Package 'vip'

August 21, 2023

Type Package

Title Variable Importance Plots

Version 0.4.1

Description A general framework for constructing variable importance plots from various types of machine learning models in R. Aside from some standard model-specific variable importance measures, this package also provides model-agnostic approaches that can be applied to any supervised learning algorithm. These include 1) an efficient permutation-based variable importance measure, 2) variable importance based on Shapley values (Strumbelj and Kononenko, 2014) <doi:10.1007/s10115-013-0679-x>, and 3) the variance-based approach described in Greenwell et al. (2018) <arXiv:1805.04755>. A variance-based method for quantifying the relative strength of interaction effects is also included (see the previous reference for details).

License GPL (>= 2)

URL `https://github.com/koalaverse/vip/`, `https://koalaverse.github.io/vip/`

BugReports `https://github.com/koalaverse/vip/issues`

Encoding UTF-8

VignetteBuilder knitr

Depends R (>= 4.1.0)

Imports foreach, ggplot2 (>= 0.9.0), stats, tibble, utils, yardstick

Suggests bookdown, DT, covr, doParallel, dplyr, fastshap (>= 0.1.0), knitr, lattice, mlbench, modeldata, NeuralNetTools, pdp, rmarkdown, tinytest (>= 1.4.1), varImp

Enhances C50, caret, Cubist, earth, gbm, glmnet, h2o, lightgbm, mixOmics, mlr, mlr3, neuralnet, nnet, parsnip (>= 0.1.7), party, partykit, pls, randomForest, ranger, rpart, RSNNS, sparklyr (>= 0.8.0), tidymodels, workflows (>= 0.2.3), xgboost

LazyData true

RoxygenNote 7.2.3

NeedsCompilation no

Author Brandon M. Greenwell [aut, cre]
(<`https://orcid.org/0000-0002-8120-0084`>),
Brad Boehmke [aut] (<`https://orcid.org/0000-0002-3611-8516`>)

Maintainer Brandon M. Greenwell <`greenwell.brandon@gmail.com`>

Repository CRAN

Date/Publication 2023-08-21 09:20:02 UTC

R topics documented:

`gen_friedman` *Friedman benchmark data*

Description

Simulate data from the Friedman 1 benchmark problem. These data were originally described in Friedman (1991) and Breiman (1996). For details, see sklearn.datasets.make_friedman1.

Usage

```
gen_friedman(
  n_samples = 100,
  n_features = 10,
  n_bins = NULL,
  sigma = 0.1,
  seed = NULL
)
```

Arguments

`n_samples`	Integer specifying the number of samples (i.e., rows) to generate. Default is 100.
`n_features`	Integer specifying the number of features to generate. Default is 10.
`n_bins`	Integer specifying the number of (roughly) equal sized bins to split the response into. Default is `NULL` for no binning. Setting to a positive integer > 1 effectively turns this into a classification problem where `n_bins` gives the number of classes.
`sigma`	Numeric specifying the standard deviation of the noise.
`seed`	Integer specifying the random seed. If `NULL` (the default) the results will be different each time the function is run.

References

Breiman, Leo (1996) Bagging predictors. Machine Learning 24, pages 123-140.

Friedman, Jerome H. (1991) Multivariate adaptive regression splines. The Annals of Statistics 19 (1), pages 1-67.

Examples

```
gen_friedman()
```

`list_metrics` *List metrics*

Description

List all available performance metrics.

Usage

```
list_metrics()
```

Value

A data frame with the following columns:

- `metric` - the optimization or tuning metric;
- `description` - a brief description about the metric;
- `task` - whether the metric is suitable for regression or classification;
- `smaller_is_better` - logical indicating whether or not a smaller value of the metric is considered better.
- `yardstick_function` - the name of the corresponding function from the yardstick package.

Examples

```
(metrics <- list_metrics())
metrics[metrics$task == "Multiclass classification", ]
```

`titanic` *Survival of Titanic passengers*

Description

A data set containing the survival outcome, passenger class, age, sex, and the number of family members for a large number of passengers aboard the ill-fated Titanic.

