Package 'ggRandomForests'

June 23, 2026

Type Package

Title Visually Exploring Random Forests

Version 3.2.0

Date 2026-06-22

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 the 'randomForest' or 'randomForestSRC' package for survival, regression and classification forests and 'ggplot2' package plotting. Implements visualisations of the methods described in Breiman (2001) <doi:10.1023/A:1010933404324> and Ishwaran, Kogalur, Blackstone, and Lauer (2008) <doi:10.1214/08-AOAS169>.

Depends R (>= 4.4.0)

Imports randomForestSRC (>= 3.4.0), randomForest, varPro (>= 3.1.0), igraph, survival, tidyr, dplyr, ggplot2, patchwork, stringr

Suggests testthat, bookdown, RColorBrewer, MASS, lintr, covr, vdiffr, datasets, rmarkdown, quarto, pkgdown, pkgload, knitr, ggraph, callr

VignetteBuilder quarto

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author John Ehrlinger [aut, cre]

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

Repository CRAN

Date/Publication 2026-06-23 17:10:02 UTC

Contents

ggRandomForests-package

ggRandomForests: Visually Exploring Random Forests

Description

ggRandomForests is a utility package for randomForestSRC (Ishwaran and Kogalur) for survival, regression and classification forests and uses the ggplot2 (Wickham 2009) package for plotting results. ggRandomForests is structured to extract data objects from the random forest and provides S3 functions for printing and plotting these objects. Requires randomForestSRC >= 3.4.0.

The randomForestSRC package provides a unified treatment of Breiman's (2001) random forests for a variety of data settings. Regression and classification forests are grown when the response is numeric or categorical (factor) while survival and competing risk forests (Ishwaran et al. 2008, 2012) are grown for right-censored survival data.

Many of the figures created by the ggRandomForests package are also available directly from within the randomForestSRC package. However, ggRandomForests offers the following advantages:

- Separation of data and figures: ggRandomForest contains functions that operate on either the rfsrc forest object directly, or on the output from randomForestSRC post processing functions (i.e. plot.variable) to generate intermediate ggRandomForests data objects. S3 functions are provide to further process these objects and plot results using the ggplot2 graphics package. Alternatively, users can use these data objects for additional custom plotting or analysis operations.
- Each data object/figure is a single, self contained object. This allows simple modification and manipulation of the data or ggplot2 objects to meet users specific needs and requirements.
- The use of ggplot2 for plotting. We chose to use the ggplot2 package for our figures to allow users flexibility in modifying the figures to their liking. Each S3 plot function returns either a single ggplot2 object, or a list of ggplot2 objects, allowing users to use additional ggplot2 functions or themes to modify and customize the figures to their liking.

The ggRandomForests package contains the following data functions:

- gg_rfsrc: randomForestSRC predictions.
- gg_error: randomForestSRC convergence rate based on the OOB error rate.
- gg_roc: ROC curves for randomForest classification models.
- gg_vimp: Variable Importance ranking for variable selection. (Ishwaran et.al. 2010).
- gg_variable: Marginal variable dependence.
- gg_survival: Kaplan-Meier/Nelson-Aalen hazard analysis.

Each of these data functions has an associated S3 plot function that returns ggplot2 objects, either individually or as a list, which can be further customized using standard ggplot2 commands.

References

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

Ishwaran H. and Kogalur U.B. randomForestSRC: Random Forests for Survival, Regression and Classification. R package version >= 3.4.0. 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.

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_varpro'
autoplot(object, ...)

