; TRUE (default) builds a directed igraph.
min.degree	Integer or NULL. When set, only nodes with degree $\geq$ min.degree are kept in $nodes, $edges, and $graph.
...	Additional arguments forwarded to varPro::sdependent().
pre.filter	Logical; forwarded to varPro::get.beta.entropy(). TRUE (default) restricts the lasso fits to the variables that pass varPro's importance pre-filter. FALSE uses every variable, so one the pre-filter drops (an outcome column, say) can still reach the graph. Ignored when beta.mat is supplied.
beta.mat	Optional precomputed dependency matrix, as returned by varPro::get.beta.entropy(object, ...), with row and column names. Supply it to reuse one expensive computation across several calls. NULL (default) computes it from object.

Value

A named list of class "gg_udependent" with elements:

$edges Data frame: variable_from, variable_to, weight (raw cross-importance value).

$nodes Data frame: variable (factor, levels by descending degree), degree (integer; out-degree when directed = TRUE, total degree when directed = FALSE), selected (logical, TRUE if in sdependent's signal set).

$graph igraph object. NULL if no dependencies detected.

A "provenance" attribute carries threshold, q.signal, directed, min.degree, pre.filter (NA when beta.mat was supplied, since it was not used), beta.mat.supplied, xvar.names, and n.

What cross-variable dependency is doing

UVarPro (Zhou, Lu and Ishwaran, 2026) extends the varpro framework to the unsupervised setting: grow a forest without a response, then use the same region-release contrasts varpro uses for supervised importance to ask, "which variables explain the structure in the data?" The lasso-driven variant frames each region-release contrast as a classification task (does an observation belong to the region or to its release?) and fits a lasso logistic regression with the other variables as predictors. The coefficient on variable j in the model for variable i's region-release contrast is the entry $I[i, j]$ of the matrix varPro::get.beta.entropy() returns.

Read that entry as "how much does knowing j help separate i's region from its release". A large $I[i, j]$ says j carries information about the structure varpro picked up in i. varPro::sdependent thresholds that matrix at a user-chosen cut and returns the set of "signal" variables: the nodes with high enough out-degree to be worth keeping. We pass the threshold through to sdependent and use the same matrix to weight the edges of the resulting graph.

The graph is directed by default because $I[i, j]$ and $I[j, i]$ are separate lasso coefficients and need not agree; setting `directed = FALSE` collapses each pair by taking the larger of the two, which is appropriate when you only want to see that two variables are dependent, not which way the dependency reads.

Mixed-type data

The matrix `get.beta.entropy()` returns is usually not square. Its rows are the variables that produced region releases; its columns are every one-hot-encoded predictor column, so factor levels that never released, and variables the pre-filter dropped, appear only as columns. We pad it to a square matrix over the union of the names, filling with 0 (no release, no dependency). We also zero the entries between levels of the same factor (`cyl4` and `cyl8`, say): the levels are mutually exclusive by construction, so that edge says nothing about the data.

What's in the output

`$edges` has one row per surviving edge with the raw weight `I[i, j]` (or, for undirected graphs, the max of the two directions). `$nodes` has one row per surviving variable with its degree (out-degree for directed, total degree for undirected) and a `selected` flag for membership in the `sdependent` signal set. `$graph` is the same information packaged as an `igraph` object, with `weight`, `degree`, and `selected` attached so `plot.gg_udependent` can render it without recomputing anything.

What you use this for

- screen a wide unsupervised dataset for the small set of variables UVarPro thinks are carrying the signal: the nodes with high degree, or those flagged `selected = TRUE`;
- spot clusters of mutually dependent variables (hubs and the spokes around them) that may be measuring the same underlying construct;
- compare two datasets, or two preprocessing pipelines, by looking at how their dependency graphs change.

An edge in this graph is a statistical dependency in the unsupervised decomposition of the data. It is not a causal arrow. A high $I[i, j]$ says j predicts i's region membership, not that j causes i.

References

Zhou, L., Lu, M. and Ishwaran, H. (2026). Variable priority for unsupervised variable selection. *Pattern Recognition*, 172:112727.

See Also

`plot.gg_udependent`

Examples

```
set.seed(42)
uv <- varPro::uvarpro(iris[, -5], ntree = 50)
gg <- gg_udependent(uv)
print(gg)
```

gg_variable *Marginal variable dependence data object.*

Description

plot.variable generates a data.frame containing the marginal variable dependence or the partial variable dependence. The gg_variable function creates a data.frame of containing the full set of covariate data (predictor variables) and the predicted response for each observation. Marginal dependence figures are created using the plot.gg_variable function. A randomForest fit does not keep the model frame, so for those objects gg_variable rebuilds it from the stored call. That lets the same predictors be paired with the in-sample predictions.

A few optional arguments tune the extraction: time (one survival time, or a vector of them), time_labels (labels for multiple survival horizons), and oob, which switches between out-of-bag and in-bag predictions when the forest carries both.

Usage

```
gg_variable(object, ...)
```

Arguments

object	A rfsrc or randomForest object, or a plot.variable result.
...	Optional arguments time, time_labels, and oob that tune the marginal dependence extraction.

Details

The marginal variable dependence is determined by comparing relation between the predicted response from the randomForest and a covariate of interest.

The gg_variable function operates on a rfsrc object, the output from the plot.variable function, or on a fitted randomForest object via the formula interface.

Value

A gg_variable object: a data.frame pairing every training predictor column with the OOB (or in-bag) predicted response. For survival forests, each requested time horizon adds a column named by time_labels. The object carries a "family" class attribute ("regr", "class", or "surv") that plot.gg_variable uses for dispatch.

See Also

plot.gg_variable, plot.variable

Examples

```
## ------------------------------------------------------------
## classification (small, runs on CRAN)
## ------------------------------------------------------------
## -------- iris data
set.seed(42)
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris, ntree = 50)

gg_dta <- gg_variable(rfsrc_iris)
plot(gg_dta, xvar = "Sepal.Width")


## ------------------------------------------------------------
## Additional classification / regression / survival examples are
## guarded with \donttest because the cumulative example time exceeds
## the 10-second CRAN budget. Run locally with `R CMD check
## --run-donttest` (or `devtools::check(run_dont_test = TRUE)`) to
## exercise them.
## ------------------------------------------------------------
plot(gg_dta, xvar = "Sepal.Length")
plot(gg_dta, xvar = rfsrc_iris$xvar.names, panel = TRUE)

## ------------------------------------------------------------
## regression
## ------------------------------------------------------------

## -------- air quality data
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ ., data = airquality, ntree = 50)
gg_dta <- gg_variable(rfsrc_airq)

# an ordinal variable
gg_dta[, "Month"] <- factor(gg_dta[, "Month"])

plot(gg_dta, xvar = "Wind")
plot(gg_dta, xvar = "Temp")
plot(gg_dta, xvar = "Solar.R")
plot(gg_dta, xvar = c("Solar.R", "Wind", "Temp", "Day"), panel = TRUE)
plot(gg_dta, xvar = "Month", notch = TRUE)

## -------- motor trend cars data
rfsrc_mtcars <- randomForestSRC::rfsrc(mpg ~ ., data = mtcars, ntree = 50)

gg_dta <- gg_variable(rfsrc_mtcars)

# mtcars$cyl is an ordinal variable
gg_dta$cyl <- factor(gg_dta$cyl)
gg_dta$am <- factor(gg_dta$am)
gg_dta$vs <- factor(gg_dta$vs)
gg_dta$gear <- factor(gg_dta$gear)
gg_dta$carb <- factor(gg_dta$carb)

plot(gg_dta, xvar = "cyl")
```

```
plot(gg_dta, xvar = "disp")
plot(gg_dta, xvar = "hp")
plot(gg_dta, xvar = "wt")
plot(gg_dta, xvar = c("disp", "hp", "drat", "wt", "qsec"), panel = TRUE)
plot(gg_dta,
  xvar = c("cyl", "vs", "am", "gear", "carb"), panel = TRUE,
  notch = TRUE
)

## -------- Boston data
if (requireNamespace("MASS", quietly = TRUE)) {
  data(Boston, package = "MASS")
  rf_boston <- randomForest::randomForest(medv ~ ., data = Boston)
  gg_dta <- gg_variable(rf_boston)
  plot(gg_dta)
  plot(gg_dta, panel = TRUE)
}

## ------------------------------------------------------------
## survival examples
## ------------------------------------------------------------

## -------- veteran data
data(veteran, package = "randomForestSRC")
rfsrc_veteran <- randomForestSRC::rfsrc(Surv(time, status) ~ ., veteran,
  nsplit = 10,
  ntree = 50
)

# get the 90-day survival time.
gg_dta <- gg_variable(rfsrc_veteran, time = 90)

# Generate variable dependence plots for age and diagtime
plot(gg_dta, xvar = "age")
plot(gg_dta, xvar = "diagtime")

# Generate coplots
plot(gg_dta, xvar = c("age", "diagtime"), panel = TRUE, se = FALSE)

# Compare survival at 30, 90, and 365 days simultaneously
gg_dta <- gg_variable(rfsrc_veteran, time = c(30, 90, 365))
plot(gg_dta, xvar = "age")
```

---

gg_varpro *Variable importance data from a varPro model*

Description

Pulls the per-tree importance scores out of a fitted varpro object and summarizes them into a data structure the plot method can draw as a boxplot. The box hinges are the 15th and 85th percentiles and the whiskers run to the 5th and 95th, not the usual Tukey 1.5 IQR whiskers. For a classification forest you can also keep the class-conditional importances.

Usage

```
gg_varpro(
  object,
  local.std = TRUE,
  cutoff = 0.79,
  faithful = FALSE,
  conditional = FALSE,
  nvar = NULL,
  trim = 0.1,
  max.tree = object$max.tree,
  max.rules.tree = object$max.rules.tree,
  noise = NULL,
  kind_values = 5L,
  ...
)
```

Arguments

object	A fitted varpro object (required).
local.std	Logical; default TRUE. Passed to varPro's importance aggregation, with the same meaning as in varPro::importance(): TRUE re-samples the release rules, scores them by local importance and reports z as their winsorised mean, so the boxes are on z's scale. FALSE uses the fit-time rules and standardises z by the winsorised standard deviation; the boxes are then on the raw importance scale, which is what type = "raw" in plot.gg_varpro needs. With TRUE the result depends on the random number stream, so call set.seed() first for a reproducible z.
cutoff	Numeric; the z-score above which a variable is treated as selected. Default 0.79. A variable with aggregate z above the cutoff is flagged selected = TRUE in $imp. varPro uses 0.79 only to highlight its plot; it is not a selection rule that carries from one fit to another. Prefer noise, or tune the cutoff with varPro::cv.varpro(). Ignored for selection when noise is set; the conditional plot still draws it.
faithful	Logical; default FALSE. When TRUE, $imp.tree is kept so plot.gg_varpro can scatter the per-tree points over the box.
conditional	Logical; default FALSE. When TRUE, and only for a classification forest, the $conditional.z matrix is extracted and stored as $conditional.
nvar	Integer; keep only the top nvar variables, ranked by median per-tree z, after the cutoff filter has been applied. NULL keeps all of them.

`trim` Winsorising fraction used for z, as in `varPro::importance()`; default `0.1`. A variable is `tested` when its coverage is at least `trim`.

`max.tree`, `max.rules.tree`
Number of trees, and release rules per tree, to score. Default to the values stored in the fit. Supplying either one re-samples the rules even with `local.std = FALSE`, as `varPro::importance()` does. The boxes, z and coverage are always computed from the same rules.

`noise` `NULL` (the default), or a regular expression matching the names of permuted noise columns in the fit's data (for example `"^noise"`). Selection then replaces `cutoff`: a variable is selected when it is tested (and scored in at least one tree) and its z beats the largest z among *tested* noise of the same kind (categorical or continuous), as in `varpro_coverage`. A kind with no tested noise borrows the other kind's bar, with a warning, and `noise_ref` records which. The plot draws that bar instead of the `cutoff` line. The bar is computed from the same rules as the boxes and over every variable, before `nvar` truncation. Needs `local.std = TRUE`. ggRandomForests does not add noise columns; add them before fitting, and fit with `split.weight = FALSE` so that varPro's first-pass filter does not cut them.

`kind_values` With `noise`, a numeric variable with at most this many distinct values counts as categorical.

`...` Not used. Earlier versions passed it to `varPro::importance()`; `trim`, `max.tree` and `max.rules.tree` are now arguments, and anything else is an error.

Value

A named list of class `"gg_varpro"` with elements:

`$imp` Summary data frame: `variable` (factor whose levels run least- to most-important by median per-tree z, so the most-important variable sits at the top of the plot after `coord_flip`), `z` (aggregate z-score, `NA` when it is undefined), `selected` (logical, `z > cutoff`), `coverage` (share of the scored trees in which a release rule for the variable returned an importance value) and `tested` (`coverage >= trim`). With `noise` set it adds `noise` (logical), `kind`, `noise_cut` and `noise_ref` (the kind of the noise that set `noise_cut`, `NA` when no noise was tested), and `selected` is a tested non-noise variable, scored in at least one tree, whose z beats `noise_cut`.

`$imp.tree` `NULL` when `faithful = FALSE`; otherwise an ntree x p matrix of per-tree importance values.

`$stats` Per-variable summary: `variable`, `median`, `q05`, `q15`, `q85`, `q95` of the per-tree values (on z's scale when `local.std = TRUE`, raw when `FALSE`), `mean` (their plain mean), and `coverage` and `tested` as in `$imp`.

`$conditional` `NULL` when `conditional = FALSE`; otherwise a data frame with columns `variable`, `class`, `z` (one row per variable x class combination). `class` is a factor of the outcome's own level names, also for a binary outcome, which varPro recodes to 0/1 internally.

A `"provenance"` attribute carries `family`, `local.std`, `cutoff`, `faithful`, `conditional`, `xvar.names`, `n`, `trim`, `max.tree` and `max.rules.tree`, and `noise` when it was set.

What varpro is doing

Permutation importance asks "what happens to OOB accuracy when I scramble this variable?" That works, but it leans on artificial data (the permuted column) and the answer can be unstable when variables are correlated. The varpro framework (Lu and Ishwaran, 2024) replaces permutation with *release rules*. The forest is grown with guided splitting; from a subset of trees varpro samples a collection of decision-rule branches; for each variable it then compares the response inside the rule's region to the response after the rule's constraint on that variable is "released". The size of that change, aggregated over many rules and trees, is the variable's importance. No synthetic covariates, no permutation: the contrast is between two real subsets of the data.

Because varpro builds importance from rules sampled over trees, every tree contributes its own importance value for each variable. Those are the per-tree scores we summarize here, and the aggregate z comes from the same rules on the same trees, so the boxes and z always agree. `varPro::importance()` takes a winsorised mean over trees (trimmed by `trim`), counting a tree with no usable release rule for the variable as 0. With `local.std = TRUE` (the default, as in varPro) the release rules are re-sampled and scored by their local importance, the standard deviation is fixed at 1, and z is that winsorised mean, so the per-tree values already sit on z's scale. With `local.std = FALSE` the fit-time rules are used and z is the winsorised mean divided by the winsorised standard deviation. `cutoff = 0.79` is varpro's default selection threshold, but the scale of z depends on how many variables share each tree, so the same cutoff does not mean the same thing across settings.

A variable scored in fewer than about `trim` of the trees (a tree scores it when one of its release rules for the variable returns an importance value) is *untested*: its z is 0 (or undefined with `local.std = FALSE`) because every non-zero tree is winsorised away, not because it was tested and found unimportant. `coverage` and `tested` in the output say which variables those are, and the plot draws them as open boxes. `varpro_coverage` counts coverage the same way and adds a per-variable roll-up and a noise-based cutoff.

For a classification forest, varpro also returns a class-conditional z table: the same importance computed restricting attention to rules relevant to each class. `conditional = TRUE` keeps that table so the plot method can show which variables matter for which class rather than only in aggregate.

What's in the output

`$imp` is the one-row-per-variable summary: aggregate z from varPro's importance aggregation, a `selected` flag for z > `cutoff`, and the variable's `coverage` and `tested` flag. `$stats` holds the box quantiles (5/15/50/85/95 percentiles, plus the raw mean) computed from the per-tree matrix; these are what the boxplot draws. `$imp.tree` is the per-tree matrix itself, kept only when `faithful = TRUE` so the plot method can scatter individual tree values over the box. `$conditional` is the tidy class x variable z table, present only when `conditional = TRUE` and the family is classification.