Usage

```
titanic
```

Format

A data frame with 1309 observations on the following 6 variables:

- `survived` - binary with levels `"yes"` for survived and `"no"` otherwise;
- `pclass` - integer giving the corresponding passenger (i.e., ticket) class with values 1–3;
- `age` - the age in years of the corresponding passenger (with 263 missing values);
- `age` - factor giving the sex of each passenger with levels `"male"` and `"female"`;
- `sibsp` - integer giving the number of siblings/spouses aboard for each passenger (ranges from 0–8);
- `parch` - integer giving the number of parents/children aboard for each passenger (ranges from 0–9).

Note

As mentioned in the column description, `age` contains 263 `NA`s (or missing values). For a complete version (or versions) of the data set, see titanic_mice.

Source

`https://hbiostat.org/data/`.

`titanic_mice` *Survival of Titanic passengers*

Description

The titanic data set contains 263 missing values (i.e., `NA`'s) in the `age` column. This version of the data contains imputed values for the `age` column using *multivariate imputation by chained equations* via the mice package. Consequently, this is a list containing 11 imputed versions of the observations containd in the titanic data frame; each completed data sets has the same dimension and column structure as titanic.

Usage

```
titanic_mice
```

Format

An object of class `mild` (inherits from `list`) of length 21.

Source

Greenwell, Brandon M. (2022). Tree-Based Methods for Statistical Learning in R. CRC Press.

`vi` *Variable importance*

Description

Compute variable importance scores for the predictors in a model.

Usage

```
vi(object, ...)

## Default S3 method:
vi(
  object,
  method = c("model", "firm", "permute", "shap"),
  feature_names = NULL,
  abbreviate_feature_names = NULL,
  sort = TRUE,
  decreasing = TRUE,
  scale = FALSE,
  rank = FALSE,
  ...
)
```

Arguments

`object`	A fitted model object (e.g., a randomForest object) or an object that inherits from class `"vi"`.
`...`	Additional optional arguments to be passed on to vi_model, vi_firm, vi_permute, or vi_shap; see their respective help pages for details.
`method`	Character string specifying the type of variable importance (VI) to compute. Current options are: • `"model"` (the default), for model-specific VI scores (see vi_model for details). • `"firm"`, for variance-based VI scores (see vi_firm for details).

- "permute", for permutation-based VI scores (see vi_permute for details).
- "shap", for Shapley-based VI scores (see vi_shap for details).

feature_names	Character string giving the names of the predictor variables (i.e., features) of interest.
abbreviate_feature_names	Integer specifying the length at which to abbreviate feature names. Default is NULL which results in no abbreviation (i.e., the full name of each feature will be printed).
sort	Logical indicating whether or not to order the sort the variable importance scores. Default is TRUE.
decreasing	Logical indicating whether or not the variable importance scores should be sorted in descending (TRUE) or ascending (FALSE) order of importance. Default is TRUE.
scale	Logical indicating whether or not to scale the variable importance scores so that the largest is 100. Default is FALSE.
rank	Logical indicating whether or not to rank the variable importance scores (i.e., convert to integer ranks). Default is FALSE. Potentially useful when comparing variable importance scores across different models using different methods.

Value

A tidy data frame (i.e., a tibble object) with two columns:

- Variable - the corresponding feature name;
- Importance - the associated importance, computed as the average change in performance after a random permutation (or permutations, if nsim > 1) of the feature in question.

For lm/glm-like objects, whenever method = "model", the sign (i.e., POS/NEG) of the original coefficient is also included in a column called Sign.

If method = "permute" and nsim > 1, then an additional column (StDev) containing the standard deviation of the individual permutation scores for each feature is also returned; this helps assess the stability/variation of the individual permutation importance for each feature.