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

## S3 method for class 'gg_isopro'
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

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\

See Also

calc_roc gg_roc

plot.gg_roc

Examples

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

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, ...)
```

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"` to request all classes (currently falls back to class 1 with a warning).
`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)

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_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 are set by mean(|sum-of-class-beta|) descending so every facet / panel shares the same row order. 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 **fitted coefficient** $\hat{\beta}$ at the chosen λ. **Sign is real** (direction of local association). **Magnitude depends on the predictor's units** (raw x, no standardisation); 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`, $\text{mean}(|\hat{\beta}|)$, 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 families are out of scope for this release and tracked for v3.1.0. The unsupported-family path errors with a message pointing at that work.

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 – 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" ceiling.

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 the continuous ranked probability score (CRPS) – the Brier score integrated over time and divided by elapsed time, a running average that summarises calibration up to each 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, normalised 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, normalised by elapsed time.

The integrated CRPS (a single scalar matching `get.brier.survival()$crps`) is attached as `attr(., "crps_integrated")`.

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
)
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
)
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, colour = 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

For randomForestSRC objects the function reshapes the rfsrc$err.rate matrix and annotates it 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. and Kogalur U.B. randomForestSRC: Random Forests for Survival, Regression and Classification. R package version >= 3.4.0. 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, tree.err = 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,
  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,
  na.action = "na.impute", tree.err = TRUE,
)

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

# Plot the gg_error object
plot(gg_dta)


## ------------- Boston data
data(Boston, package = "MASS")
Boston$chas <- as.logical(Boston$chas)
rfsrc_boston <- randomForestSRC::rfsrc(medv ~ .,
  data = Boston,
  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, 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,
                      tree.err = TRUE)

gg_dta <- gg_error(rfsrc_veteran)
plot(gg_dta)

## ------------- pbc data
# Load a cached randomForestSRC object
# We need to create this dataset
data(pbc, package = "randomForestSRC",)
# For whatever reason, the age variable is in days... makes no sense to me
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 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)
dta_train <- pbc[-which(is.na(pbc$treatment)), ]
# Create a test set from the remaining patients
pbc_test <- pbc[which(is.na(pbc$treatment)), ]

#========
# build the forest:
rfsrc_pbc <- randomForestSRC::rfsrc(
  Surv(years, status) ~ .,
 dta_train,
 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: splits are chosen to separate the data along the directions of highest variance. 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 colours 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 transformed onto a `[0, 1]` scale via the empirical CDF of `case.depth` (higher = more anomalous). 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 colour 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 colours 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., Mantero, A., and Lu, M. (2025). varPro: Model-Independent Variable Selection via the Rule-Based Variable Priority Framework. *R package version 3.x*.

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). 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 are set by mean(|local_imp|) descending across all rows (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) standardises 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 are set by `mean(|local_imp|)` descending across all rows; 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 (positive-class importances only - the wrapper normalises this to a single-element list under the last factor level). 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_imp|), 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 families are out of scope for this release. The non-regression / non-class path errors with a message naming v3.1.0 as the tracker.

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 `rfsrc::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 `rfsrc::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 bar charts). 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

Note

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

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_labelled <- gg_partial(pv, model = "airq_model")
unique(result_labelled$continuous$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$xvar.names.
xvar2.name	Optional single character name of a grouping variable in newx. When supplied, partial dependence is computed separately for each unique level of this variable and a grp column is appended.
newx	Optional data.frame of predictor values to evaluate partial effects at. Defaults to the training data stored in rf_model$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.
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 – 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 coloured lines.

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 only) for all continuous predictors.

categorical A `data.frame` with the same columns but `x` kept as character, for low-cardinality predictors.

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)
```

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.

See Also

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

Examples

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

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

## 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 the release after **ggRandomForests** v3.0.0; use `gg_partial_varpro()` directly.

Usage

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

gg_partialpro(
  part_dta,
  object = NULL,
  scale = c("auto", "rmst", "mortality", "surv", "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; sets the y-axis label and, for survival forests, the output type. One of "auto" (default), "mortality", "rmst", "surv", or "chf".
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 %in% c("surv","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 (list elements) to process. Defaults to every variable in part_dta.
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). Ignored, with a warning, when part_dta is supplied.

Details

Scale detection: with scale = "auto" and an object in hand, the scale resolves to "mortality" for a survival forest and "generic" for a regression or classification forest. The RMST horizon τ is *not* stored in the varpro object (varPro 3.1.0), 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 – passing a horizon through ... is silently dropped, and multi-horizon plots built that way differ only by Monte-Carlo noise, not by τ. 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.

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.

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 %in% c("surv","chf")`) takes a different route: we hand the underlying `rfsrc` forest to `gg_partial_rfsrc` so you get a survival-probability or cumulative-hazard curve on the usual rfsrc scale instead.

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.

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.

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)
  )
)
result <- gg_partial_varpro(mock_data)
head(result$continuous)
head(result$categorical)
```

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, ...)
```

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_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, ...)
```

Arguments

`object`	A classification `rfsrc` or `randomForest` object. Only forests with `family == "class"` (rfsrc) or `type == "classification"` (randomForest) are supported.
`which_outcome`	Integer index or character name of the class to score. For binary forests this is usually `1` or `2`; for multi-class forests, any valid class index or level name. `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. Honoured 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.

See Also

`plot.gg_roc`, `calc_roc`, `calc_auc`, `rfsrc`, `randomForest`

Examples

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

# 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_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

Comparing the forest's ensemble survival curve to the marginal Kaplan-Meier baseline is a quick sanity check: if they diverge the forest has found structure the predictors carry; if they track each other closely the predictors may add little. 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_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,
  ...
)
```

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().