What you use this for

- rank candidate variables by importance and pick a working set above varpro's z cutoff;
- see, via the boxplot's spread and the per-tree points (`faithful = TRUE`), how stable each variable's importance is across trees: a high median with a wide box is a different story from a high median with a tight box;
- for a classification forest, ask which variables drive which class (`conditional = TRUE`) rather than just which variables drive the model overall.

The z-score is a standardized ranking statistic, not a p-value or a probability. Two variables with the same z are "similarly important by this method", not "equally likely to be true signal". For a data-driven cutoff rather than the 0.79 default, see `varPro::cv.varpro`.

Reading the plot

Rows are ordered by the median tree, but the highlight follows z, varPro's winsorised mean across the trees, against `cutoff`. The two can disagree: a variable a few trees find very important and many find unimportant has a long box and a low median.

A tree with no usable release rule for a variable contributes a zero, which pulls the lower whisker and box toward 0 as coverage falls; at high coverage the box need not reach 0. varPro winsorises the per-tree values (`trim = 0.1`: values beyond the 10th and 90th percentiles are pulled in to them), so a variable scored in fewer than about `trim` of the trees comes back with z exactly 0. That variable is untested, not unimportant; it is drawn as an open box, and `$imp$coverage` and `$imp$tested` say so.

By default varPro scores a random sample of 150 trees, so on a large forest the plot can change from run to run. Pass `max.tree = object$rf$ntree` to score every tree. On small or sparse data the 0.79 cutoff decides a great deal; see `varpro_coverage` and `vignette("varpro-small-data", package = "ggRandomForests")`.

References

Lu, M. and Ishwaran, H. (2024). Model-independent variable selection via the rule-based variable priority framework. *arXiv preprint* arXiv:2409.09003.

See Also

`plot.gg_varpro`, `varpro_coverage`, `gg_vimp`

Examples

```
set.seed(42)
vp <- varPro::varpro(mpg ~ ., data = mtcars, ntree = 50)
gg <- gg_varpro(vp)
print(gg)
plot(gg)
```

---

gg_vimp *Variable Importance (VIMP) data object*

Description

`gg_vimp` extracts the variable importance (VIMP) information from a `rfsrc` or `randomForest` object and reshapes it into a tidy data set.

Usage

```
gg_vimp(object, nvar, ...)
```

Arguments

object	A rfsrc object, the output from vimp, or a fitted randomForest.
nvar	Number of variables to include in the output.
...	arguments passed to the vimp.rfsrc function if the rfsrc object does not contain importance information.

Details

gg_vimp() reports whatever importance the forest stored; it computes nothing itself. Usually that is **permutation (Breiman-Cutler) variable importance**: the forest permutes a variable's observed values across the out-of-bag (OOB) cases, runs those perturbed cases down the already-grown trees, and measures how much the OOB prediction error climbs. That perturbation is synthetic (the variable's link to the response is broken on purpose) so a large increase means the variable was carrying genuine signal; near-zero or negative values mean it added noise or nothing at all.

A randomForest fit needs importance = TRUE to give you this. randomForest::randomForest() defaults to importance = FALSE, and that fit stores only a node-impurity measure (IncNodePurity, RSS, for regression; MeanDecreaseGini for classification), which is not a permutation quantity and is not comparable to one. It is the only importance the forest kept, so it is what gg_vimp() reports, in the vimp column, same as any other. Nothing marks the difference in the plot. So gg_vimp(randomForest(y ~ ., data)) ranks by node purity; pass importance = TRUE and you get permutation VIMP (%IncMSE for regression), and colnames(object$importance) tells you which one you have. randomForestSRC::rfsrc() has no such trap: its importance argument yields permutation VIMP.

When a randomForest fit carries both measures, gg_vimp() reports the permutation one and leaves node purity out of the ranking; the two run on different scales and mean different things, so putting them in one ordering would be meaningless. Read randomForest::importance(object) if you want both. A classification fit names that pair MeanDecreaseAccuracy and MeanDecreaseGini, and stores a permutation column per class besides. Those per-class columns are all permutation measures on one scale, so gg_vimp() keeps them together, names each in the set column, and drops only the Gini one.

gg_varpro() takes the opposite route, comparing local estimators on real observed data through varPro's release rules, with no permutation and no synthetic features. The two approaches answer "which variables matter?" by opposite mechanisms, so a variable can rank differently under each, and that disagreement is itself informative: it often signals interaction structure or non-monotone effects that one mechanism surfaces and the other obscures.

For survival forests, VIMP is measured against the ensemble cumulative hazard function (CHF); the error metric is one minus the concordance index (C-statistic). Variables with non-positive VIMP are flagged in the positive column and colored differently by plot.gg_vimp.

Value

gg_vimp object. A data.frame of VIMP measures in rank order, with columns vars, set (the importance measure), vimp and positive. A classification forest with class-wise permutation im-

portance has one set per class plus the overall measure; a randomForest classification fit grown with importance = FALSE stores only node impurity, so it has a single set. When randomForest objects lack stored importance values a warning is issued and NA placeholders (columns vimp, vars and positive) are returned so plots remain reproducible.

References

Ishwaran H. (2007). Variable importance in binary regression trees and forests, *Electronic J. Statist.*, 1:519-537.

See Also

plot.gg_vimp rfsrc

vimp gg_varpro

Examples

```
## ------------------------------------------------------------
## classification example
## ------------------------------------------------------------
## -------- iris data
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ .,
  data = iris,
  ntree = 50,
  importance = TRUE
)
gg_dta <- gg_vimp(rfsrc_iris)
plot(gg_dta)

## ------------------------------------------------------------
## regression example
## ------------------------------------------------------------

## -------- air quality data
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ ., airquality,
  ntree = 50,
  importance = TRUE
)
gg_dta <- gg_vimp(rfsrc_airq)
plot(gg_dta)


## -------- Boston data
if (requireNamespace("MASS", quietly = TRUE)) {
  data(Boston, package = "MASS")
  rfsrc_boston <- randomForestSRC::rfsrc(medv ~ ., Boston,
    ntree = 50,
    importance = TRUE
  )
  gg_dta <- gg_vimp(rfsrc_boston)
  plot(gg_dta)
```

```
  ## importance = TRUE for permutation VIMP; without it randomForest stores
  ## only IncNodePurity, which is what you would be ranking (see Details).
  rf_boston <- randomForest::randomForest(medv ~ ., Boston,
    importance = TRUE
  )
  gg_dta <- gg_vimp(rf_boston)
  plot(gg_dta)
}


## -------- mtcars data
rfsrc_mtcars <- randomForestSRC::rfsrc(mpg ~ .,
  data = mtcars,
  ntree = 50,
  importance = TRUE
)
gg_dta <- gg_vimp(rfsrc_mtcars)
plot(gg_dta)

## ------------------------------------------------------------
## survival example
## ------------------------------------------------------------

## -------- veteran data
data(veteran, package = "randomForestSRC")
rfsrc_veteran <- randomForestSRC::rfsrc(Surv(time, status) ~ .,
  data = veteran,
  ntree = 50,
  importance = TRUE
)

gg_dta <- gg_vimp(rfsrc_veteran)
plot(gg_dta)

## -------- pbc data
data(pbc, package = "randomForestSRC", envir = environment())
# Recode: binary columns to logical, low-cardinality columns to factors.
for (ind in seq_len(dim(pbc)[2])) {
  if (!is.factor(pbc[, ind])) {
    if (length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 2) {
      if (sum(range(pbc[, ind], na.rm = TRUE) == c(0, 1)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
    }
  } else {
    if (length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 2) {
      if (sum(sort(unique(pbc[, ind])) == c(0, 1)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
      if (sum(sort(unique(pbc[, ind])) == c(FALSE, TRUE)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
    }
```

```
  }
  if (!is.logical(pbc[, ind]) &&
    length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 5) {
    pbc[, ind] <- factor(pbc[, ind])
  }
}
# Convert age from days to years
pbc$age <- pbc$age / 364.24
pbc$years <- pbc$days / 364.24
pbc <- pbc[, -which(colnames(pbc) == "days")]
pbc$treatment <- as.numeric(pbc$treatment)
pbc$treatment[which(pbc$treatment == 1)] <- "DPCA"
pbc$treatment[which(pbc$treatment == 2)] <- "placebo"
pbc$treatment <- factor(pbc$treatment)
# Split into a training set (assigned treatment) and a test set (not assigned)
dta_train <- pbc[-which(is.na(pbc$treatment)), ]
pbc_test <- pbc[which(is.na(pbc$treatment)), ]

# ========
# build the forest:
rfsrc_pbc <- randomForestSRC::rfsrc(
  Surv(years, status) ~ .,
  dta_train,
  ntree = 50,
  nsplit = 10,
  na.action = "na.impute",
  forest = TRUE,
  importance = TRUE,
  save.memory = TRUE
)

gg_dta <- gg_vimp(rfsrc_pbc)
plot(gg_dta)

# Restrict to only the top 10.
gg_dta <- gg_vimp(rfsrc_pbc, nvar = 10)
plot(gg_dta)
```

---

kaplan *nonparametric Kaplan-Meier estimates*

Description

nonparametric Kaplan-Meier estimates

Usage

```
kaplan(interval, censor, data, by = NULL, ...)
```

Arguments

interval	name of the interval variable in the training dataset.
censor	name of the censoring variable in the training dataset.
data	name of the training set data.frame
by	stratifying variable in the training dataset, defaults to NULL
...	arguments passed to the survfit function

Value

gg_survival object

See Also

gg_survival nelson plot.gg_survival

Examples

```
# These get run through the gg_survival examples.
data(pbc, package = "randomForestSRC")
pbc$time <- pbc$days / 364.25

# This is the same as gg_survival
gg_dta <- kaplan(
  interval = "time", censor = "status",
  data = pbc
)

plot(gg_dta, error = "none")
plot(gg_dta)

# Stratified on treatment variable.
gg_dta <- gg_survival(
  interval = "time", censor = "status",
  data = pbc, by = "treatment"
)

plot(gg_dta, error = "none")
plot(gg_dta)
```

---

nelson *nonparametric Nelson-Aalen estimates*

Description

cum_haz is the Nelson-Aalen estimate of the cumulative hazard: at each event time the number of events is divided by the number at risk, and the ratios are summed. The surv column, its standard error and its confidence limits are the Kaplan-Meier estimates, as kaplan returns them, and hazard, density, life and proplife are derived from that surv. The two functions therefore differ only in cum_haz, where kaplan reports $-\log S(t)$. The two agree closely while the risk set is large and diverge in the tail; when the last observation is an event, $-\log S(t)$ is infinite there and the Nelson-Aalen sum is not.

Usage

```
nelson(interval, censor, data, by = NULL, weight = NULL, ...)
```

Arguments

interval	name of the interval variable in the training dataset.
censor	name of the censoring variable in the training dataset.
data	name of the survival training data.frame
by	stratifying variable in the training dataset, defaults to NULL
weight	optional numeric vector of event weights, one per row of data (default NULL, every event counts once). The weights apply to events only: each increment of cum_haz is the summed weight of the events at that time over the unweighted number at risk, so a censored observation's weight has no effect. Use it for severity-weighted events. The Kaplan-Meier columns are not weighted.
...	arguments passed to the survfit function

Value

gg_survival object

See Also

gg_survival nelson plot.gg_survival

Examples

```
# These get run through the gg_survival examples.
data(pbc, package = "randomForestSRC")
pbc$time <- pbc$days / 364.25

# This is the same as gg_survival
gg_dta <- nelson(
  interval = "time", censor = "status",
  data = pbc
)

plot(gg_dta, error = "none")
plot(gg_dta)
```

```
# Stratified on treatment variable.
gg_dta <- gg_survival(
  interval = "time", censor = "status",
  data = pbc, by = "treatment"
)

plot(gg_dta, error = "none")
plot(gg_dta, error = "lines")
plot(gg_dta)

gg_dta <- gg_survival(
  interval = "time", censor = "status",
  data = pbc, by = "treatment",
  type = "nelson"
)

plot(gg_dta, error = "bars")
plot(gg_dta)
```

`plot.gg_ale_interaction` *Plot a* `gg_ale_rfsrc` *interaction object*

Description

Renders the second-order (interaction) ALE surface from `gg_ale_rfsrc` (when called with `xvar2.name`) as a ggplot2 heatmap. Values near zero mean the two predictors act additively over that region of their joint range; large positive or negative values mark where their combined effect departs from the sum of their individual main effects.

Usage

```
## S3 method for class 'gg_ale_interaction'
plot(x, ...)
```

Arguments

`x`	A `gg_ale_interaction` object.
`...`	Not currently used.

Value

A `ggplot` object.

See Also

`gg_ale_rfsrc`, `plot.gg_ale_rfsrc`

Examples

```
airq.obj <- randomForestSRC::rfsrc(Ozone ~ ., data = na.omit(airquality),
                                   ntree = 50)
ale_int <- gg_ale_rfsrc(airq.obj, xvar.names = "Wind",
                        xvar2.name = "Temp", n_eval = 12)
plot(ale_int)
```

`plot.gg_ale_rfsrc` *Plot a* `gg_ale_rfsrc` *object*

Description

Renders first-order Accumulated Local Effects as a ggplot2 figure. Continuous predictors are drawn as line plots and categorical predictors as bar charts, both faceted by variable name, the same arrangement as `plot.gg_partial_rfsrc`, so the two are directly comparable side by side.

Usage

```
## S3 method for class 'gg_ale_rfsrc'
plot(x, labels = NULL, ...)
```

Arguments

`x`	A `gg_ale_rfsrc` object.
`labels`	Optional variable labels for the facet strips. One of: a named character vector (`c(bpd_last = "BP Diastole")`); a labelled data frame, whose `attr(col, "label")` values are read; or a two-column `key`/`label` data frame. Variables with no label keep their raw name. Defaults to `NULL` (raw names).
`...`	Not currently used.

Value

A `ggplot` (or `patchwork`) object. When both continuous and categorical variables are present the two panels are combined vertically via `patchwork::wrap_plots()`.

See Also

`gg_ale_rfsrc`, `plot.gg_partial_rfsrc`

Examples

```
airq.obj <- randomForestSRC::rfsrc(Ozone ~ ., data = na.omit(airquality),
                                   ntree = 50)
ale_dta <- gg_ale_rfsrc(airq.obj, xvar.names = c("Wind", "Temp"),
                        n_eval = 10)
plot(ale_dta)
```

`plot.gg_auct` *Plot a time-varying AUC curve*

Description

Draws AUC(t) from a `gg_auct()` object: a line over time, a bootstrap confidence ribbon when available, and a dashed reference line at 0.5 (chance). The integrated AUC (iAUC) appears in the caption. Times with a non-finite AUC are dropped before drawing: from **randomForestRHF** 2.0.0 the final grid time can be `NA`, where the censoring-weight denominator is undefined once the control set is nearly exhausted.

Usage

```
## S3 method for class 'gg_auct'
plot(x, ...)
```

Arguments

`x`	A `gg_auct` object from `gg_auct()`.
`...`	Not currently used.

Value

A `ggplot` object.

See Also

`gg_auct()`.

Examples

```
if (requireNamespace("randomForestRHF", quietly = TRUE)) {
  data(pbc, package = "randomForestSRC")
  d <- randomForestRHF::convert.counting(
    survival::Surv(days, status) ~ ., na.omit(pbc))
  o <- randomForestRHF::rhf("Surv(id, start, stop, event) ~ .", d, ntree = 30)
  plot(gg_auct(o))
}
```

`plot.gg_beta_uvarpro` *Plot a* `gg_beta_uvarpro` *object*

Description

Horizontal bar chart of the mean absolute lasso coefficient $\mathrm{mean}(|\hat{\beta}|)$ per variable from an unsupervised varPro fit, sorted descending so the eye lands on the top variable first. Bars are filled blue above the selection cutoff, gray otherwise, with a dashed red line at the cutoff.

Usage

```
## S3 method for class 'gg_beta_uvarpro'
plot(x, labels = NULL, ...)
```

Arguments

`x`	A `gg_beta_uvarpro` object from `gg_beta_uvarpro()`.
`labels`	Optional variable labels for the variable axis. One of: a named character vector (`c(wt = "Weight")`); a labelled data frame, whose `attr(col, "label")` values are read; or a two-column `key`/`label` data frame. Variables with no label keep their raw name. Defaults to `NULL` (raw names).
`...`	Not currently used.

Value

A `ggplot` object.