Examples

```
#
# A projection pursuit regression example
#

# Load the sample data
data(mtcars)

# Fit a projection pursuit regression model
mtcars.ppr <- ppr(mpg ~ ., data = mtcars, nterms = 1)

# Prediction wrapper that tells vi() how to obtain new predictions from your
# fitted model
pfun <- function(object, newdata) predict(object, newdata = newdata)
```

```
# Compute permutation-based variable importance scores
set.seed(1434)  # for reproducibility
(vis <- vi(mtcars.ppr, method = "permute", target = "mpg", nsim = 10,
           metric = "rmse", pred_wrapper = pfun, train = mtcars))

# Plot variable importance scores
vip(vis, include_type = TRUE, all_permutations = TRUE,
    geom = "point", aesthetics = list(color = "forestgreen", size = 3))

#
# A binary classification example
#
## Not run:
library(rpart)  # for classification and regression trees

# Load Wisconsin breast cancer data; see ?mlbench::BreastCancer for details
data(BreastCancer, package = "mlbench")
bc <- subset(BreastCancer, select = -Id)  # for brevity

# Fit a standard classification tree
set.seed(1032)  # for reproducibility
tree <- rpart(Class ~ ., data = bc, cp = 0)

# Prune using 1-SE rule (e.g., use `plotcp(tree)` for guidance)
cp <- tree$cptable
cp <- cp[cp[, "nsplit"] == 2L, "CP"]
tree2 <- prune(tree, cp = cp)  # tree with three splits

# Default tree-based VIP
vip(tree2)

# Computing permutation importance requires a prediction wrapper. For
# classification, the return value depends on the chosen metric; see
# `?vip::vi_permute` for details.
pfun <- function(object, newdata) {
  # Need vector of predicted class probabilities when using  log-loss metric
  predict(object, newdata = newdata, type = "prob")[, "malignant"]
}

# Permutation-based importance (note that only the predictors that show up
# in the final tree have non-zero importance)
set.seed(1046)  # for reproducibility
vi(tree2, method = "permute", nsim = 10, target = "Class", train = bc,
   metric = "logloss", pred_wrapper = pfun, reference_class = "malignant")

# Equivalent (but not sorted)
set.seed(1046)  # for reproducibility
vi_permute(tree2, nsim = 10, target = "Class", metric = "logloss",
           pred_wrapper = pfun, reference_class = "malignant")

## End(Not run)
```

vip — *Variable importance plots*

Description

Plot variable importance scores for the predictors in a model.

Usage

```
vip(object, ...)

## Default S3 method:
vip(
  object,
  num_features = 10L,
  geom = c("col", "point", "boxplot", "violin"),
  mapping = NULL,
  aesthetics = list(),
  horizontal = TRUE,
  all_permutations = FALSE,
  jitter = FALSE,
  include_type = FALSE,
  ...
)

## S3 method for class 'model_fit'
vip(object, ...)

## S3 method for class 'workflow'
vip(object, ...)

## S3 method for class 'WrappedModel'
vip(object, ...)

## S3 method for class 'Learner'
vip(object, ...)
```

Arguments

`object`	A fitted model (e.g., of class randomForest object) or a vi object.
`...`	Additional optional arguments to be passed on to vi.
`num_features`	Integer specifying the number of variable importance scores to plot. Default is `10`.
`geom`	Character string specifying which type of plot to construct. The currently available options are described below.

- geom = "col" uses geom_col to construct a bar chart of the variable importance scores.
- geom = "point" uses geom_point to construct a Cleveland dot plot of the variable importance scores.
- geom = "boxplot" uses geom_boxplot to construct a boxplot plot of the variable importance scores. This option can only for the permutation-based importance method with nsim > 1 and keep = TRUE; see vi_permute for details.
- geom = "violin" uses geom_violin to construct a violin plot of the variable importance scores. This option can only for the permutation-based importance method with nsim > 1 and keep = TRUE; see vi_permute for details.

`mapping`	Set of aesthetic mappings created by aes-related functions and/or tidy eval helpers. See example usage below.
`aesthetics`	List specifying additional arguments passed on to layer. These are often aesthetics, used to set an aesthetic to a fixed value, likecolour = "red" or size = 3. See example usage below.
`horizontal`	Logical indicating whether or not to plot the importance scores on the x-axis (TRUE). Default is TRUE.
`all_permutations`	Logical indicating whether or not to plot all permutation scores along with the average. Default is FALSE. (Only used for permutation scores when nsim > 1.)
`jitter`	Logical indicating whether or not to jitter the raw permutation scores. Default is FALSE. (Only used when all_permutations = TRUE.)
`include_type`	Logical indicating whether or not to include the type of variable importance computed in the axis label. Default is FALSE.