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, 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.

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 summarises 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,
  ...
)
```

Arguments

`object`	A fitted `varpro` object (required).
`local.std`	Logical; default `TRUE`. When `TRUE` the per-tree importances are put on the z-scale before the box statistics are computed. Set it `FALSE` to keep the raw importance scale, which is what `type = "raw"` in `plot.gg_varpro` needs.
`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`.

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.
...	Additional arguments passed to varPro::importance().

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 from importance()), selected (logical, z > cutoff).

$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 (on z-scale when local.std = TRUE, raw when FALSE), plus mean (raw importance mean, always stored).

$conditional NULL when conditional = FALSE; otherwise a data frame with columns variable, class, z (one row per variable x class combination).

A "provenance" attribute carries family, local.std, cutoff, faithful, conditional, xvar.names, and n.

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 summarise here. With local.std = TRUE (the default) the per-tree values are standardised by their column standard deviation so the column mean equals the aggregate z-score returned by varPro::importance(); that z-score is the canonical "is this variable in or out?" statistic, and cutoff = 0.79 is varpro's default selection threshold.

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

`$imp` is the one-row-per-variable summary: aggregate z from `varPro::importance()`, plus a `selected` flag for `z > cutoff`. `$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 standardised 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`.

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`, `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	argument to control the number of variables included in the output.
...	arguments passed to the vimp.rfsrc function if the rfsrc object does not contain importance information.

Details

gg_vimp() shows **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.

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, optionally containing class-specific scores and a relative importance column. When randomForest objects lack stored importance values a warning is issued and NA placeholders 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,
  importance = TRUE
)
gg_dta <- gg_vimp(rfsrc_iris)
plot(gg_dta)

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

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


## -------- Boston data
data(Boston, package = "MASS")
rfsrc_boston <- randomForestSRC::rfsrc(medv ~ ., Boston,
  importance = TRUE
)
gg_dta <- gg_vimp(rfsrc_boston)
plot(gg_dta)

## -------- Boston data
rf_boston <- randomForest::randomForest(medv ~ ., Boston)
gg_dta <- gg_vimp(rf_boston)
plot(gg_dta)


## -------- mtcars data
rfsrc_mtcars <- randomForestSRC::rfsrc(mpg ~ .,
  data = mtcars,
  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 = 100,
  importance = TRUE
)

gg_dta <- gg_vimp(rfsrc_veteran)
plot(gg_dta)

## -------- pbc data
# We need to create this dataset
data(pbc, package = "randomForestSRC", )
# For whatever reason, the age variable is in days...
# makes no sense to me
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 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)
dta_train <- pbc[-which(is.na(pbc$treatment)), ]
# Create a test set from the remaining patients
pbc_test <- pbc[which(is.na(pbc$treatment)), ]

# ========
# build the forest:
rfsrc_pbc <- randomForestSRC::rfsrc(
  Surv(years, status) ~ .,
```

```
  dta_train,
  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

nonparametric Nelson-Aalen estimates

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`	for each observation (default=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 <- 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_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, grey otherwise. Dashed red line marks the cutoff.

Usage

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

Arguments

x	A gg_beta_varpro object from gg_beta_varpro().
...	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 coloured blue and flagged selected; variables below are grey. 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 ceiling.

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)
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 is used to track the convergence of the randomForest. This figure is a reproduction of 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 coloured 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. and Kogalur U.B. randomForestSRC: Random Forests for Survival, Regression and Classification. R package version >= 3.4.0. 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,
  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,
  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
data(Boston, package = "MASS")
Boston$chas <- as.logical(Boston$chas)
rfsrc_boston <- randomForestSRC::rfsrc(medv ~ .,
  data = Boston,
  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,
  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,
                       tree.err = TRUE)

gg_dta <- gg_error(rfsrc_veteran)
plot(gg_dta)

## ------------- pbc data
# Load a cached randomForestSRC object
# We need to create this dataset
data(pbc, package = "randomForestSRC",)
# For whatever reason, the age variable is in days... makes no sense to me
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 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)
dta_train <- pbc[-which(is.na(pbc$treatment)), ]
# Create a test set from the remaining patients
pbc_test <- pbc[which(is.na(pbc$treatment)), ]

#========
# build the forest:
rfsrc_pbc <- randomForestSRC::rfsrc(
  Surv(years, status) ~ .,
 dta_train,
 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\ 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 colour 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, ...)
```

Arguments

`x`	A `gg_ivarpro` object from `gg_ivarpro()`.
`...`	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 grey, 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 bar charts. 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, ...)
```

Arguments

x	A gg_partial object (output of gg_partial).
...	Not currently used; reserved for future arguments.

Details

When a model label was attached in gg_partial(), lines are coloured by model – 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, ...)
```

Arguments

x	A gg_partial_rfsrc object.
...	Not currently used.