Reading the chart

Each bar is the average magnitude of a per-region lasso coefficient for that variable, computed by `varPro::get.beta.entropy()` over the unsupervised entropy regions of a `varPro::uvarpro()` fit. There is no response: the score measures how strongly a variable is reconstructed by the others within released regions, i.e. an unsupervised importance / redundancy signal rather than a predictive one. As with `gg_beta_varpro()`, the numeric scale carries the predictors' units, so bar lengths are comparable within a data set but not blindly across variables on very different scales.

See Also

`gg_beta_uvarpro()`, `gg_beta_varpro()`, `gg_udependent()`.

Examples

```
if (requireNamespace("varPro", quietly = TRUE)) {
  set.seed(1)
  o <- varPro::uvarpro(mtcars, ntree = 50)
  plot(gg_beta_uvarpro(o))
}
```

`plot.gg_beta_varpro` *Plot a* `gg_beta_varpro` *object*

Description

Horizontal bar chart of the mean absolute coefficient $\mathrm{mean}(|\hat{\beta}|)$ per variable, sorted descending so the eye lands on the top variable first. Bars filled blue when above the selection cutoff, gray otherwise. Dashed red line marks the cutoff.

Usage

```
## S3 method for class 'gg_beta_varpro'
plot(x, labels = NULL, ...)
```

Arguments

`x`	A `gg_beta_varpro` object from `gg_beta_varpro()`.
`labels`	Optional variable labels for the variable axis. One of: a named character vector (`c(wt = "Weight")`); a labelled data frame, whose `attr(col, "label")` values are read; or a two-column `key`/`label` data frame. Variables with no label keep their raw name. Defaults to `NULL` (raw names).
`...`	Not currently used.

Value

A `ggplot` object.

Reading the chart

Each bar is the average magnitude of a per-rule lasso coefficient for that variable. **The numeric scale carries the predictor's units.** If "age" is in years and "creatinine" is in mg/dL, a longer bar for age does not mean age is "more important" in any unit-free sense. Comparisons across data sets or across variables with very different units require keeping the units context in mind. Within one data set, bars are comparable up to that unit caveat.

Variables above the cutoff are colored blue and flagged `selected`; variables below are gray. Lasso shrinkage can drive a rule's $\hat{\beta}$ to exactly zero; those rules are kept in the average, so a variable with many shrunk-to-zero rules will sit lower in the ranking than one whose released coefficients are consistently non-zero.

For a classification fit, variables are sorted by `mean(|sum-of-class-beta|)` descending and that ordering is shared across every facet, so rows line up between classes for visual comparison. Each facet has its own cutoff line.

What this tells you

Use the bar chart as a selection ranking, not as an effect-size axis. Pair it with `gg_varpro()` to see where split-strength importance and local lasso-beta importance agree or disagree; disagreement is often the interesting signal.

See Also

gg_beta_varpro().

Examples

```
if (requireNamespace("varPro", quietly = TRUE)) {
  set.seed(1)
  v <- varPro::varpro(mpg ~ ., data = mtcars, ntree = 50)
  plot(gg_beta_varpro(v))
}
```

---

plot.gg_brier *Plot a* gg_brier *object*

Description

Draws the time-resolved Brier score or the running CRPS from a gg_brier object. The curve moves across the event-time grid on the x-axis; lower values mean the forest's predicted survival probabilities are closer to what actually happened. Think of 0 as "perfect" and roughly 0.25 as "uninformative"; a forest that predicts 0.5 for every subject regardless of prognosis would sit near that reference. It is not a ceiling; a forest can do worse.

Usage

```
## S3 method for class 'gg_brier'
plot(x, type = c("brier", "crps"), envelope = FALSE, ...)
```

Arguments

x	A gg_brier object.
type	Which series to plot: "brier" (default) or "crps".
envelope	Logical. When TRUE, overlays a ribbon spanning the 15th-85th percentile of per-subject Brier (or running CRPS) contributions at each time, around the overall line. When FALSE (default), draws the overall series only.
...	Extra arguments forwarded to geom_line().

Details

Set envelope = TRUE to add a ribbon around the overall curve spanning the 15th to 85th percentile of the per-subject Brier contributions at each time. The ribbon shows how heterogeneous the scoring is across subjects: a narrow ribbon means most subjects are predicted equally well (or equally poorly); a wide ribbon means a minority of subjects are driving the average.

Value

A ggplot object.

See Also

gg_brier, get.brier.survival

Examples

```
library(survival)   # Surv() must be on the search path for rfsrc()
data(pbc, package = "randomForestSRC")
rf <- randomForestSRC::rfsrc(Surv(days, status) ~ ., data = pbc,
                             nsplit = 10, ntree = 100)
gg_dta <- gg_brier(rf)
plot(gg_dta)
plot(gg_dta, type = "crps")
plot(gg_dta, envelope = TRUE)   # adds 15-85% envelope
```

---

plot.gg_error *Plot a gg_error object*

Description

A plot of the cumulative OOB error rates of the random forest as a function of number of trees.

Usage

```
## S3 method for class 'gg_error'
plot(x, ...)
```

Arguments

x	A gg_error object created from either a rfsrc or a randomForest object. A raw forest object may also be supplied and will be passed through gg_error automatically before plotting.
...	Extra arguments forwarded to the underlying ggplot2 geometry calls (e.g. size, linetype).

Details

The gg_error plot tracks the convergence of the randomForest. It reproduces the error plot from the plot.rfsrc function.

Value

A ggplot object with ntree on the x-axis and OOB error rate on the y-axis. Single-outcome forests (regression, survival) produce a single line; multi-outcome forests (classification) produce one colored line per class.

References

Breiman L. (2001). Random forests, Machine Learning, 45:5-32.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, Rnews, 7(2):25-31.

Ishwaran H, Kogalur U (2026). Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC). R package version 3.6.2. https://cran.r-project.org/package=randomForestSRC

See Also

gg_error rfsrc plot.rfsrc

Examples

```
## Examples from RFSRC package...
## ------------------------------------------------------------
## classification example
## ------------------------------------------------------------
## ------------- iris data
## You can build a randomForest
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris,
  ntree = 100,
  forest = TRUE,
  importance = TRUE,
  tree.err = TRUE,
  save.memory = TRUE)

# Get a data.frame containing error rates
gg_dta <- gg_error(rfsrc_iris)

# Plot the gg_error object
plot(gg_dta)

## RandomForest example
rf_iris <- randomForest::randomForest(Species ~ .,
  data = iris,
  forest = TRUE,
  importance = TRUE,
  tree.err = TRUE,
  save.memory = TRUE
)
gg_dta <- gg_error(rf_iris)
plot(gg_dta)

gg_dta <- gg_error(rf_iris, training = TRUE)
plot(gg_dta)
## ------------------------------------------------------------
## Regression example
## ------------------------------------------------------------
## ------------- airq data
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ .,
  data = airquality,
  ntree = 100,
```

```
  na.action = "na.impute",
  forest = TRUE,
  importance = TRUE,
  tree.err = TRUE,
  save.memory = TRUE
)

# Get a data.frame containing error rates
gg_dta <- gg_error(rfsrc_airq)

# Plot the gg_error object
plot(gg_dta)


## ------------- Boston data
if (requireNamespace("MASS", quietly = TRUE)) {
  data(Boston, package = "MASS")
  Boston$chas <- as.logical(Boston$chas)
  rfsrc_boston <- randomForestSRC::rfsrc(medv ~ .,
    data = Boston,
    ntree = 100,
    forest = TRUE,
    importance = TRUE,
    tree.err = TRUE,
    save.memory = TRUE
  )

  # Get a data.frame containing error rates
  gg_dta <- gg_error(rfsrc_boston)

  # Plot the gg_error object
  plot(gg_dta)
}

## ------------- mtcars data
rfsrc_mtcars <- randomForestSRC::rfsrc(mpg ~ ., data = mtcars,
  ntree = 100,
  importance = TRUE,
  save.memory = TRUE,
  forest = TRUE,
  tree.err = TRUE)

# Get a data.frame containing error rates
gg_dta<- gg_error(rfsrc_mtcars)

# Plot the gg_error object
plot(gg_dta)


## ------------------------------------------------------------
## Survival example
## ------------------------------------------------------------
## ------------- veteran data
```

```
## randomized trial of two treatment regimens for lung cancer
data(veteran, package = "randomForestSRC")
rfsrc_veteran <- randomForestSRC::rfsrc(Surv(time, status) ~ ., data = veteran,
                      ntree = 100, tree.err = TRUE)

gg_dta <- gg_error(rfsrc_veteran)
plot(gg_dta)

## ------------- pbc data
data(pbc, package = "randomForestSRC", envir = environment())
# Recode: binary columns to logical, low-cardinality columns to factors.
for (ind in seq_len(dim(pbc)[2])) {
  if (!is.factor(pbc[, ind])) {
    if (length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 2) {
      if (sum(range(pbc[, ind], na.rm = TRUE) == c(0, 1)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
    }
  } else {
    if (length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 2) {
      if (sum(sort(unique(pbc[, ind])) == c(0, 1)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
      if (sum(sort(unique(pbc[, ind])) == c(FALSE, TRUE)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
    }
  }
  if (!is.logical(pbc[, ind]) &&
    length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 5) {
    pbc[, ind] <- factor(pbc[, ind])
  }
}
# Convert age from days to years
pbc$age <- pbc$age / 364.24
pbc$years <- pbc$days / 364.24
pbc <- pbc[, -which(colnames(pbc) == "days")]
pbc$treatment <- as.numeric(pbc$treatment)
pbc$treatment[which(pbc$treatment == 1)] <- "DPCA"
pbc$treatment[which(pbc$treatment == 2)] <- "placebo"
pbc$treatment <- factor(pbc$treatment)
# Split into a training set (assigned treatment) and a test set (not assigned)
dta_train <- pbc[-which(is.na(pbc$treatment)), ]
pbc_test <- pbc[which(is.na(pbc$treatment)), ]

#========
# build the forest:
rfsrc_pbc <- randomForestSRC::rfsrc(
  Surv(years, status) ~ .,
 dta_train,
 ntree = 100,
 nsplit = 10,
 na.action = "na.impute",
```

```
 tree.err = TRUE,
 forest = TRUE,
 importance = TRUE,
 save.memory = TRUE
)


gg_dta <- gg_error(rfsrc_pbc)
plot(gg_dta)
```

`plot.gg_isopro` *Plot a varPro isolation-forest anomaly score*

Description

Renders a `gg_isopro` object as a ranked elbow (observations sorted by anomaly score), a density of scores, or both side-by-side. Optionally annotates a threshold either in score-space (`threshold`) or in quantile-space (`top_n_pct`).

Usage

```
## S3 method for class 'gg_isopro'
plot(
  x,
  panel = c("both", "elbow", "density"),
  threshold = NULL,
  top_n_pct = NULL,
  ...
)
```

Arguments

`x`	A `gg_isopro` object from `gg_isopro`.
`panel`	One of `"both"` (default: a `patchwork` of elbow + density), `"elbow"`, or `"density"` (each returns a single `ggplot`).
`threshold`	Numeric in `[0, 1]`, or `NULL` (default). If set, draws a reference line at that `howbad` value on the elbow and density.
`top_n_pct`	Numeric in `(0, 100)`, or `NULL` (default). If set, resolves to the matching `howbad` quantile and draws the same reference line. If both `threshold` and `top_n_pct` are supplied, `threshold` wins with a `message()`.
`...`	Currently unused.

Value

A `ggplot` (single panel) or a `patchwork` (`panel = "both"`).

Reading the elbow

The elbow plot is the canonical anomaly-detection picture. The x-axis is observation rank (observations sorted from most ordinary to most anomalous) and the y-axis is the howbad score. For a clean population the curve sits flat near zero across the bulk of the data and then bends sharply upward in the right tail; that bend is where the anomalous observations live. The point of the plot is not to read off a single score, it is to *see where the curve breaks*. Pick a cutoff there. Pass it back in as `threshold` (for a score) or `top_n_pct` (for "the top 5%") and the plot draws a dashed reference line so you can record the choice you made.

Reading the density

The density panel is the same scores viewed as a distribution. A single tight mode near zero with a long thin right tail is the picture you hope for: bulk of the data ordinary, a few clear anomalies. A bimodal density says you may have two populations rather than one clean cluster plus outliers, and the cutoff question becomes harder. Either way, this panel is a sanity check on what the elbow suggests.

Comparing methods

When the input object carries a `method` column (because you bound several `gg_isopro` calls together), both panels color by method automatically. The point of comparing `"rnd"`, `"unsupv"`, and `"auto"` is not to pick a winner from the figure alone, it is to see whether the methods agree on which observations are anomalous. Curves that overlap in the right tail and elbow at roughly the same rank are telling you the same story three ways. Curves that diverge are telling you the score is method-sensitive, which is itself useful information.

See Also

`gg_isopro`, `isopro`

`plot.gg_ivarpro` *Plot a* `gg_ivarpro` *object*

Description

Branches on the presence of `which_obs` provenance and the `class` column. Distribution view: jittered points showing per-observation local importances per variable. Per-observation view: horizontal bar chart of one observation's local importances across variables. Classification: faceted by class unless `which_class` collapses to a single class.

Usage

```
## S3 method for class 'gg_ivarpro'
plot(x, labels = NULL, ...)
```

Arguments

x	A gg_ivarpro object from gg_ivarpro().
labels	Optional variable labels for the variable axis. One of: a named character vector (c(wt = "Weight")); a labelled data frame, whose attr(col, "label") values are read; or a two-column key/label data frame. Variables with no label keep their raw name. Defaults to NULL (raw names).
...	Not currently used.

Value

A ggplot object.

Reading the chart

Each point in the distribution view is one observation's local importance for that variable. Variables are sorted by descending mean(|local_imp|). The cutoff line picks the variables whose local importance is, on average, large enough to flag. For a classification fit, every facet shares the same row order so you can read across.

For a classification fit, variables are sorted by descending mean(|local_imp|) across all (obs, class) rows and that ordering is shared across every facet, so rows line up between classes for visual comparison. Each facet has its own cutoff line.

The per-observation view (which_obs) is a horizontal bar chart of one observation's local importances; bars below the cutoff are gray, above are blue. The visual resembles a SHAP waterfall, but the values are release-rule contributions: scaled per-rule contrasts on observed data, not Shapley values and not permutation-based.

See Also

gg_ivarpro().

Examples

```
if (requireNamespace("varPro", quietly = TRUE) &&
    requireNamespace("MASS", quietly = TRUE)) {
  set.seed(1)
  v <- varPro::varpro(medv ~ ., data = MASS::Boston, ntree = 50)
  plot(gg_ivarpro(v))
}
```

plot.gg_partial *Plot a* gg_partial *object*

Description

Turns a gg_partial object into a ggplot2 figure. Each curve is a partial dependence trace, the forest's average prediction as one predictor is swept across its range while the rest are marginalized over the training data. Continuous predictors appear as line plots. Categorical predictors appear as box plots, one box per level, drawn from the per-observation predictions that `plot.variable()` returns for them, so the box shows how the prediction varies across the training data at that level. Both panels are faceted by variable name so you can compare the shape and scale of each variable's effect at a glance.

Usage

```
## S3 method for class 'gg_partial'
plot(x, labels = NULL, ...)
```

Arguments

x	A gg_partial object (output of gg_partial).
labels	Optional variable labels for the facet strips. One of: a named character vector (`c(bpd_last = "BP Diastole")`); a labelled data frame, whose `attr(col, "label")` values are read; or a two-column key/label data frame. Variables with no label keep their raw name. Defaults to `NULL` (raw names).
...	Not currently used; reserved for future arguments.

Details

When a `model` label was attached in `gg_partial()`, lines are colored by model, which is handy for overlaying results from two forests (e.g., one tuned, one default) in the same figure.

Value

A `ggplot` (or `patchwork`) object. When only one variable type is present a single `ggplot` is returned. When both continuous and categorical variables are present the two panels are combined vertically via `patchwork::wrap_plots()`, which also satisfies `inherits(p, "ggplot")`.

See Also

gg_partial, plot.gg_variable

Examples

```
set.seed(42)
airq <- na.omit(airquality)
rf <- randomForestSRC::rfsrc(Ozone ~ ., data = airq, ntree = 50)
pv <- randomForestSRC::plot.variable(rf, partial = TRUE, show.plots = FALSE)
pd <- gg_partial(pv)
plot(pd)
```

---

plot.gg_partial_rfsrc *Plot a* gg_partial_rfsrc *object*

Description

Renders the partial dependence curves from gg_partial_rfsrc as a ggplot2 figure. The layout adapts automatically to what the object contains.