Examples

```
#
# A projection pursuit regression example using permutation-based importance
#

# Load the sample data
data(mtcars)

# Fit a projection pursuit regression model
model <- ppr(mpg ~ ., data = mtcars, nterms = 1)

# Construct variable importance plot (permutation importance, in this case)
set.seed(825)  # for reproducibility
pfun <- function(object, newdata) predict(object, newdata = newdata)
vip(model, method = "permute", train = mtcars, target = "mpg", nsim = 10,
    metric = "rmse", pred_wrapper = pfun)

# Better yet, store the variable importance scores and then plot
set.seed(825)  # for reproducibility
vis <- vi(model, method = "permute", train = mtcars, target = "mpg",
          nsim = 10, metric = "rmse", pred_wrapper = pfun)
```

```
vip(vis, geom = "point", horiz = FALSE)
vip(vis, geom = "point", horiz = FALSE, aesthetics = list(size = 3))

# Plot unaggregated permutation scores (boxplot colored by feature)
library(ggplot2)  # for `aes()`-related functions and tidy eval helpers
vip(vis, geom = "boxplot", all_permutations = TRUE, jitter = TRUE,
    #mapping = aes_string(fill = "Variable"),   # for ggplot2 (< 3.0.0)
    mapping = aes(fill = .data[["Variable"]]),  # for ggplot2 (>= 3.0.0)
    aesthetics = list(color = "grey35", size = 0.8))

#
# A binary classification example
#
## Not run:
library(rpart)  # for classification and regression trees

# Load Wisconsin breast cancer data; see ?mlbench::BreastCancer for details
data(BreastCancer, package = "mlbench")
bc <- subset(BreastCancer, select = -Id)  # for brevity

# Fit a standard classification tree
set.seed(1032)  # for reproducibility
tree <- rpart(Class ~ ., data = bc, cp = 0)

# Prune using 1-SE rule (e.g., use `plotcp(tree)` for guidance)
cp <- tree$cptable
cp <- cp[cp[, "nsplit"] == 2L, "CP"]
tree2 <- prune(tree, cp = cp)  # tree with three splits

# Default tree-based VIP
vip(tree2)

# Computing permutation importance requires a prediction wrapper. For
# classification, the return value depends on the chosen metric; see
# `?vip::vi_permute` for details.
pfun <- function(object, newdata) {
  # Need vector of predicted class probabilities when using  log-loss metric
  predict(object, newdata = newdata, type = "prob")[, "malignant"]
}

# Permutation-based importance (note that only the predictors that show up
# in the final tree have non-zero importance)
set.seed(1046)  # for reproducibility
vip(tree2, method = "permute", nsim = 10, target = "Class",
    metric = "logloss", pred_wrapper = pfun, reference_class = "malignant")

## End(Not run)
```

vi_firm *Variance-based variable importance*

Description

Compute variance-based variable importance (VI) scores using a simple *feature importance ranking measure* (FIRM) approach; for details, see Greenwell et al. (2018) and Scholbeck et al. (2019).

Usage

```
vi_firm(object, ...)

## Default S3 method:
vi_firm(
  object,
  feature_names = NULL,
  train = NULL,
  var_fun = NULL,
  var_continuous = stats::sd,
  var_categorical = function(x) diff(range(x))/4,
  ...
)
```

Arguments

`object`	A fitted model object (e.g., a randomForest object).
`...`	Additional arguments to be passed on to the `pdp::partial()` function (e.g., `ice = TRUE`, `prob = TRUE`, or a prediction wrapper via the `pred.fun` argument); see `?pdp::partial` for details on these and other useful arguments.
`feature_names`	Character string giving the names of the predictor variables (i.e., features) of interest. If `NULL` (the default) then the internal `get_feature_names()` function will be called to try and extract them automatically. It is good practice to always specify this argument.
`train`	A matrix-like R object (e.g., a data frame or matrix) containing the training data. If `NULL` (the default) then the internal `get_training_data()` function will be called to try and extract it automatically. It is good practice to always specify this argument.
`var_fun`	Deprecated; use `var_continuous` and `var_categorical` instead.
`var_continuous`	Function used to quantify the variability of effects for continuous features. Defaults to using the sample standard deviation (i.e., `stats::sd()`).
`var_categorical`	Function used to quantify the variability of effects for categorical features. Defaults to using the range divided by four; that is, `function(x) diff(range(x)) / 4`.