Details

For a standard regression or classification forest, 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.

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 coloured curve over the predictor's value, still faceted by variable. 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 coloured lines, faceted by the primary predictor.

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"), ...)

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

Arguments

x	A gg_partial_varpro object.
type	Character vector; one or more of "parametric", "nonparametric", "causal". Defaults to all three. Ignored for path-C objects.
...	Unused for path-A objects; forwarded to plot.gg_partial_rfsrc for path-C objects.

Details

Ensemble mortality (scale = "mortality"): when the provenance scale is "mortality", the y-axis is labelled *"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 centre 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 = "surv"` or `"chf"`), the y-axis is on the probability or cumulative-hazard scale, which is usually the scale you want to report to a clinical audience.

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.

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)
plot(pp)
plot(pp, type = "parametric")
```

---

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, averaging 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, ...)
```

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.
...	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, coloured 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 colour 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 coloured 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. and Kogalur U.B. randomForestSRC: Random Forests for Survival, Regression and Classification. R package version >= 3.4.0. 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 -- skipped on CRAN)

data(pbc, package = "randomForestSRC")
# For whatever reason, the age variable is in days; convert to years
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)
# Remove test-set patients (those with no assigned treatment)
dta_train <- pbc[-which(is.na(pbc$treatment)), ]

set.seed(42)
rfsrc_pbc <- randomForestSRC::rfsrc(
  Surv(years, status) ~ .,
  dta_train,
  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_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. Hand it a forest and `gg_roc` is called for you.
`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, coloured 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 colour 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. and Kogalur U.B. randomForestSRC: Random Forests for Survival, Regression and Classification. R package version >= 3.4.0. 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_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_udependent` *Plot a* `gg_udependent` *variable dependency graph*

Description

Draws the dependency graph held in a `gg_udependent` object as a ggraph network. Node colour 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", ...)
```

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"`.
`...`	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 grey (`#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.

Grey, 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 neighbours 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,
  time,
  time_labels,
  panel = FALSE,
  oob = TRUE,
  points = TRUE,
  smooth = TRUE,
  ...
)
```

Arguments

x	gg_variable object created from a rfsrc object
xvar	variable (or list of variables) of interest.
time	For survival, one or more times of interest
time_labels	string labels for times
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)
...	arguments passed to the ggplot2 functions.

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. and Kogalur U.B. randomForestSRC: Random Forests for Survival, Regression and Classification. R package version >= 3.4.0. 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 visualisation
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, ...)
```

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.
`...`	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.

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 centre 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`). 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 coloured blue and flagged `selected = TRUE`, the rest are grey. 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, lbls, ...)
```

Arguments

`x`	`gg_vimp` object created from a `rfsrc` object
`relative`	should we plot vimp or relative vimp. Defaults to vimp.
`lbls`	A vector of alternative variable labels. Item names should be the same as the variable names.
`...`	optional arguments passed to gg_vimp if necessary

Details

Bars are coloured by the `positive` flag: a bar at or below zero (non-positive VIMP) is colour-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 colour 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. and Kogalur U.B. randomForestSRC: Random Forests for Survival, Regression and Classification. R package version >= 3.4.0. https://cran.r-project.org/package=randomForestSRC

See Also

gg_vimp

Examples

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

## ---------------------------------------------------------
## regression example
## ---------------------------------------------------------
## -------- air quality data
rfsrc_airq <- randomForestSRC::rfsrc(Ozone ~ ., airquality)
gg_dta <- gg_vimp(rfsrc_airq)
plot(gg_dta)
```

`print.gg` *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_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_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, ...)
```

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

```
data(Boston, package = "MASS")
rfsrc_boston <- randomForestSRC::rfsrc(medv ~ ., Boston)

# 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)
```

---

summary.gg *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 '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_isopro'
summary(object, ...)

## S3 method for class 'gg_beta_varpro'
summary(object, ...)

## S3 method for class 'gg_ivarpro'
summary(object, ...)
```

Arguments

x	A summary.gg object (for print.summary.gg).
...	Not currently used.
object	A gg_* data object.

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_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