Usage

```
## S3 method for class 'gg_partial_rfsrc'
plot(x, labels = NULL, ...)
```

Arguments

x	A gg_partial_rfsrc object.
labels	Optional variable labels for the facet strips. One of: a named character vector (c(bpd_last = "BP Diastole")); a labelled data frame, whose attr(col, "label") values are read; or a two-column key/label data frame. Variables with no label keep their raw name. Defaults to NULL (raw names).
...	Not currently used.

Details

For a standard regression or classification forest, continuous predictors are drawn as line plots and categorical predictors as box plots, both faceted by variable name, the same arrangement as plot.gg_partial. The categorical data hold one prediction per training observation per level, not their average, so each box shows the spread of the prediction at that level and its middle line the median. For a survival forest the boxes are filled by time horizon, and with xvar2.name by the level of the second variable. When a survival forest has both, the fill is the time horizon and each level of the second variable gets its own panel.

For a survival forest, each call to partial.rfsrc returns a predicted quantity (survival probability, cumulative hazard function, or mortality) at one or more chosen time horizons. When a time column is present in the data, each horizon becomes a separate colored curve over the predictor's value, still faceted by variable. Mortality has no time horizon, so it is drawn as a single curve. The y-axis label ("Predicted Survival", "Predicted CHF", or "Predicted Mortality") tracks the partial.type attribute set by gg_partial_rfsrc().

For a two-variable interaction surface (when `xvar2.name` was supplied to `gg_partial_rfsrc`), the secondary variable's levels become separate colored lines, faceted by the primary predictor. On a survival forest the color is already the time horizon, so each level of the secondary variable gets its own panel instead, as in the categorical panel.

Value

A `ggplot` (or `patchwork`) object. When both continuous and categorical variables are present the two panels are combined vertically via `patchwork::wrap_plots()`.

See Also

`gg_partial_rfsrc`, `plot.gg_partial`

Examples

```
## ------------------------------------------------------------
## Regression forest -- one continuous curve per variable
## ------------------------------------------------------------
set.seed(42)
airq <- na.omit(airquality)
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ ., data = airq, ntree = 50)

pd <- gg_partial_rfsrc(rfsrc_airq, xvar.names = c("Wind", "Temp"),
                       n_eval = 10)
plot(pd)


## ------------------------------------------------------------
## Survival forest -- one curve per requested time horizon,
## faceted by variable. Y-axis label tracks `partial.type`.
## ------------------------------------------------------------
# randomForestSRC's formula parser requires the unqualified Surv() symbol;
# it Depends on `survival`, so Surv is on the search path once
# randomForestSRC is loaded.
data(veteran, package = "randomForestSRC")
set.seed(42)
rfsrc_v <- randomForestSRC::rfsrc(Surv(time, status) ~ .,
                                  data = veteran, ntree = 50)
ti  <- rfsrc_v$time.interest
t30 <- ti[which.min(abs(ti - 30))]
t90 <- ti[which.min(abs(ti - 90))]

# Default partial.type = "surv" -> y-axis "Predicted Survival"
pd_s <- gg_partial_rfsrc(rfsrc_v, xvar.names = "age",
                         partial.time = c(t30, t90), n_eval = 8)
plot(pd_s)

# partial.type = "chf" -> y-axis "Predicted CHF"
pd_c <- gg_partial_rfsrc(rfsrc_v, xvar.names = "age",
                         partial.time = c(t30, t90),
                         partial.type = "chf", n_eval = 8)
```

```
plot(pd_c)
```

---

plot.gg_partial_varpro *Plot a* gg_partial_varpro *object*

Description

Draws the partial dependence curves from the list that gg_partial_varpro returns. Continuous predictors get overlaid line curves, one per effect type; categorical predictors get side-by-side boxplots. Survival path-C objects (the ones you get when `scale %in% c("surv","chf")` was passed to the extractor) are handed off to plot.gg_partial_rfsrc for drawing.

Usage

```
## S3 method for class 'gg_partialpro'
plot(x, type = c("parametric", "nonparametric", "causal"), labels = NULL, ...)

## S3 method for class 'gg_partial_varpro'
plot(
  x,
  type = c("parametric", "nonparametric", "causal"),
  labels = NULL,
  ...,
  which = c("both", "continuous", "categorical"),
  panels = NULL,
  points = FALSE,
  smooth = FALSE,
  palette = "black",
  ncol = NULL,
  point_size = 1.1,
  point_alpha = 0.55,
  linewidth = 0.5,
  complement = FALSE,
  ylim = NULL
)
```

Arguments

x	A gg_partial_varpro object.
type	Character vector; one or more of "parametric", "nonparametric", "causal". Defaults to the two level curves, "parametric" and "nonparametric". "causal" is a centered contrast rather than a level, so it has to be asked for, and it is shown only on an unbounded scale. Ignored for path-C objects.

`labels` Optional variable labels for the facet strips. One of: a named character vector (`c(bpd_last = "BP Diastole")`); a labelled data frame, whose `attr(col, "label")` values are read; or a two-column `key`/`label` data frame. Variables with no label keep their raw name. Defaults to `NULL` (raw names).

`...` Unused for path-A objects; forwarded to `plot.gg_partial_rfsrc` for path-C objects.

`which` Character; which frame to draw. `"both"` (default) keeps the historical behaviour, returning a **patchwork** stack when the object carries both continuous and categorical variables. `"continuous"` or `"categorical"` returns a bare `ggplot` for that frame alone. Use it when you want to add scales or themes with `+`: on a patchwork `+` reaches only the last panel, so `which` is how you get one plot to modify. Implied by `panels`.

`panels` Optional data frame giving per-panel scales, one row per panel. Supplying it selects the continuous frame and switches rendering from `facet_wrap()` to **patchwork**, which is the only way to vary the x scale between panels. Only `name` is required; every other column is optional and an absent one leaves that decision to **ggplot2**.

name variable, matched against `x$continuous$name`. A name the frame does not carry is an error rather than a silent drop.

xlab panel x axis title. Falls back to the `labels` value for that variable, then to `name`.

xmin, xmax clipped range, applied with `coord_cartesian()` so points outside are hidden rather than dropped from the smooth. Supplying both also removes the axis padding.

xby tick spacing, expanded to `seq(xmin, xmax, xby)`.

span per-panel `geom_smooth()` span.

Panels are drawn in row order, so the frame pins panel order explicitly. A variable absent from `panels` is not drawn; selecting a subset is the usual reason to supply it.

The y axis is shared across panels, matching the facet route, and is computed over the selected variables only, so dropping a variable rescales the figure. Only the x scale varies between panels; four partial dependence curves on four different y ranges would not compare.

`points` Logical; add the grid-point values as points. Default `FALSE`.

`smooth` Logical; draw a `geom_smooth()` loess instead of the line. Default `FALSE`. `parametric` is already partialpro's local-polynomial fit, so smoothing it is a smooth of a smooth; this is here for the raw-looking figure some journals ask for, not as a better estimate.

`palette` Character; the effect-type colour or fill scale. Defaults to `"black"`: this package's figures are made for manuscripts, and `linetype` is mapped to the effect type as well, so the three estimators stay legible as solid, dotted and dashed with no colour at all. `"mono"` is a synonym, and `"grey"` / `"gray"` give a flat grey. Any ColorBrewer palette name (e.g. `"Set1"`) routes to the brewer scale instead, which is worth reaching for when you are comparing two or three estimators on screen and colour separates them faster than a dash pattern. `NULL` keeps **ggplot2**'s own scale.

`ncol` Integer; columns in the `panels` layout. `NULL` (default) lets **patchwork** choose.

`point_size`, `point_alpha`
Numeric; size and alpha of the points drawn when `points = TRUE`. Defaults `1.1` and `0.55`. Print figures usually want a smaller point than a screen figure.

`linewidth` Numeric; width of the line or smooth. Default `0.5`, **ggplot2**'s own.

`complement` Logical; plot $1 - p$ instead of p, and prefix the y axis label with `"1 - "`. Use it when the fit targets the class you do *not* want on the axis (a model of weaning failure read as the probability of weaning success, say), so you do not have to recompute `partialpro` against the other target. Requires a probability scale (`scale = "prob"` or `"surv"`); on the additive, multiplicative and unbounded scales $1 - x$ has no referent and this is an error rather than a silent no-op. Default `FALSE`.

`ylim` Numeric length-2; the shared y range for every panel. `NULL` (default) takes the range of the plotted values, which is what the facet route has always done. Supply it to pin a scale that means something independent of the data: `c(0, 1)` on a probability scale, say, so a flat curve reads as flat rather than filling the panel. It cannot be set from outside: on the `panels` route a `coord_cartesian()` added with `&` replaces the per-panel coordinate system and takes the per-panel x ranges down with it, and `scale_y_continuous(limits = )` is overridden by that coordinate system.

Details

Ensemble mortality (scale = "mortality"): when the provenance scale is `"mortality"`, the y-axis is labeled *"Ensemble mortality (expected events)"*. The wording is deliberate: this is an **unbounded relative-risk score**, not a survival probability and not $1 - S(t)$ (Ishwaran, Kogalur, Blackstone & Lauer, 2008 doi:10.1214/08-AOAS169).

Value

A `ggplot` (or `patchwork`) object.

Reading the partial dependence

For a continuous variable the x-axis is the variable's grid of values and the y-axis is the partial prediction; each of the three effect types (`parametric`, `nonparametric`, `causal`) is drawn as its own line. The shape of the line is the story: a clear slope says the model uses the variable, a flat line says it essentially does not, and a U-shape or a threshold says the effect is nonlinear in a way a single coefficient would miss. For a categorical variable the picture is a boxplot per level; here the eye is looking at level-to-level shifts in the center of each box.

Where the three effect types track each other, the parametric story is a fair summary of what the forest is doing. Where they fan apart (typically the parametric curve smoother than the nonparametric, or the causal curve flatter than either) the variable is one to inspect more carefully before reading a single effect off the plot.

What this tells you

Use these curves to describe how the model uses each variable, not to claim how the world works. They are a window into the fitted relationship; they do not by themselves establish that intervening on the variable would move the outcome. For survival path-C (`scale = "chf"`), the y-axis is on the cumulative-hazard scale.

Reading a probability curve (scale = "prob")

The y-axis is $P(Y = \text{target})$, the model's predicted probability of the target class as the focal variable varies (others held at their UVT-plausible average). `"odds"` and `"logodds"` are the same curve on the odds and log-odds scales. The `causal` curve is a contrast (below) and is *not* shown on `"prob"`/`"odds"`; use `"logodds"` to see it.

Reading a survival-probability curve (scale = "surv")

The y-axis is $S(\tau \mid x)$, the predicted probability of surviving past τ, bounded in $[0, 1]$ and read in the model's time units. Higher is better (more survival). τ defaults to the median follow-up time when not supplied.

What the causal curve is, and when to use it

`causal` is the **baseline-subtracted local effect**, varPro's virtual- ("digital-") twins estimator (Ishwaran & Blackstone, 2025). It shows how the prediction shifts as the focal variable moves away from the reference grid point, with the other covariates held at on-manifold (UVT-plausible) values; it is a **contrast** (it starts at 0), not a level. Use it when you want the local effect (change-from-baseline) rather than the absolute predicted level, and as a cross-check on the parametric and nonparametric curves. It is varpro's local estimator *within the fitted model*, **not a structural causal claim** about the data-generating process. Because it is a contrast it cannot share a probability/odds axis with the absolute curves, so it is shown only on the additive scales (`"logodds"`, `"mortality"`, `"rmst"`).

Reading an RMST curve (scale = "rmst")

The y-axis is restricted mean survival time at horizon τ, $\mathrm{RMST}(\tau) = \int_0^\tau S(t)\,dt$: the **expected event-free time during the first τ time-units**, the area under the survival curve out to τ. Read it in the **model's own time units**, where it is bounded by $0 \le \mathrm{RMST}(\tau) \le \tau$.

Two things follow. First, τ must be given in the fit's time units; a τ past the largest event time truncates to the full restricted mean and stops changing. Second, higher is better here (more time event-free), which is the opposite of the ensemble-mortality scale.

A continuous variable's curve sloping *up* means higher values of that covariate buy you *more* restricted-mean event-free time within τ (with the other covariates held at their UVT-plausible average); a flat curve means the covariate does not move it. Unlike ensemble mortality, RMST reads on a directly clinical scale, "so many event-free time-units within τ", which is usually the one you want to report.

References

Ishwaran H, Kogalur UB, Blackstone EH, Lauer MS (2008). Random survival forests. *The Annals of Applied Statistics*, **2**(3), 841–860. doi:10.1214/08AOAS169.

Ishwaran H, Blackstone EH (2025). Harnessing the power of virtual (digital) twins: Graphical causal tools for understanding patient and hospital differences. *Computational and Structural Biotechnology Journal*, **28**, 312.

See Also

gg_partial_varpro

Examples

```
set.seed(42)
n_obs <- 30; n_pts <- 15
mock_data <- list(
  age = list(
    xvirtual    = seq(30, 80, length.out = n_pts),
    xorg        = sample(seq(30, 80, by = 5), n_obs, replace = TRUE),
    yhat.par    = matrix(rnorm(n_obs * n_pts), nrow = n_obs),
    yhat.nonpar = matrix(rnorm(n_obs * n_pts), nrow = n_obs),
    yhat.causal = matrix(rnorm(n_obs * n_pts), nrow = n_obs)
  ),
  sex = list(
    xvirtual    = c(0, 1),
    xorg        = sample(c(0, 1), n_obs, replace = TRUE),
    yhat.par    = matrix(rnorm(n_obs * 2), nrow = n_obs),
    yhat.nonpar = matrix(rnorm(n_obs * 2), nrow = n_obs),
    yhat.causal = matrix(rnorm(n_obs * 2), nrow = n_obs)
  )
)
pp <- gg_partial_varpro(mock_data, scale = "logodds")
plot(pp)
plot(pp, type = "parametric")

## The continuous frame alone, so `+` reaches the plot you meant.
plot(pp, which = "continuous") + ggplot2::labs(title = "Continuous only")

## Per-panel scales.  Only 'name' is required; the rest tune one axis each.
spec <- data.frame(name = "age", xlab = "Age (years)",
                   xmin = 30, xmax = 80, xby = 10, span = 0.6)
plot(pp, type = "parametric", panels = spec, points = TRUE, smooth = TRUE)

## Colour earns its place when several estimators share a panel: two curves
## separate faster by hue than by dash pattern.  Any ColorBrewer name works.
plot(pp, type = c("parametric", "nonparametric"), palette = "Set1")

## The default is monochrome, for print.
plot(pp, type = c("parametric", "nonparametric"))

## A patchwork takes `&`, not `+`, to reach every panel.
plot(pp, type = "parametric", panels = spec) & ggplot2::theme_minimal()

## complement = TRUE reads a failure model as its success probability.
## (Mock data has no fit to restore subject levels from, hence the warning.)
```

```
pp_prob <- suppressWarnings(gg_partial_varpro(mock_data, scale = "prob"))
plot(pp_prob, type = "parametric", complement = TRUE)
```

`plot.gg_rfsrc` *Predicted response plot from a* `gg_rfsrc` *object.*

Description

Visualizes the ensemble predictions extracted by `gg_rfsrc`. By default those are out-of-bag (OOB) predictions, the forest's built-in cross-validation estimate, which averages only over the trees that left a given observation out of their bootstrap sample.

Usage

```
## S3 method for class 'gg_rfsrc'
plot(x, notch = TRUE, time_units = NULL, ...)
```

Arguments

`x`	A `gg_rfsrc` object, or a raw `rfsrc` object (which will be passed through `gg_rfsrc` automatically before plotting).
`notch`	Logical; whether to draw notched boxplots for regression and classification forests (default `TRUE`). Set `notch = FALSE` to suppress notches when sample sizes are too small for reliable confidence intervals on the median.
`time_units`	Optional name of the time unit the forest was fit in, used only in the survival x axis title. The default titles the axis `"time"`; `time_units = "days"` makes that `"time (days)"`. Nothing on an `rfsrc` object records the unit, so the package cannot infer it. Defaults to `NULL` (no unit printed).
`...`	Additional arguments forwarded to the underlying `ggplot2` geometry calls. Commonly useful arguments include: `alpha` Numeric in $[0, 1]$; point/ribbon transparency. For survival plots with confidence bands the ribbon alpha is automatically halved relative to the value supplied here. `size` Point or line size passed to `geom_jitter`, `geom_step`, etc. Arguments that control `gg_rfsrc` (e.g. `conf.int`, `surv_type`, `by`) should be applied when constructing the `gg_rfsrc` object before calling `plot()`.