Details

This approach is based on quantifying the relative "flatness" of the effect of each feature and assumes the user has some familiarity with the `pdp::partial()` function. The Feature effects can be assessed using *partial dependence* (PD) plots (Friedman, 2001) or *individual conditional expectation* (ICE) plots (Goldstein et al., 2014). These methods are model-agnostic and can be applied to any supervised learning algorithm. By default, relative "flatness" is defined by computing the standard deviation of the y-axis values for each feature effect plot for numeric features; for categorical features, the default is to use range divided by 4. This can be changed via the `var_continuous` and `var_categorical` arguments. See Greenwell et al. (2018) for details and additional examples.

Value

A tidy data frame (i.e., a tibble object) with two columns:

- `Variable` - the corresponding feature name;
- `Importance` - the associated importance, computed as described in Greenwell et al. (2018).

Note

This approach can provide misleading results in the presence of interaction effects (akin to interpreting main effect coefficients in a linear with higher level interaction effects).

References

J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, **29**: 1189-1232, 2001.

Goldstein, A., Kapelner, A., Bleich, J., and Pitkin, E., Peeking Inside the Black Box: Visualizing Statistical Learning With Plots of Individual Conditional Expectation. (2014) *Journal of Computational and Graphical Statistics*, **24**(1): 44-65, 2015.

Greenwell, B. M., Boehmke, B. C., and McCarthy, A. J. A Simple and Effective Model-Based Variable Importance Measure. arXiv preprint arXiv:1805.04755 (2018).

Scholbeck, C. A. Scholbeck, and Molnar, C., and Heumann C., and Bischl, B., and Casalicchio, G. Sampling, Intervention, Prediction, Aggregation: A Generalized Framework for Model-Agnostic Interpretations. arXiv preprint arXiv:1904.03959 (2019).

Examples

```
## Not run:
#
# A projection pursuit regression example
#

# Load the sample data
data(mtcars)

# Fit a projection pursuit regression model
mtcars.ppr <- ppr(mpg ~ ., data = mtcars, nterms = 1)

# Compute variable importance scores using the FIRM method; note that the pdp
# package knows how to work with a "ppr" object, so there's no need to pass
# the training data or a prediction wrapper, but it's good practice.
vi_firm(mtcars.ppr, train = mtcars)

# For unsopported models, need to define a prediction wrapper; this approach
# will work for ANY model (supported or unsupported, so better to just always
# define it pass it)
pfun <- function(object, newdata) {
  # To use partial dependence, this function needs to return the AVERAGE
  # prediction (for ICE, simply omit the averaging step)
  mean(predict(object, newdata = newdata))
}

# Equivalent to the previous results (but would work if this type of model
# was not explicitly supported)
vi_firm(mtcars.ppr, pred.fun = pfun, train = mtcars)

# Equivalent VI scores, but the output is sorted by default
vi(mtcars.ppr, method = "firm")

# Use MAD to estimate variability of the partial dependence values
vi_firm(mtcars.ppr, var_continuous = stats::mad)

# Plot VI scores
vip(mtcars.ppr, method = "firm", train = mtcars, pred.fun = pfun)

## End(Not run)
```

vi_model *Model-specific variable importance*

Description

Compute model-specific variable importance scores for the predictors in a fitted model.

Usage

```
vi_model(object, ...)

## Default S3 method:
vi_model(object, ...)

## S3 method for class 'C5.0'
vi_model(object, type = c("usage", "splits"), ...)

## S3 method for class 'train'
vi_model(object, ...)

## S3 method for class 'cubist'
vi_model(object, ...)

## S3 method for class 'earth'
vi_model(object, type = c("nsubsets", "rss", "gcv"), ...)

## S3 method for class 'gbm'
```

```
vi_model(object, type = c("relative.influence", "permutation"), ...)

## S3 method for class 'glmnet'
vi_model(object, lambda = NULL, ...)

## S3 method for class 'cv.glmnet'
vi_model(object, lambda = NULL, ...)

## S3 method for class 'H2OBinomialModel'
vi_model(object, ...)

## S3 method for class 'H2OMultinomialModel'
vi_model(object, ...)

## S3 method for class 'H2ORegressionModel'
vi_model(object, ...)

## S3 method for class 'lgb.Booster'
vi_model(object, type = c("gain", "cover", "frequency"), ...)

## S3 method for class 'mixo_pls'
vi_model(object, ncomp = NULL, ...)

## S3 method for class 'mixo_spls'
vi_model(object, ncomp = NULL, ...)

## S3 method for class 'WrappedModel'
vi_model(object, ...)

## S3 method for class 'Learner'
vi_model(object, ...)

## S3 method for class 'nn'
vi_model(object, type = c("olden", "garson"), ...)

## S3 method for class 'nnet'
vi_model(object, type = c("olden", "garson"), ...)

## S3 method for class 'RandomForest'
vi_model(object, type = c("accuracy", "auc"), ...)

## S3 method for class 'constparty'
vi_model(object, ...)

## S3 method for class 'cforest'
vi_model(object, ...)

## S3 method for class 'mvr'
```

```
vi_model(object, ...)

## S3 method for class 'mixo_pls'
vi_model(object, ncomp = NULL, ...)

## S3 method for class 'mixo_spls'
vi_model(object, ncomp = NULL, ...)

## S3 method for class 'WrappedModel'
vi_model(object, ...)

## S3 method for class 'Learner'
vi_model(object, ...)

## S3 method for class 'randomForest'
vi_model(object, ...)

## S3 method for class 'ranger'
vi_model(object, ...)

## S3 method for class 'rpart'
vi_model(object, ...)

## S3 method for class 'mlp'
vi_model(object, type = c("olden", "garson"), ...)

## S3 method for class 'ml_model_decision_tree_regression'
vi_model(object, ...)

## S3 method for class 'ml_model_decision_tree_classification'
vi_model(object, ...)

## S3 method for class 'ml_model_gbt_regression'
vi_model(object, ...)

## S3 method for class 'ml_model_gbt_classification'
vi_model(object, ...)

## S3 method for class 'ml_model_generalized_linear_regression'
vi_model(object, ...)

## S3 method for class 'ml_model_linear_regression'
vi_model(object, ...)

## S3 method for class 'ml_model_random_forest_regression'
vi_model(object, ...)

## S3 method for class 'ml_model_random_forest_classification'
```

```
vi_model(object, ...)

## S3 method for class 'lm'
vi_model(object, type = c("stat", "raw"), ...)

## S3 method for class 'model_fit'
vi_model(object, ...)

## S3 method for class 'workflow'
vi_model(object, ...)

## S3 method for class 'xgb.Booster'
vi_model(object, type = c("gain", "cover", "frequency"), ...)
```

Arguments

`object`	A fitted model object (e.g., a randomForest object). See the details section below to see how variable importance is computed for supported model types.
`...`	Additional optional arguments to be passed on to other methods. See the details section below for arguments that can be passed to specific object types.
`type`	Character string specifying the type of variable importance to return (only used for some models). See the details section below for which methods this argument applies to.
`lambda`	Numeric value for the penalty parameter of a glmnet model (this is equivalent to the `s` argument in coef.glmnet). See the section on glmnet in the details below.
`ncomp`	An integer for the number of partial least squares components to be used in the importance calculations. If more components are requested than were used in the model, all of the model's components are used.

Details

Computes model-specific variable importance scores depending on the class of `object`:

- C5.0 - Variable importance is measured by determining the percentage of training set samples that fall into all the terminal nodes after the split. For example, the predictor in the first split automatically has an importance measurement of 100 percent since all samples are affected by this split. Other predictors may be used frequently in splits, but if the terminal nodes cover only a handful of training set samples, the importance scores may be close to zero. The same strategy is applied to rule-based models and boosted versions of the model. The underlying function can also return the number of times each predictor was involved in a split by using the option `metric = "usage"`. See C5imp for details.
- cubist - The Cubist output contains variable usage statistics. It gives the percentage of times where each variable was used in a condition and/or a linear model. Note that this output will probably be inconsistent with the rules shown in the output from summary.cubist. At each split of the tree, Cubist saves a linear model (after feature selection) that is allowed to have terms for each variable used in the current split or any split above it. Quinlan (1992) discusses a smoothing algorithm where each model prediction is a linear combination of the parent and child model along the tree. As such, the final prediction is a function of all the linear models

from the initial node to the terminal node. The percentages shown in the Cubist output reflects all the models involved in prediction (as opposed to the terminal models shown in the output). The variable importance used here is a linear combination of the usage in the rule conditions and the model. See summary.cubist and varImp for details.