Details

The geometry adapts to the forest family. For regression or classification, you get a jitter-and-boxplot: every observation is a dot placed at its OOB predicted value, colored by observed response, with a notched boxplot overlaid to show the central tendency and spread. For a survival forest, each observation contributes one ensemble survival curve (or CHF / mortality, whichever `surv_type` was chosen in `gg_rfsrc`); the bundle of step functions shows the spread of predicted risk across the cohort. Pass `by` to `gg_rfsrc` beforehand to color curves by a predictor group, or `conf.int` to replace the individual curves with a mean curve and bootstrap ribbon.

Value

A ggplot object. The plot appearance depends on the forest family stored in x:

Regression ("regr") Jitter + notched boxplot of OOB predicted values. If a group column is present the x-axis shows each group label; otherwise observations are collapsed to a single x-position.

Classification ("class") Binary: jitter + notched boxplot of the predicted class probability. Multi-class: jitter plot with one panel per class (class probabilities in long form).

Survival ("surv") Step curves of the ensemble survival function. When gg_rfsrc was called with conf.int, a shaded ribbon is added. When called with by, curves are colored by group.

References

Breiman L. (2001). Random forests, Machine Learning, 45:5-32.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, Rnews, 7(2):25-31.

Ishwaran H, Kogalur U (2026). Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC). R package version 3.6.2. https://cran.r-project.org/package=randomForestSRC

See Also

gg_rfsrc rfsrc randomForest

Examples

```
## ------------------------------------------------------------
## classification example
## ------------------------------------------------------------
## -------- iris data
# Build a small classification forest (ntree=50 keeps example fast)
set.seed(42)
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris, ntree = 50)
gg_dta <- gg_rfsrc(rfsrc_iris)

plot(gg_dta)

## ------------------------------------------------------------
## Regression example
## ------------------------------------------------------------
## -------- air quality data
# na.action = "na.impute" handles missing Ozone / Solar.R values
set.seed(42)
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ ., data = airquality,
                    na.action = "na.impute", ntree = 50)
gg_dta <- gg_rfsrc(rfsrc_airq)

plot(gg_dta)

## -------- mtcars data
set.seed(42)
rfsrc_mtcars <- randomForestSRC::rfsrc(mpg ~ ., data = mtcars, ntree = 50)
```

```
gg_dta <- gg_rfsrc(rfsrc_mtcars)

plot(gg_dta)

## ------------------------------------------------------------
## Survival example
## ------------------------------------------------------------
## -------- veteran data
## randomized trial of two treatment regimens for lung cancer
data(veteran, package = "randomForestSRC")
set.seed(42)
rfsrc_veteran <- randomForestSRC::rfsrc(Surv(time, status) ~ ., data = veteran, ntree = 50)
gg_dta <- gg_rfsrc(rfsrc_veteran)
plot(gg_dta)

# With 95% pointwise bootstrap confidence bands
gg_dta <- gg_rfsrc(rfsrc_veteran, conf.int = .95)
plot(gg_dta)

# Stratified by treatment arm
gg_dta <- gg_rfsrc(rfsrc_veteran, by = "trt")
plot(gg_dta)

## -------- pbc data (larger dataset -- forest fit skipped on CRAN)
data(pbc, package = "randomForestSRC", envir = environment())
# Recode: binary columns to logical, low-cardinality columns to factors.
for (ind in seq_len(dim(pbc)[2])) {
  if (!is.factor(pbc[, ind])) {
    if (length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 2) {
      if (sum(range(pbc[, ind], na.rm = TRUE) == c(0, 1)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
    }
  } else {
    if (length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 2) {
      if (sum(sort(unique(pbc[, ind])) == c(0, 1)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
      if (sum(sort(unique(pbc[, ind])) == c(FALSE, TRUE)) == 2) {
        pbc[, ind] <- as.logical(pbc[, ind])
      }
    }
  }
  if (!is.logical(pbc[, ind]) &&
    length(unique(pbc[which(!is.na(pbc[, ind])), ind])) <= 5) {
    pbc[, ind] <- factor(pbc[, ind])
  }
}
# Convert age from days to years
pbc$age <- pbc$age / 364.24
pbc$years <- pbc$days / 364.24
pbc <- pbc[, -which(colnames(pbc) == "days")]
pbc$treatment <- as.numeric(pbc$treatment)
```

```
pbc$treatment[which(pbc$treatment == 1)] <- "DPCA"
pbc$treatment[which(pbc$treatment == 2)] <- "placebo"
pbc$treatment <- factor(pbc$treatment)
# Split into a training set (assigned treatment) and a test set (not assigned)
dta_train <- pbc[-which(is.na(pbc$treatment)), ]
pbc_test <- pbc[which(is.na(pbc$treatment)), ]


set.seed(42)
rfsrc_pbc <- randomForestSRC::rfsrc(
  Surv(years, status) ~ .,
  dta_train,
  ntree = 100,
  nsplit = 10,
  na.action = "na.impute",
  forest = TRUE,
  importance = TRUE,
  save.memory = TRUE
)

gg_dta <- gg_rfsrc(rfsrc_pbc)
plot(gg_dta)

gg_dta <- gg_rfsrc(rfsrc_pbc, conf.int = .95)
plot(gg_dta)

gg_dta <- gg_rfsrc(rfsrc_pbc, by = "treatment")
plot(gg_dta)
```

`plot.gg_rhf` *Plot Random Hazard Forest hazard / cumulative-hazard curves*

Description

Draws case-specific ensemble curves from a `gg_rhf()` object: hazard (default) or cumulative hazard, one line per case selected by `idx`.

Usage

```
## S3 method for class 'gg_rhf'
plot(x, idx = NULL, hazard.only = TRUE, ...)
```

Arguments

`x`	A `gg_rhf` object from `gg_rhf()`.
`idx`	Integer vector of case ids (matched against the `id` column) to draw. `NULL` (default) draws every case.

`hazard.only`	Logical; `TRUE` (default) plots the hazard, `FALSE` plots the cumulative hazard.
`...`	Not currently used.

Details

Cells the forest left undefined are dropped before drawing. From **randomForestRHF** 2.0.0 the hazard is `NA` outside each case's observed `(start, stop]` path, so a hazard curve ends with that case's follow-up, and a case whose hazard is masked throughout is left out of the legend rather than shown as an empty one. From 2.0.3 the cumulative hazard is masked too, though on a different rule: it is `NA` after each case's final stop. It is documented upstream as flat through an internal gap, but **randomForestRHF** 2.1.0 returns `NA` from a case's first gap onward, so on 2.1.0 the `hazard.only = FALSE` curve of such a case ends where the gap begins. Either curve breaks at a masked run instead of being joined across it, so a gap shows as a gap.

Each case gets its own colour and legend key only while there are at most 10 of them. Beyond that the colours stop being distinguishable and the legend would list every id, so the curves are drawn in a single translucent colour with no legend. To pick out individual cases, pass a short `idx`.

Value

A `ggplot` object.

See Also

`gg_rhf()`.

Examples

```
if (requireNamespace("randomForestRHF", quietly = TRUE)) {
  data(pbc, package = "randomForestSRC")
  d <- randomForestRHF::convert.counting(
    survival::Surv(days, status) ~ ., na.omit(pbc))
  o <- randomForestRHF::rhf("Surv(id, start, stop, event) ~ .", d, ntree = 30)
  plot(gg_rhf(o), idx = c(1, 5, 10))
}
```

---

plot.gg_rhf_importance

Plot Random Hazard Forest variable priority over time

Description

Draws a point matrix from a `gg_rhf_importance()` object. Each row is a variable, each column is a time window, and point size and color carry the time-localized RHF variable-priority score.

Usage

```
## S3 method for class 'gg_rhf_importance'
plot(
  x,
  vars = NULL,
  top_n_union = 15L,
  transform = c("none", "log10"),
  size_cap = 0.99,
  color_cap = 0.99,
  display_note = TRUE,
  labels = NULL,
  ...
)
```

Arguments

x	A gg_rhf_importance object from gg_rhf_importance().
vars	Optional nonempty character vector of variables to display. Unknown names are an error. When supplied, this takes precedence over top_n_union.
top_n_union	NULL or one positive integer. When vars is NULL, each time window contributes this many leading variables and the plot displays their union. NULL displays every variable.
transform	Display transformation: "none" (default) or "log10", which uses log10(priority + 1). The returned extractor object is never changed.
size_cap, color_cap	One numeric value in (0, 1]. Point size and color are capped at these quantiles of the finite display values. A value of 1 applies no cap.
display_note	Logical; if TRUE, an applied size or color cap is reported in the caption.
labels	Optional variable labels for the variable axis. One of: a named character vector (c(bili = "Serum bilirubin")); a labelled data frame, whose attr(col, "label") values are read; or a two-column key/label data frame. Variables with no label keep their raw name. Defaults to NULL (raw names).
...	Additional arguments passed to ggplot2::geom_point().

Details

Variables retain the global q90 ordering prepared by gg_rhf_importance(), with the highest-ranked variable at the top. Time windows remain chronological. This follows the variable-priority matrix in Ishwaran et al. (2026) while returning a ggplot object you can extend.

The transformation and caps affect display values only. The priority column in x remains on the upstream scale. A zero priority is drawn at the minimum point size; missing priorities are not drawn. If variable filtering leaves no finite values, the method stops rather than returning an empty plot.

Value

A ggplot object.

References

Ishwaran H, Hsich EM, Kogalur UB, Lee DKK (2026). Random Hazard Forests. arXiv:2608.21597. doi:10.48550/arXiv.2608.21597.

See Also

gg_rhf_importance(), randomForestRHF::dotmatrix.importance.rhf()

Examples

```
if (requireNamespace("randomForestRHF", quietly = TRUE)) {
  data(pbc, package = "randomForestSRC")
  d <- randomForestRHF::convert.counting(
    survival::Surv(days, status) ~ ., na.omit(pbc))
  o <- randomForestRHF::rhf(
    "Surv(id, start, stop, event) ~ .", d, ntree = 30)
  priority_fit <- randomForestRHF::importance.rhf(o)
  priority <- gg_rhf_importance(o, importance_fit = priority_fit)

  plot(priority, top_n_union = 10)
}
```

---

plot.gg_roc *ROC plot generic function for a gg_roc object.*

Description

ROC plot generic function for a gg_roc object.

Usage

```
## S3 method for class 'gg_roc'
plot(x, which_outcome = NULL, ..., panel = c("overlay", "facet"))
```

Arguments

x	A gg_roc object, or a raw rfsrc or randomForest classification forest. A raw forest is accepted, but plain plot(forest) does not arrive here. Both randomForestSRC and randomForest register their own plot methods, so S3 dispatch sends a raw forest to plot.rfsrc or plot.randomForest instead. This branch is reached only by naming the method outright, as plot.gg_roc(forest). That branch also does not use gg_roc's own default for which_outcome: given a multi-class forest and which_outcome = NULL it calls gg_roc once per class and overlays the one-vs-rest curves, where gg_roc(x) alone returns a single curve (a macro-average for randomForest, or class 1 with a warning for rfsrc). Prefer plot(gg_roc(x, which_outcome)), which is explicit about both the class and the engine. Issue #72 tracks reconciling the entry points.

which_outcome	Integer; for multi-class problems, the index of the class to plot. When NULL (default) and the forest has more than two classes, the curves for all classes are overlaid in one plot. For binary forests, NULL defaults to class index 2.
...	Additional arguments passed to gg_roc when x is a raw forest (e.g. oob = FALSE).
panel	Character; layout for per-class ROC objects, the ones from gg_roc(..., per_class = TRUE). "overlay" (default) draws every class curve in one panel, colored by class; "facet" gives each class its own panel. Ignored for single-class gg_roc objects.

Value

A ggplot object. The x-axis is 1 - Specificity (FPR), the y-axis is Sensitivity (TPR), and a dashed red diagonal marks the random-classifier baseline. Single-class curves carry the AUC as an annotation; multi-class plots color and style each class curve distinctly.

References

Breiman L. (2001). Random forests, Machine Learning, 45:5-32.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, Rnews, 7(2):25-31.

Ishwaran H, Kogalur U (2026). Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC). R package version 3.6.2. https://cran.r-project.org/package=randomForestSRC

See Also

gg_roc calc_roc calc_auc rfsrc randomForest

Examples

```
## ------------------------------------------------------------
## classification example
## ------------------------------------------------------------
## -------- iris data
# Build a small classification forest (ntree=50 keeps example fast)
set.seed(42)
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris, ntree = 50)

# ROC for setosa (outcome index 1)
gg_dta <- gg_roc(rfsrc_iris, which_outcome = 1)
plot(gg_dta)

# ROC for versicolor (outcome index 2)
gg_dta <- gg_roc(rfsrc_iris, which_outcome = 2)
plot(gg_dta)

# ROC for virginica (outcome index 3)
gg_dta <- gg_roc(rfsrc_iris, which_outcome = 3)
plot(gg_dta)

# Plot all three ROC curves in one call by iterating over outcome indices
```

```
n_cls <- ncol(rfsrc_iris$predicted)
for (i in seq_len(n_cls)) print(plot(gg_roc(rfsrc_iris, which_outcome = i)))
```

`plot.gg_sdependent` *Plot a* `gg_sdependent` *object*

Description

Ranked lollipop of the per-variable `sdependent()` signal score, sorted descending so the strongest signal lands at the top. Points are colored blue for variables flagged as signal (`signal == TRUE`) and gray otherwise.

Usage

```
## S3 method for class 'gg_sdependent'
plot(x, labels = NULL, ...)
```

Arguments

`x`	A `gg_sdependent` object from `gg_sdependent()`.
`labels`	Optional variable labels for the variable axis. One of: a named character vector (`c(bpd = "BP Diastole")`); a labelled data frame, whose `attr(col, "label")` values are read; or a two-column `key`/`label` data frame. Variables with no label keep their raw name. Defaults to `NULL` (raw names).
`...`	Not currently used.

Value

A `ggplot` object.

Reading the chart

Each lollipop is a variable's signal score from `varPro::sdependent()`, computed on the unsupervised entropy-region lasso structure of a `varPro::uvarpro()` fit. Blue variables cleared the detection threshold and are reported in `sdependent()$signal.vars`; gray ones did not. Pair with `gg_udependent()` (the dependency graph) to see *how* the signal variables connect, and with `gg_beta_uvarpro()` for the lasso-importance ranking.

See Also

`gg_sdependent()`, `gg_udependent()`, `gg_beta_uvarpro()`.

Examples

```
if (requireNamespace("varPro", quietly = TRUE)) {
  set.seed(1)
  o <- varPro::uvarpro(mtcars, ntree = 50)
  plot(gg_sdependent(o))
}
```

`plot.gg_shap` *Plot a gg_shap object*

Description

Routes to one of three SHAP views. `type = "beeswarm"` (default) draws the signature SHAP summary; `"importance"` draws a mean-absolute-SHAP bar chart; `"dependence"` draws SHAP value against a single feature's value.

Usage

```
## S3 method for class 'gg_shap'
plot(
  x,
  type = c("beeswarm", "importance", "dependence"),
  xvar = NULL,
  labels = NULL,
  ...
)
```

Arguments

`x`	A gg_shap object.
`type`	One of `"beeswarm"`, `"importance"`, or `"dependence"`.
`xvar`	For `type = "dependence"`, the variable to plot. When `NULL`, the top-ranked variable is used.
`labels`	Optional variable labels, forwarded to the mode function. One of: a named character vector (`c(Temp = "Temperature")`); a labelled data frame, whose `attr(col, "label")` values are read; or a two-column key/`label` data frame. Variables with no label keep their raw name. In `"dependence"` mode the label is substituted into the axis titles; `xvar` still matches on raw names.
`...`	Passed to the underlying builder.

Value

A `ggplot` object.

See Also

gg_shap shap_importance

Examples

```
if (requireNamespace("kernelshap", quietly = TRUE)) {
  rf <- randomForestSRC::rfsrc(Ozone ~ ., data = na.omit(airquality),
                               ntree = 50)
  gg_dta <- gg_shap(rf, bg_n = 20)
  plot(gg_dta, type = "importance")
}
```

---

plot.gg_survival *Plot a gg_survival object.*

Description

Draws a Kaplan-Meier (or Nelson-Aalen) survival curve from a gg_survival object. You can overlay a confidence envelope around the curve using error: "shade" fills the area between the pointwise confidence limits, "lines" draws them as dashed step functions, and "bars" shows them as error bars. When gg_survival was called with a by argument, each group gets its own step function and the label argument renames the legend.