- glmnet - Similar to (generalized) linear models, the absolute value of the coefficients are returned for a specific model. It is important that the features (and hence, the estimated coefficients) be standardized prior to fitting the model. You can specify which coefficients to return by passing the specific value of the penalty parameter via the `lambda` argument (this is equivalent to the `s` argument in coef.glmnet). By default, `lambda = NULL` and the coefficients corresponding to the final penalty value in the sequence are returned; in other words, you should ALWAYS SPECIFY `lambda`! For cv.glmnet objects, the largest value of lambda such that the error is within one standard error of the minimum is used by default. For a multinomial response, the coefficients corresponding to the first class are used; that is, the first component of coef.glmnet.
- cforest - Variable importance is measured in a way similar to those computed by importance. Besides the standard version, a conditional version is available that adjusts for correlations between predictor variables. If `conditional = TRUE`, the importance of each variable is computed by permuting within a grid defined by the predictors that are associated (with 1 - *p*-value greater than threshold) to the variable of interest. The resulting variable importance score is conditional in the sense of beta coefficients in regression models, but represents the effect of a variable in both main effects and interactions. See Strobl et al. (2008) for details. Note, however, that all random forest results are subject to random variation. Thus, before interpreting the importance ranking, check whether the same ranking is achieved with a different random seed - or otherwise increase the number of trees ntree in ctree_control. Note that in the presence of missings in the predictor variables the procedure described in Hapfelmeier et al. (2012) is performed. See varimp for details.
- earth - The earth package uses three criteria for estimating the variable importance in a MARS model (see evimp for details):
  - The `nsubsets` criterion (`type = "nsubsets"`) counts the number of model subsets that include each feature. Variables that are included in more subsets are considered more important. This is the criterion used by summary.earth to print variable importance. By "subsets" we mean the subsets of terms generated by `earth()`'s backward pass. There is one subset for each model size (from one to the size of the selected model) and the subset is the best set of terms for that model size. (These subsets are specified in the `$prune.terms` component of `earth()`'s return value.) Only subsets that are smaller than or equal in size to the final model are used for estimating variable importance. This is the default method used by vi_model.
  - The `rss` criterion (`type = "rss"`) first calculates the decrease in the RSS for each subset relative to the previous subset during `earth()`'s backward pass. (For multiple response models, RSS's are calculated over all responses.) Then for each variable it sums these decreases over all subsets that include the variable. Finally, for ease of interpretation the summed decreases are scaled so the largest summed decrease is 100. Variables which cause larger net decreases in the RSS are considered more important.
  - The `gcv` criterion (`type = "gcv"`) is similar to the `rss` approach, but uses the GCV statistic instead of the RSS. Note that adding a variable can sometimes increase the GCV. (Adding the variable has a deleterious effect on the model, as measured in terms of its estimated predictive power on unseen data.) If that happens often enough, the variable can have a negative total importance, and thus appear less important than unused variables.