Usage

```
## S3 method for class 'gg_survival'
plot(
  x,
  type = c("surv", "cum_haz", "hazard", "density", "mid_int", "life", "proplife"),
  error = c("shade", "bars", "lines", "none"),
  label = NULL,
  ...
)
```

Arguments

x	gg_survival or a survival gg_rfsrc object created from a rfsrc object
type	"surv", "cum_haz", "hazard", "density", "mid_int", "life", "proplife"
error	"shade", "bars", "lines" or "none"
label	Modify the legend label when gg_survival has stratified samples
...	Additional arguments forwarded to geom_step(). When alpha is supplied it is passed to the step geom and a halved value is used for the ribbon overlay.

Details

The type argument selects which quantity to plot on the y-axis. Survival probability ("surv") is the default, but cumulative hazard, density, and several transformed scales are available for the cases where a linear scale reveals more about the tails.

Value

A ggplot object. The y-axis shows the chosen type (e.g. survival probability for "surv") and the x-axis shows time. Confidence shading, bars, or lines are added when the input object carries confidence-interval columns.

See Also

gg_survival, kaplan, nelson, gg_rfsrc

Examples

```
## -------- pbc data
data(pbc, package = "randomForestSRC")
pbc$time <- pbc$days / 364.25

# This is the same as kaplan
gg_dta <- gg_survival(
  interval = "time", censor = "status",
  data = pbc
)

plot(gg_dta, error = "none")
plot(gg_dta)

# Stratified on treatment variable.
gg_dta <- gg_survival(
  interval = "time", censor = "status",
  data = pbc, by = "treatment"
)

plot(gg_dta, error = "none")
plot(gg_dta)
plot(gg_dta, label = "treatment")

# ...with smaller confidence limits.
gg_dta <- gg_survival(
  interval = "time", censor = "status",
  data = pbc, by = "treatment", conf.int = .68
)

plot(gg_dta, error = "lines")
plot(gg_dta, label = "treatment", error = "lines")

# ...with smaller confidence limits.
gg_dta <- gg_survival(
```

```
  interval = "time", censor = "status",
  data = pbc, by = "sex", conf.int = .68
)

plot(gg_dta, error = "lines")
plot(gg_dta, label = "sex", error = "lines")
```

`plot.gg_tune_rhf` *Plot a Random Hazard Forest tuning path*

Description

Draws the saved evaluated metric at each tree size and highlights the upstream selected size. OOB risk paths show the criterion minimized by upstream tuning; OOB iAUC paths show the criterion it maximizes. An iAUC path includes a standard-error band only when finite supplied standard errors are available.

Usage

```
## S3 method for class 'gg_tune_rhf'
plot(x, se_band = TRUE, se_mult = 1, ...)
```

Arguments

`x`	A `gg_tune_rhf` object from `gg_tune_rhf()`.
`se_band`	Logical; draw an iAUC standard-error band when available.
`se_mult`	Positive finite multiplier for the standard-error band.
`...`	Additional arguments passed to the evaluated-point layer.

Value

A `ggplot` object.

See Also

`gg_tune_rhf()`.

plot.gg_udependent *Plot a* gg_udependent *variable dependency graph*

Description

Draws the dependency graph held in a gg_udependent object as a ggraph network. Node color marks whether a variable made the signal set, and the width and opacity of an edge tell you how strong the dependency between its two variables is.

Usage

```
## S3 method for class 'gg_udependent'
plot(x, layout = "fr", labels = NULL, ...)
```

Arguments

x	A gg_udependent object from gg_udependent.
layout	Character; the igraph/ggraph layout algorithm. Common choices are "fr" (Fruchterman-Reingold, the default), "kk" (Kamada-Kawai), "stress", "circle", and "grid".
labels	Optional variable labels for the node text. One of: a named character vector (c(bpd = "BP Diastole")); a labelled data frame, whose attr(col, "label") values are read; or a two-column key/label data frame. Nodes with no label keep their raw name. Defaults to NULL (raw names).
...	Not currently used.

Details

This plot needs the **ggraph** package, which is in Suggests rather than installed for you. If it is missing, run install.packages("ggraph").

A signal variable (selected = TRUE) gets a blue node (#4e8fcd); the rest are gray (#888888). Node size grows with degree. Edge width and opacity both grow with the raw dependency weight I[i,j].

Value

A ggplot object (built via ggraph).

Reading the network

Each node is a variable; each edge is a cross-variable dependency that cleared the threshold passed to gg_udependent. The Fruchterman-Reingold layout (the default) places mutually connected variables near each other, so the picture tends to show hubs and the clusters around them rather than a tidy ring. The eye usually goes first to the largest blue node: a variable that is both in the signal set and connects to many others is a hub of the dependency structure. Edges with wider, more opaque strokes are stronger dependencies; thin, faint edges sit near the threshold and are the ones that would disappear first if you raised it.

Gray, low-degree nodes are the ones UVarPro thinks are not contributing much to the structure. (Truly isolated nodes are dropped by `gg_udependent()` before the graph is drawn; what you see is the connected component.) A cluster of mutually connected variables is worth checking for redundancy; they may be several views of the same underlying quantity.

What this tells you

Use the figure to pick a working set of variables: the hubs and their immediate neighbors are the candidates UVarPro flags as carrying structure. If a cluster of high-degree variables looks like it might be measuring the same thing, that is a cue to look at their pairwise correlations or fit them as a block rather than individually. The threshold and layout are recorded in the caption so a different choice is easy to spot in a later figure.

See Also

gg_udependent

Examples

```
if (requireNamespace("ggraph", quietly = TRUE)) {
  set.seed(42)
  uv <- varPro::uvarpro(iris[, -5], ntree = 50)
  plot(gg_udependent(uv))
}
```

---

plot.gg_variable *Plot a* gg_variable *object,*

Description

Plot a gg_variable object,

Usage

```
## S3 method for class 'gg_variable'
plot(
  x,
  xvar,
  panel = FALSE,
  oob = TRUE,
  points = TRUE,
  smooth = TRUE,
  labels = NULL,
  ...,
  time_units = NULL
)
```

Arguments

x	gg_variable object created from a rfsrc object
xvar	variable (or list of variables) of interest.
panel	Should plots be faceted along multiple xvar?
oob	oob estimates (boolean)
points	plot the raw data points (boolean)
smooth	include a smooth curve (boolean)
labels	Optional variable labels. One of: a named character vector (c(wt = "Weight")); a labelled data frame, whose attr(col, "label") values are read; or a two-column key/label data frame. Variables with no label keep their raw name. Applied to the facet strips in the panel plot and to the x axis title in the individual plot. Defaults to NULL (raw names).
...	arguments passed to the ggplot2 functions.
time_units	Optional name of the time unit the forest was fit in, used only in the survival y axis title. The horizon is chosen at gg_variable(rf, time = 1191); plot(gg_dta) then titles the axis "Survival at 1191", and time_units = "days" makes that "Survival at 1191 days". Nothing on an rfsrc object records the unit, so the package cannot infer it. Defaults to NULL (no unit printed). Not printed when multiple times are faceted.

Value

A single ggplot object when length(xvar) == 1 or panel = TRUE; otherwise a patchwork composite that stacks one panel per variable in xvar. Either way the result is one plottable object, never a bare list, so it composes with patchwork and dispatches through ggplot2::autoplot(). For the patchwork case, to inspect one panel with ggplot2::layer_data() pull that panel out first (e.g. ggplot2::layer_data(p[[1]])).

References

Breiman L. (2001). Random forests, Machine Learning, 45:5-32.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, Rnews, 7(2):25-31.

Ishwaran H, Kogalur U (2026). Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC). R package version 3.6.2. https://cran.r-project.org/package=randomForestSRC

See Also

gg_variable, gg_partial, plot.variable

Examples

```
## ------------------------------------------------------------
## classification
## ------------------------------------------------------------
## -------- iris data
set.seed(42)
```

```
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris, ntree = 50)

gg_dta <- gg_variable(rfsrc_iris)
plot(gg_dta, xvar = "Sepal.Width")
plot(gg_dta, xvar = "Sepal.Length")

## Panel plot across all predictors
plot(gg_dta,
  xvar = rfsrc_iris$xvar.names,
  panel = TRUE, se = FALSE
)

## ------------------------------------------------------------
## regression
## ------------------------------------------------------------
## -------- air quality data
# na.action = "na.impute" handles missing Ozone / Solar.R values
set.seed(42)
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ ., data = airquality,
                    na.action = "na.impute", ntree = 50)
gg_dta <- gg_variable(rfsrc_airq)

# Treat Month as an ordinal factor for better visualization
gg_dta[, "Month"] <- factor(gg_dta[, "Month"])

plot(gg_dta, xvar = "Wind")
plot(gg_dta, xvar = "Temp")
plot(gg_dta, xvar = "Solar.R")

# Factor variable uses notched boxplots
plot(gg_dta, xvar = "Month", notch = TRUE)


# Panel plot across continuous predictors (loess smooths; slower)
plot(gg_dta, xvar = c("Solar.R", "Wind", "Temp", "Day"), panel = TRUE)


## ------------------------------------------------------------
## survival examples
## ------------------------------------------------------------
## -------- veteran data

data(veteran, package = "randomForestSRC")
set.seed(42)
rfsrc_veteran <- randomForestSRC::rfsrc(Surv(time, status) ~ ., veteran,
  nsplit = 10,
  ntree = 50
)

# Marginal survival at 90 days
gg_dta <- gg_variable(rfsrc_veteran, time = 90)

# Single-variable dependence plots
```

```
plot(gg_dta, xvar = "age")
plot(gg_dta, xvar = "diagtime")

# Panel coplot for two predictors at a single time
plot(gg_dta, xvar = c("age", "diagtime"), panel = TRUE)

# Compare survival at 30, 90, and 365 days simultaneously
gg_dta <- gg_variable(rfsrc_veteran, time = c(30, 90, 365))

# Single-variable plot (one facet per time point)
plot(gg_dta, xvar = "age")

# Panel coplot across two predictors and three time points
plot(gg_dta, xvar = c("age", "diagtime"), panel = TRUE)
```

`plot.gg_varpro` *Plot a* `gg_varpro` *variable importance object*

Description

Draws a horizontal boxplot of the per-tree importance z-scores, or of the raw importances if you asked for those. Set `faithful = TRUE` at extract time and the per-tree points are scattered over the box; for a classification forest, `conditional = TRUE` splits the plot into one facet per class.

Usage

```
## S3 method for class 'gg_varpro'
plot(x, type, labels = NULL, ...)
```

Arguments

`x`	A `gg_varpro` object from `gg_varpro`.
`type`	Character; the display scale. Leave it off and it is read from `provenance$local.std`: `"z"` when `local.std = TRUE` (the default), `"raw"` when `local.std = FALSE`. Asking for a scale that the extract step did not prepare raises an error.
`labels`	Optional variable labels for the variable axis. One of: a named character vector (`c(bpd_last = "BP Diastole")`); a labelled data frame, whose `attr(col, "label")` values are read; or a two-column `key/label` data frame. Variables with no label keep their raw name. Defaults to `NULL` (raw names).
`...`	Not currently used.

Details

Boxplot geometry: the hinges are the 15th and 85th percentiles of the per-tree z-distribution, and the whiskers run to the 5th and 95th. This is **not** a Tukey boxplot, and the plot carries a caption that says so.

`faithful = TRUE`**:** the per-tree values are jittered over the box as semi-transparent points, on the same scale as the box itself (z when `local.std = TRUE`, raw when `local.std = FALSE`). The box is drawn faint to let the points show through, and a white-outlined dot marks the mean.

`conditional = TRUE`**:** the class-conditional importances (`$conditional`) are shown as a faceted bar chart (`facet_wrap(~class, nrow = 1)`). Variables keep the sort order set by the unconditional median z in `$stats`, so the facets line up. A variable whose class-conditional z is undefined (0/0, possible only with `local.std = FALSE`) gets no bar in that facet, and the caption says so; a zero bar would read as a measured zero. 0/0 arises when too few trees tested the variable, but also when every tree scored it exactly 0, so the plot does not say which. Every class keeps its facet, even one with no defined z.

Value

A `ggplot` object.

Reading the boxplot

Variables are sorted top to bottom by descending median per-tree importance, so the eye lands on the most important variable first. For each variable the box spans the 15th to 85th percentile of the per-tree scores, the center line is the median, and the whiskers run out to the 5th and 95th percentile, not the usual Tukey 1.5 IQR whiskers. The dashed vertical line is the selection `cutoff` (default `0.79`), or, when `gg_varpro()` was given `noise`, the noise bar: the largest z among tested noise of the variable's kind, or of the other kind when its own has none (the caption then says "from continuous noise", say). With both a categorical and a continuous bar each row gets its own dashed tick, and noise variables are drawn in orange. The conditional plot always draws `cutoff`. On the default z-score axis (`local.std = TRUE`) that line is a z; on the raw-importance axis (`local.std = FALSE`, `type = "raw"`) it is the same numeric value but in raw-importance units. Boxes whose aggregate value sits above the line are colored blue and flagged `selected = TRUE`, the rest are gray. A variable with a usable release rule in fewer than `trim` of the trees (`tested = FALSE`) is drawn as an open box, usually a flat line at zero: it was barely tested, which is not the same as tested and found unimportant. A selected variable with a tight, high box is a variable the forest agrees on across trees. A selected variable with a wide box that straddles the cutoff is one to look at twice before relying on it.

With `faithful = TRUE` the box is drawn faint and the per-tree values are jittered over it as semi-transparent points, on the same scale as the box (z when `local.std = TRUE`, raw otherwise). A white-outlined dot marks the mean. Use this view when you want to see how individual trees voted rather than just the summary.

For a classification forest with `conditional = TRUE` the plot splits into one facet per class. Variables keep the unconditional sort order, so the rows line up across facets and you can read across to see which class a variable is informative for.

What this tells you

Take the variables above the cutoff as your candidate set. Use the width of the box and the per-tree overlay to gauge confidence: a narrow box well above the cutoff is a confident pick, a wide box that crosses it is a coin flip you should not lean on. For classification, conditional importance tells you which variables drive which class; a variable that is unconditionally important but only important for one class out of several is still useful, just useful for a narrower question.

See Also

gg_varpro

Examples

```
set.seed(42)
vp <- varPro::varpro(mpg ~ ., data = mtcars, ntree = 50)
plot(gg_varpro(vp))
plot(gg_varpro(vp, faithful = TRUE))
```

---

plot.gg_vimp — *Plot a gg_vimp object, extracted variable importance of a rfsrc object*

Description

Draws a horizontal bar chart of the VIMP scores extracted by gg_vimp. Each bar represents one predictor; bar length is proportional to its permutation VIMP, the average rise in OOB prediction error when that predictor's OOB values are randomly shuffled. Predictors are sorted in descending order of importance so the most influential variables appear at the top.

Usage

```
## S3 method for class 'gg_vimp'
plot(x, relative = FALSE, lbls, labels = NULL, ...)
```

Arguments

x	gg_vimp object created from a rfsrc object
relative	If TRUE, plot relative VIMP: each variable's VIMP divided by the largest VIMP in its set, so the top variable reads 1 (for classification, the top variable within each class). A set with no positive VIMP is divided by its largest absolute VIMP instead, and an all-zero set stays at zero. Defaults to FALSE, raw VIMP.
lbls	*Deprecated* as of v4.0.0; use labels. A named character vector of alternative variable labels.

labels	Optional variable labels for the variable axis. One of: a named character vector (c(bpd_last = "BP Diastole")); a labelled data frame, whose attr(col, "label") values are read; or a two-column key/label data frame. Variables with no label keep their raw name. Defaults to NULL (raw names).
...	optional arguments passed to gg_vimp if necessary

Details

Bars are colored by the positive flag: a bar at or below zero (non-positive VIMP) is color-coded differently to flag predictors that *hurt* OOB accuracy when their signal is removed, usually a sign of collinearity or a very noisy variable. In a well-behaved forest most bars are positive; the color distinction matters when a handful are not.

Value

ggplot object

References

Breiman L. (2001). Random forests, Machine Learning, 45:5-32.

Ishwaran H. and Kogalur U.B. (2007). Random survival forests for R, Rnews, 7(2):25-31.

Ishwaran H, Kogalur U (2026). Fast Unified Random Forests for Survival, Regression, and Classification (RF-SRC). R package version 3.6.2. https://cran.r-project.org/package=randomForestSRC

See Also

gg_vimp

Examples

```
## ------------------------------------------------------------
## classification example
## ------------------------------------------------------------
## -------- iris data
rfsrc_iris <- randomForestSRC::rfsrc(Species ~ ., data = iris, ntree = 50)
gg_dta <- gg_vimp(rfsrc_iris)
plot(gg_dta)

## ------------------------------------------------------------
## regression example
## ------------------------------------------------------------
## -------- air quality data
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ ., airquality, ntree = 50)
gg_dta <- gg_vimp(rfsrc_airq)
plot(gg_dta)
```

print.gg_beta_uvarpro *Print methods for gg_* data objects*

Description

Each `print.gg_*()` method prints a one-line header: the class label and, where the forest recorded it, provenance (source package, family, ntree, n). It returns the object invisibly, so `print()` sits cleanly in a pipe.

Usage

```
## S3 method for class 'gg_beta_uvarpro'
print(x, ...)

## S3 method for class 'gg_sdependent'
print(x, ...)

## S3 method for class 'gg_error'
print(x, ...)

## S3 method for class 'gg_vimp'
print(x, ...)

## S3 method for class 'gg_rfsrc'
print(x, ...)

## S3 method for class 'gg_variable'
print(x, ...)

## S3 method for class 'gg_partial'
print(x, ...)

## S3 method for class 'gg_partial_rfsrc'
print(x, ...)

## S3 method for class 'gg_partialpro'
print(x, ...)

## S3 method for class 'gg_partial_varpro'
print(x, ...)

## S3 method for class 'gg_roc'
print(x, ...)

## S3 method for class 'gg_survival'
print(x, ...)
```

```
## S3 method for class 'gg_brier'
print(x, ...)

## S3 method for class 'gg_rhf'
print(x, ...)

## S3 method for class 'gg_tune_rhf'
print(x, ...)

## S3 method for class 'gg_auct'
print(x, ...)

## S3 method for class 'gg_rhf_importance'
print(x, ...)

## S3 method for class 'gg_udependent'
print(x, ...)

## S3 method for class 'summary.gg_udependent'
print(x, ...)

## S3 method for class 'gg_varpro'
print(x, ...)

## S3 method for class 'gg_isopro'
print(x, ...)

## S3 method for class 'gg_beta_varpro'
print(x, ...)

## S3 method for class 'gg_ivarpro'
print(x, ...)

## S3 method for class 'gg_shap'
print(x, ...)

## S3 method for class 'gg_ale_rfsrc'
print(x, ...)

## S3 method for class 'gg_ale_interaction'
print(x, ...)
```

Arguments

x	A gg_* data object.
...	Not currently used.

Details

To see the rows themselves, use `head()`; for per-class diagnostics, use `summary.gg`.

Value

The object `x`, invisibly.

See Also

`summary.gg`, `autoplot.gg`

Examples

```
set.seed(42)
airq <- na.omit(airquality)
rf <- randomForestSRC::rfsrc(Ozone ~ ., data = airq, ntree = 50)
print(gg_error(rf))
print(gg_vimp(rf))
```

---

quantile_pts *Quantile-based cut points for coplots*

Description

This helper wraps `quantile` to create well-spaced cut points for conditioning plots. When `intervals = TRUE` the lower boundary is nudged down so that `cut()` treats the minimum value as a valid observation.

The output can be passed directly into the breaks argument of the `cut` function for creating groups for coplots.

Usage

```
quantile_pts(object, groups, intervals = FALSE)
```

Arguments

`object`	Numeric vector of predictor values.
`groups`	Number of quantile points (or intervals) to compute.
`intervals`	Logical indicating whether to return interval boundaries suitable for `cut()` (length `groups + 1`) or the interior quantile points (length `groups`).

Value

Numeric vector of quantile points. When `intervals = TRUE` the result is strictly increasing and can be supplied to `cut()` to produce `groups` balanced strata.

See Also

cut

Examples

```
if (requireNamespace("MASS", quietly = TRUE)) {
  data(Boston, package = "MASS")
  rfsrc_boston <- randomForestSRC::rfsrc(medv ~ ., Boston, ntree = 100)

  # To create 6 intervals, we want 7 points.
  # quantile_pts will find balanced intervals
  rm_pts <- quantile_pts(rfsrc_boston$xvar$rm, groups = 6, intervals = TRUE)

  # Use cut to create the intervals
  rm_grp <- cut(rfsrc_boston$xvar$rm, breaks = rm_pts)

  summary(rm_grp)
}
```

---

shap_beeswarm *SHAP beeswarm summary plot*

Description

The signature SHAP summary: one jittered point per (observation, variable), positioned by SHAP value and colored by the feature value, min-max scaled to `[0, 1]` within each variable so every variable's own range maps to the full color gradient. Categorical features have no numeric value and render as a neutral gray (no numeric value to scale).

Usage

```
shap_beeswarm(x, labels = NULL, ...)
```

Arguments

x	A gg_shap object.
labels	Optional variable labels. One of: a named character vector (`c(Temp = "Temperature")`); a labelled data frame, whose `attr(col, "label")` values are read; or a two-column key/label data frame. Variables with no label keep their raw name. Defaults to `NULL` (raw names).
...	Unused.

Value

A ggplot object.

See Also

gg_shap plot.gg_shap

---

shap_dependence *SHAP dependence plot*

Description

SHAP value against the value of a single feature, the SHAP analog of a partial-dependence plot. Numeric features use a continuous x-axis; factor or character features fall back to their labels on a discrete axis.

Usage

```
shap_dependence(x, xvar = NULL, labels = NULL, ...)
```

Arguments

x	A gg_shap object.
xvar	The variable to plot. When NULL, the top-ranked variable (largest mean absolute SHAP) is used.
labels	Optional variable labels. One of: a named character vector (c(Temp = "Temperature")); a labelled data frame, whose attr(col, "label") values are read; or a two-column key/label data frame. Variables with no label keep their raw name. Defaults to NULL (raw names).
...	Unused.

Value

A ggplot object.

See Also

gg_shap plot.gg_shap

shap_importance *SHAP global importance bar chart*

Description

Bar chart of mean absolute SHAP value per variable, the SHAP analog of plot.gg_vimp.

Usage

```
shap_importance(x, labels = NULL, ...)
```

Arguments

x	A gg_shap object.
labels	Optional variable labels. One of: a named character vector (c(Temp = "Temperature")); a labelled data frame, whose attr(col, "label") values are read; or a two-column key/label data frame. Variables with no label keep their raw name. Defaults to NULL (raw names).
...	Unused.

Value

A ggplot object.

See Also

gg_shap plot.gg_shap

summary.gg_beta_uvarpro *Summary methods for gg_* data objects*

Description

Where print gives you a one-line header, summary digs a level deeper. Each summary.gg_*() method returns a summary.gg object: a header line plus a few diagnostic statistics for that object type (the OOB error curve, the top VIMP variables, a time range, the integrated CRPS, and so on). print.summary.gg() renders it to the console.

Usage

```
## S3 method for class 'gg_beta_uvarpro'
summary(object, ...)

## S3 method for class 'gg_sdependent'
summary(object, ...)

## S3 method for class 'summary.gg'
print(x, ...)

## S3 method for class 'gg_error'
summary(object, ...)

## S3 method for class 'gg_vimp'
summary(object, ...)

## S3 method for class 'gg_rfsrc'
summary(object, ...)

## S3 method for class 'gg_variable'
summary(object, ...)

## S3 method for class 'gg_partial'
summary(object, ...)

## S3 method for class 'gg_partial_rfsrc'
summary(object, ...)

## S3 method for class 'gg_partialpro'
summary(object, ...)

## S3 method for class 'gg_partial_varpro'
summary(object, ...)

## S3 method for class 'gg_roc'
summary(object, ...)

## S3 method for class 'gg_survival'
summary(object, ...)

## S3 method for class 'gg_varpro'
summary(object, ...)

## S3 method for class 'gg_udependent'
summary(object, ...)

## S3 method for class 'gg_brier'
summary(object, ...)
```

```
## S3 method for class 'gg_rhf'
summary(object, ...)

## S3 method for class 'gg_tune_rhf'
summary(object, ...)

## S3 method for class 'gg_rhf_importance'
summary(object, ...)

## S3 method for class 'gg_isopro'
summary(object, ...)

## S3 method for class 'gg_beta_varpro'
summary(object, ...)

## S3 method for class 'gg_ivarpro'
summary(object, ...)

## S3 method for class 'gg_auct'
summary(object, ...)

## S3 method for class 'gg_shap'
summary(object, ...)

## S3 method for class 'gg_ale_rfsrc'
summary(object, ...)

## S3 method for class 'gg_ale_interaction'
summary(object, ...)
```

Arguments

object	A gg_* data object.
...	Not currently used.
x	A summary.gg object (for print.summary.gg).

Value

A summary.gg object: a list with header and body character vectors. print.summary.gg returns it invisibly.

See Also

print.gg, autoplot.gg

Examples

```
set.seed(42)
```

```
airq <- na.omit(airquality)
rf <- randomForestSRC::rfsrc(Ozone ~ ., data = airq, ntree = 50)
summary(gg_error(rf))
summary(gg_vimp(rf))
```

`surv_partial.rfsrc` *Survival partial dependence data for one or more predictors*

Description

Deprecated. Use `gg_partial_rfsrc` instead, which returns a classed `gg_partial_rfsrc` object with a dedicated `plot()` method.

Usage

```
surv_partial.rfsrc(rforest, var_list, npts = 25, partial.type = "surv")
```

Arguments

`rforest`	A fitted `rfsrc` survival or competing-risk forest object.
`var_list`	Character vector of predictor names for which partial dependence should be computed. Each must appear in `rforest$xvar.names`.
`npts`	Integer; the number of predictor grid points to evaluate (default 25). Evenly-spaced unique values are sampled from each predictor.
`partial.type`	The prediction type to return. For survival forests one of `"surv"` (default), `"mort"`, or `"chf"`. For competing risk forests one of `"years.lost"`, `"cif"`, or `"chf"`. See `partial.rfsrc` for full details.

Details

Computes partial dependence curves for a survival or competing-risk `rfsrc` forest. For each predictor it calls `partial.rfsrc` at `npts` evenly-spaced unique values, across every stored event time.

Value

A named list with one element per variable in `var_list`. Each element is itself a list with:

name The predictor variable name (character).

dta The raw output of `get.partial.plot.data`, a list containing at minimum `x` (predictor values) and `yhat` (partial predictions), and for survival/competing risk, `partial.time`.

See Also

`gg_partial_rfsrc`, `partial.rfsrc`, `get.partial.plot.data`

Examples

```
## ------------------------------------------------------------
## survival
## ------------------------------------------------------------

data(veteran, package = "randomForestSRC")
v.obj <- randomForestSRC::rfsrc(Surv(time,status)~.,
  veteran, nsplit = 10, ntree = 100)

spart <- surv_partial.rfsrc(v.obj, var_list="age", partial.type = "mort")

## partial effect of age on mortality
partial.obj <- randomForestSRC::partial(v.obj,
                      partial.type = "mort",
                      partial.xvar = "age",
                      partial.values = v.obj$xvar$age,
                      partial.time = v.obj$time.interest)
pdta <- randomForestSRC::get.partial.plot.data(partial.obj)

plot(lowess(pdta$x, pdta$yhat, f = 1/3),
     type = "l", xlab = "age", ylab = "adjusted mortality")

## example where x is discrete - partial effect of age on mortality
## we use the granule=TRUE option
partial.obj <- randomForestSRC::partial(v.obj,
                      partial.type = "mort",
                      partial.xvar = "trt",
                      partial.values = v.obj$xvar$trt,
                      partial.time = v.obj$time.interest)
pdta <- randomForestSRC::get.partial.plot.data(partial.obj, granule = TRUE)
boxplot(pdta$yhat ~ pdta$x, xlab = "treatment", ylab = "partial effect")


## partial effects of karnofsky score on survival
karno <- quantile(v.obj$xvar$karno)
partial.obj <- randomForestSRC::partial(v.obj,
                      partial.type = "surv",
                      partial.xvar = "karno",
                      partial.values = karno,
                      partial.time = v.obj$time.interest)
pdta <- randomForestSRC::get.partial.plot.data(partial.obj)

matplot(pdta$partial.time, t(pdta$yhat), type = "l", lty = 1,
        xlab = "time", ylab = "karnofsky adjusted survival")
legend("topright", legend = paste0("karnofsky = ", karno), fill = 1:5)


## ------------------------------------------------------------
## competing risk
## ------------------------------------------------------------

data(follic, package = "randomForestSRC")
```

```
follic.obj <- randomForestSRC::rfsrc(Surv(time, status) ~ ., follic, nsplit = 3, ntree = 100)

## partial effect of age on years lost
partial.obj <- randomForestSRC::partial(follic.obj,
                      partial.type = "years.lost",
                      partial.xvar = "age",
                      partial.values = follic.obj$xvar$age,
                      partial.time = follic.obj$time.interest)
pdta1 <- randomForestSRC::get.partial.plot.data(partial.obj, target = 1)
pdta2 <- randomForestSRC::get.partial.plot.data(partial.obj, target = 2)

# Save and restore the user's graphical parameters per CRAN policy.
oldpar <- par(no.readonly = TRUE)
on.exit(par(oldpar))
par(mfrow = c(2, 2))
plot(lowess(pdta1$x, pdta1$yhat),
     type = "l", xlab = "age", ylab = "adjusted years lost relapse")
plot(lowess(pdta2$x, pdta2$yhat),
     type = "l", xlab = "age", ylab = "adjusted years lost death")

## partial effect of age on cif
partial.obj <- randomForestSRC::partial(follic.obj,
                      partial.type = "cif",
                      partial.xvar = "age",
                      partial.values = quantile(follic.obj$xvar$age),
                      partial.time = follic.obj$time.interest)
pdta1 <- randomForestSRC::get.partial.plot.data(partial.obj, target = 1)
pdta2 <- randomForestSRC::get.partial.plot.data(partial.obj, target = 2)

matplot(pdta1$partial.time, t(pdta1$yhat), type = "l", lty = 1,
        xlab = "time", ylab = "age adjusted cif for relapse")
matplot(pdta2$partial.time, t(pdta2$yhat), type = "l", lty = 1,
        xlab = "time", ylab = "age adjusted cif for death")
```

`varpro_coverage` *How well a varpro fit tested each variable*

Description

Reports, for every term varpro could test, how many of the forest's trees scored it with a release rule, next to its importance z. Use it to tell a variable that was tested and found unimportant from one that was barely tested at all.

Usage

```
varpro_coverage(
  object,
  by = c("term", "variable"),
  trim = 0.1,
```

```
  max.tree = object$rf$ntree,
  max.rules.tree = object$max.rules.tree,
  noise = NULL,
  kind_values = 5L,
  seed = NULL,
  max_untested = NULL
)
```

Arguments

object	A fitted varpro object.
by	"term" (one row per term in the fit's data, factor levels separately) or "variable" (one-hot levels rolled up to their original variable: tested in a tree when any level is, z the largest level z).
trim	Coverage below which a term counts as untested, and the winsorisation trim used for z. Defaults to 0.1, as in varPro::importance(): the share of trees below which z is 0.
max.tree	Number of trees to score. Defaults to every tree, rather than varPro's sample of 150, so that coverage rests on all the evidence. Coverage and z always come from the same extracted rules.
max.rules.tree	Maximum release rules sampled per tree; defaults to the value stored in object. Sampling only happens in trees with more branches than this.
noise	NULL, or a regular expression matching the names of noise variables (for example "^noise").
kind_values	A numeric variable with at most this many distinct values counts as categorical for the noise comparison.
seed	Seed passed to varPro::varpro.strength(), which samples the release rules; with a seed (or a set.seed() call first) the table is reproducible.
max_untested	NULL (the default: report only), or the largest share of real variables, from 0 to 1, that may be untested. Rows are counted as by lists them and noise rows are left out. A row counts as tested only when some term of it (for a factor, some single level) was scored in at least trim of the trees and in at least one, so ineligible rows always count as untested. When the share is larger the call stops with an error, so that a screen that tested too little cannot be read as one that found no effect.

Value

A data frame of class c("varpro_coverage", "data.frame"), one row per term (or variable), most-covered first, with columns term (term rows only), variable, n_terms (variable rows only), eligible (varpro built rules for it; FALSE when the split weights or nvar cut it first), n_trees (trees with a rule that scored it), coverage (n_trees over the trees z is averaged over), n_rules (rules that scored it), z (computed as varPro::importance() computes it, with its default local.std = TRUE, from the same rules; NA when never returned) and tested (coverage >= trim). With noise set it adds noise (logical), kind, noise_cut, noise_ref (the kind of the noise that set noise_cut: the row's own kind, the other kind when its own has no tested noise, or NA when no noise was tested) and selected (a tested real row, scored in at least one tree, whose z beats noise_cut). Attributes by, trees (the coverage denominator), trim and max.rules.tree record the settings.

Why coverage matters

`varPro::importance()` scores a term by a trimmed (winsorised) mean of its importance in each tree, and a tree without a usable release rule for the term counts as zero. A term tested in fewer than about `trim` of the trees therefore has every non-zero value trimmed away and comes back with z exactly 0: it is *untested*, not unimportant. A term varpro does not return at all was never tested, either because it was cut before rule-building (split weights and `nvar`) or because no rule had cases on both sides of its release. Neither is evidence of no effect.

A rule is usable when its out-of-bag region and its released complement both hold cases; it scores the term when its local importance also comes back as a value. Coverage is the share of scored trees with at least one rule that scored the term, out of the trees z is averaged over (those holding any usable rule). It counts exactly what `gg_varpro` counts, so on the same rules the two agree on `coverage` and `tested`.

Noise reference

varpro's default cutoff ($z > 0.79$) does not carry from one fit to another with the default `local.std = TRUE`: z is then an unstandardised mean, and it shrinks as more terms share each tree. If the data include permuted copies of real columns (noise by construction), name them with a common prefix and pass a pattern as `noise`. Each real row is then compared with the largest z among *tested* noise rows of the same kind (categorical or continuous); an untested noise row has $z = 0$ by construction and sets no bar. Here tested also means scored in at least one tree, so `trim = 0` does not make unscored noise a bar at 0. A kind with no tested noise falls back to the largest tested-noise z overall; `noise_ref` records it and a warning says so. ggRandomForests does not add the noise columns; do that before fitting.

The first-pass filter

Before it builds any rules, `varPro::varpro()` screens the terms with split weights. With its default `split.weight = TRUE` it weights every term (by lasso, variable importance or tree usage), drops any term whose weight is 0, and keeps only the `nvar` heaviest (30 by default). Every other term is ineligible: no rule is ever built for it, so it is untested however strong its effect. On wide data most terms can be cut here. Fit with `split.weight = FALSE` to make every term eligible; `nvar` then has no effect. Custom weights passed as `split.weight.custom` take precedence over that switch: terms they weight 0 or leave out, and terms beyond `nvar`, are still cut, so drop the custom weights too. varPro's hidden `split.weight.tolerance` only floors the weights of the terms that survive and does not change which terms are cut.

See Also

`gg_varpro`, `varpro_feature_names`

Examples

```
if (requireNamespace("varPro", quietly = TRUE)) {
  set.seed(20261014)
  d <- mtcars
  d$noise_wt <- sample(d$wt)        # permuted copy: noise by construction
  vp <- varPro::varpro(mpg ~ ., data = d, ntree = 50)
  set.seed(20261014)
```

```
  varpro_coverage(vp, noise = "^noise")
}
```

`varpro_feature_names` *Recover original variable names from varpro one-hot encoded feature names*

Description

`varpro` one-hot encodes factor variables, appending a numeric suffix for each level: `sex` becomes `sex0` and `sex1`. To map those back, this function strips the suffix one character at a time until every name in `varpro_names` matches a column in `dataset`.

Usage

```
varpro_feature_names(varpro_names, dataset)
```

Arguments

`varpro_names`	character vector of names as output by varpro (may include one-hot encoded suffixed names such as `"sex0"`, `"sex1"`)
`dataset`	the original data frame passed to varpro, used to look up valid column names

Value

character vector of unique original variable names (no suffixes)

See Also

`gg_partialpro`

Examples

```
## ------------------------------------------------------------------
## Simple case: one continuous variable + one binary factor
## ------------------------------------------------------------------
ds <- data.frame(age = c(25, 30, 45), sex = c("M", "F", "M"))

# varpro one-hot encodes 'sex' into 'sex0' and 'sex1'
varpro_names <- c("age", "sex0", "sex1")
varpro_feature_names(varpro_names, ds)
# Returns: c("age", "sex")

## ------------------------------------------------------------------
## Multi-level factor: three-level 'group' variable
## ------------------------------------------------------------------
ds2 <- data.frame(score = 1:6,
```

```
                  group = factor(rep(c("A", "B", "C"), 2)))

# varpro appends 0/1/2 for each level
vn2 <- c("score", "group0", "group1", "group2")
varpro_feature_names(vn2, ds2)
# Returns: c("score", "group")

## ------------------------------------------------------------------
## Already-clean names pass through unchanged
## ------------------------------------------------------------------
ds3 <- data.frame(x = 1:5, y = 1:5)
varpro_feature_names(c("x", "y"), ds3)
# Returns: c("x", "y")
```

Index

autoplot.gg, 6, *118*, *123*
autoplot.gg_ale_interaction *(*autoplot.gg*)*, 6
autoplot.gg_ale_rfsrc *(*autoplot.gg*)*, 6
autoplot.gg_auct *(*autoplot.gg*)*, 6
autoplot.gg_brier *(*autoplot.gg*)*, 6
autoplot.gg_error *(*autoplot.gg*)*, 6
autoplot.gg_isopro *(*autoplot.gg*)*, 6
autoplot.gg_partial *(*autoplot.gg*)*, 6
autoplot.gg_partial_rfsrc *(*autoplot.gg*)*, 6
autoplot.gg_partial_varpro *(*autoplot.gg*)*, 6
autoplot.gg_partialpro *(*autoplot.gg*)*, 6
autoplot.gg_rfsrc *(*autoplot.gg*)*, 6
autoplot.gg_rhf *(*autoplot.gg*)*, 6
autoplot.gg_rhf_importance *(*autoplot.gg*)*, 6
autoplot.gg_roc *(*autoplot.gg*)*, 6
autoplot.gg_shap *(*autoplot.gg*)*, 6
autoplot.gg_survival *(*autoplot.gg*)*, 6
autoplot.gg_tune_rhf *(*autoplot.gg*)*, 6
autoplot.gg_udependent *(*autoplot.gg*)*, 6
autoplot.gg_variable *(*autoplot.gg*)*, 6
autoplot.gg_varpro *(*autoplot.gg*)*, 6
autoplot.gg_vimp *(*autoplot.gg*)*, 6

calc_auc, *4*, 8, *10*, *51*, *102*
calc_roc, *4*, *9*, *51*, *102*
calc_roc *(*calc_roc.rfsrc*)*, 9
calc_roc.rfsrc, 9

factor, *35*

get.beta.entropy, *58*
get.brier.survival, *20, 21*, *80*
get.partial.plot.data, *35*, *124*
gg_ale_rfsrc, 11, *74, 75*
gg_auct, *5*, 13
gg_auct(), *76*
gg_beta_uvarpro, *4*, 15
gg_beta_uvarpro(), *52, 53*, *77*, *103*
gg_beta_varpro, *4*, 16
gg_beta_varpro(), *15, 16*, *30, 31*, *77–79*
gg_brier, *4*, 19, *79, 80*
gg_error, *4*, *21*, 21, *80, 81*
gg_isopro, *4*, 25, *27*, *84, 85*
gg_ivarpro, *4*, 28
gg_ivarpro(), *17*, *86*
gg_partial, *4*, 31, *34, 35*, *87*, *110*
gg_partial_rfsrc, *4*, *11, 12*, *32*, 33, *37*, *42*, *88, 89*, *124*
gg_partial_varpro, *4*, *36*, 36, *40–42*, *90*, *94*
gg_partialpro, *4*, *32*, *42*, *129*
gg_partialpro *(*gg_partial_varpro*)*, 36
gg_rfsrc, *4*, *56*, *95, 96*, *105, 106*
gg_rfsrc *(*gg_rfsrc.rfsrc*)*, 44
gg_rfsrc.rfsrc, 44
gg_rhf, *5*, 46
gg_rhf(), *98, 99*
gg_rhf_importance, *5*, 48
gg_rhf_importance(), *99–101*
gg_roc, *4*, *8–10*, *101, 102*
gg_roc *(*gg_roc.rfsrc*)*, 50
gg_roc.rfsrc, 50
gg_sdependent, *4*, 52
gg_sdependent(), *103*
gg_shap, *4*, *11*, 53, *104, 105*, *119–121*
gg_survival, *4*, *45*, 54, *72, 73*, *105, 106*
gg_tune_rhf, *5*, *57*, 57
gg_tune_rhf(), *107*
gg_udependent, *4*, 58, *108, 109*
gg_udependent(), *16*, *52, 53*, *77*, *103*
gg_variable, *4*, *22*, *32*, 61, *109, 110*
gg_varpro, *4*, *42*, 63, *68, 69*, *112*, *114*, *128*
gg_varpro(), *17–19*, *29–31*, *78*
gg_vimp, *4*, *22*, *42*, *67*, 67, *114, 115*
gg_vimp(), *17*, *19*, *29*, *31*
ggplot2::geom_point(), *100*

ggRandomForests-package, 3

isopro, *25, 26*, *28*, *85*

kaplan, *55, 56*, 71, *73*, *106*
kernelshap, *4*, *53, 54*

nelson, *55, 56*, *72*, 72, *73*, *106*

partial.rfsrc, *34, 35*, *124*
partialpro, *37*, *92*
plot(), *58*
plot.gg_ale_interaction, 74
plot.gg_ale_rfsrc, *12*, *74*, 75
plot.gg_auct, 76
plot.gg_auct(), *14*
plot.gg_beta_uvarpro, 77
plot.gg_beta_varpro, 78
plot.gg_beta_varpro(), *19*
plot.gg_brier, *21*, 79
plot.gg_error, *22*, 80
plot.gg_isopro, *27, 28*, 84
plot.gg_ivarpro, 85
plot.gg_partial, *32*, 87, *88, 89*
plot.gg_partial_rfsrc, *35*, *75*, 88, *90*
plot.gg_partial_varpro, *39*, *42*, 90
plot.gg_partialpro
(plot.gg_partial_varpro), 90
plot.gg_rfsrc, *44, 45*, 95
plot.gg_rhf, 98
plot.gg_rhf(), *47*
plot.gg_rhf_importance, 99
plot.gg_rhf_importance(), *49*
plot.gg_roc, *9, 10*, *51*, 101
plot.gg_sdependent, 103
plot.gg_shap, 104, *120, 121*
plot.gg_survival, *56*, *72, 73*, 105
plot.gg_tune_rhf, 107
plot.gg_udependent, *60*, 108
plot.gg_variable, *61*, *87*, 109
plot.gg_varpro, *67*, 112
plot.gg_vimp, *68, 69*, 114, *121*
plot.rfsrc, *22*, *80, 81*, *101*
plot.variable, *61*, *110*
predict.isopro, *25*, *27*
predict.rfsrc, *9*
print.gg, *123*
print.gg (print.gg_beta_uvarpro), 116
print.gg_ale_interaction
(print.gg_beta_uvarpro), 116
print.gg_ale_rfsrc
(print.gg_beta_uvarpro), 116
print.gg_auct (print.gg_beta_uvarpro),
116
print.gg_beta_uvarpro, 116
print.gg_beta_varpro
(print.gg_beta_uvarpro), 116
print.gg_brier (print.gg_beta_uvarpro),
116
print.gg_error (print.gg_beta_uvarpro),
116
print.gg_isopro
(print.gg_beta_uvarpro), 116
print.gg_ivarpro
(print.gg_beta_uvarpro), 116
print.gg_partial
(print.gg_beta_uvarpro), 116
print.gg_partial_rfsrc
(print.gg_beta_uvarpro), 116
print.gg_partial_varpro
(print.gg_beta_uvarpro), 116
print.gg_partialpro
(print.gg_beta_uvarpro), 116
print.gg_rfsrc (print.gg_beta_uvarpro),
116
print.gg_rhf (print.gg_beta_uvarpro),
116
print.gg_rhf_importance
(print.gg_beta_uvarpro), 116
print.gg_roc (print.gg_beta_uvarpro),
116
print.gg_sdependent
(print.gg_beta_uvarpro), 116
print.gg_shap (print.gg_beta_uvarpro),
116
print.gg_survival
(print.gg_beta_uvarpro), 116
print.gg_tune_rhf
(print.gg_beta_uvarpro), 116
print.gg_udependent
(print.gg_beta_uvarpro), 116
print.gg_variable
(print.gg_beta_uvarpro), 116
print.gg_varpro
(print.gg_beta_uvarpro), 116
print.gg_vimp (print.gg_beta_uvarpro),
116
print.summary.gg

(summary.gg_beta_uvarpro), 121
print.summary.gg_udependent
(print.gg_beta_uvarpro), 116

quantile, *118*
quantile_pts, 118

randomForest, *3*, *9*, *22*, *44*, *50, 51*, *53, 54*, *61*, *67, 68*, *80*, *96*, *101, 102*
randomForestRHF::auct.rhf(), *13, 14*
randomForestRHF::dotmatrix.importance.rhf(), *101*
randomForestRHF::importance.rhf(), *48*, *49*
randomForestRHF::rhf(), *46, 47*
randomForestRHF::tune.iAUC.rhf(), *57*
randomForestRHF::tune.rhf(), *57*
randomForestRHF::tune.treesize.rhf(), *57, 58*
randomForestRHF::varpro.cache.rhf(), *48*
rfsrc, *9*, *11*, *20*, *22*, *33*, *44, 45*, *50, 51*, *53–55*, *61*, *67–69*, *80, 81*, *95, 96*, *101, 102*, *105*, *110*, *114*, *124*

sdependent, *58*
shap_beeswarm, *4*, 119
shap_dependence, *4*, 120
shap_importance, *4*, *105*, 121
summary.gg, *118*
summary.gg (summary.gg_beta_uvarpro), 121
summary.gg_ale_interaction
(summary.gg_beta_uvarpro), 121
summary.gg_ale_rfsrc
(summary.gg_beta_uvarpro), 121
summary.gg_auct
(summary.gg_beta_uvarpro), 121
summary.gg_beta_uvarpro, 121
summary.gg_beta_varpro
(summary.gg_beta_uvarpro), 121
summary.gg_brier
(summary.gg_beta_uvarpro), 121
summary.gg_error
(summary.gg_beta_uvarpro), 121
summary.gg_isopro
(summary.gg_beta_uvarpro), 121
summary.gg_ivarpro
(summary.gg_beta_uvarpro), 121
summary.gg_partial
(summary.gg_beta_uvarpro), 121
summary.gg_partial_rfsrc
(summary.gg_beta_uvarpro), 121
summary.gg_partial_varpro
(summary.gg_beta_uvarpro), 121
summary.gg_partialpro
(summary.gg_beta_uvarpro), 121
summary.gg_rfsrc
(summary.gg_beta_uvarpro), 121
summary.gg_rhf
(summary.gg_beta_uvarpro), 121
summary.gg_rhf_importance
(summary.gg_beta_uvarpro), 121
summary.gg_roc
(summary.gg_beta_uvarpro), 121
summary.gg_sdependent
(summary.gg_beta_uvarpro), 121
summary.gg_shap
(summary.gg_beta_uvarpro), 121
summary.gg_survival
(summary.gg_beta_uvarpro), 121
summary.gg_tune_rhf
(summary.gg_beta_uvarpro), 121
summary.gg_udependent
(summary.gg_beta_uvarpro), 121
summary.gg_variable
(summary.gg_beta_uvarpro), 121
summary.gg_varpro
(summary.gg_beta_uvarpro), 121
summary.gg_vimp
(summary.gg_beta_uvarpro), 121
surv_partial.rfsrc, 124

varPro::beta.varpro(), *16*, *19*
varPro::get.beta.entropy(), *15, 16*, *52*, *77*
varPro::ivarpro(), *28, 29*, *31*
varPro::sdependent(), *52, 53*, *103*
varPro::uvarpro(), *15, 16*, *52, 53*, *77*, *103*
varPro::varpro(), *16*, *29*
varpro_coverage, *65–67*, 126
varpro_feature_names, *42*, *128*, 129
vimp, *68, 69*
vimp.rfsrc, *68*