- gbm - Variable importance is computed using one of two approaches (See summary.gbm for details):
  - The standard approach (`type = "relative.influence"`) described in Friedman (2001). When `distribution = "gaussian"` this returns the reduction of squared error attributable to each variable. For other loss functions this returns the reduction attributable to each variable in sum of squared error in predicting the gradient on each iteration. It describes the *relative influence* of each variable in reducing the loss function. This is the default method used by vi_model.
  - An experimental permutation-based approach (`type = "permutation"`). This method randomly permutes each predictor variable at a time and computes the associated reduction in predictive performance. This is similar to the variable importance measures Leo Breiman uses for random forests, but gbm currently computes using the entire training dataset (not the out-of-bag observations).
- H2OModel - See h2o.varimp or visit `https://docs.h2o.ai/h2o/latest-stable/h2o-docs/variable-importance.html` for details.
- nnet - Two popular methods for constructing variable importance scores with neural networks are the Garson algorithm (Garson 1991), later modified by Goh (1995), and the Olden algorithm (Olden et al. 2004). For both algorithms, the basis of these importance scores is the network's connection weights. The Garson algorithm determines variable importance by identifying all weighted connections between the nodes of interest. Olden's algorithm, on the other hand, uses the product of the raw connection weights between each input and output neuron and sums the product across all hidden neurons. This has been shown to outperform the Garson method in various simulations. For DNNs, a similar method due to Gedeon (1997) considers the weights connecting the input features to the first two hidden layers (for simplicity and speed); but this method can be slow for large networks.. To implement the Olden and Garson algorithms, use `type = "olden"` and `type = "garson"`, respectively. See garson and olden for details.
- lm/glm - In (generalized) linear models, variable importance is typically based on the absolute value of the corresponding t-statistics (Bring, 1994). For such models, the sign of the original coefficient is also returned. By default, `type = "stat"` is used; however, if the inputs have been appropriately standardized then the raw coefficients can be used with `type = "raw"`. Note that Bring (1994) provides motivation for using the absolute value of the associated t-statistics.
- sparklyr - The Spark ML library provides standard variable importance measures for tree-based methods (e.g., random forests). See ml_feature_importances for details.
- randomForest Random forests typically provide two measures of variable importance.
  - The first measure is computed from permuting out-of-bag (OOB) data: for each tree, the prediction error on the OOB portion of the data is recorded (error rate for classification and MSE for regression). Then the same is done after permuting each predictor variable. The difference between the two are then averaged over all trees in the forest, and normalized by the standard deviation of the differences. If the standard deviation of the differences is equal to 0 for a variable, the division is not done (but the average is almost always equal to 0 in that case).
  - The second measure is the total decrease in node impurities from splitting on the variable, averaged over all trees. For classification, the node impurity is measured by the Gini index. For regression, it is measured by residual sum of squares.

See importance for details, including additional arguments that can be passed via the `...` argument in vi_model.

- cforest - Same approach described in cforest (from package **partykit**) above. See varimp and varimpAUC (if `type = "auc"`) for details.
- ranger - Variable importance for ranger objects is computed in the usual way for random forests. The approach used depends on the `importance` argument provided in the initial call to ranger. See importance for details.
- rpart - As stated in one of the rpart vignettes. A variable may appear in the tree many times, either as a primary or a surrogate variable. An overall measure of variable importance is the sum of the goodness of split measures for each split for which it was the primary variable, plus "goodness" * (adjusted agreement) for all splits in which it was a surrogate. Imagine two variables which were essentially duplicates of each other; if we did not count surrogates, they would split the importance with neither showing up as strongly as it should. See rpart for details.
- caret - Various model-specific and model-agnostic approaches that depend on the learning algorithm employed in the original call to caret. See varImp for details.
- xgboost - For linear models, the variable importance is the absolute magnitude of the estimated coefficients. For that reason, in order to obtain a meaningful ranking by importance for a linear model, the features need to be on the same scale (which you also would want to do when using either L1 or L2 regularization). Otherwise, the approach described in Friedman (2001) for gbms is used. See xgb.importance for details. For tree models, you can obtain three different types of variable importance:
  - Using `type = "gain"` (the default) gives the fractional contribution of each feature to the model based on the total gain of the corresponding feature's splits.
  - Using `type = "cover"` gives the number of observations related to each feature.
  - Using `type = "frequency"` gives the percentages representing the relative number of times each feature has been used throughout each tree in the ensemble.
- lightgbm - Same as for xgboost models, except lgb.importance (which this method calls internally) has an additional argument, `percentage`, that defaults to `TRUE`, resulting in the VI scores shown as a relative percentage; pass `percentage = FALSE` in the call to `vi_model()` to produce VI scores for lightgbm models on the raw scale.

Value

A tidy data frame (i.e., a tibble object) with two columns:

- `Variable` - the corresponding feature name;
- `Importance` - the associated importance, computed as the average change in performance after a random permutation (or permutations, if `nsim > 1`) of the feature in question.

For lm/glm-like objects, the sign (i.e., POS/NEG) of the original coefficient is also included in a column called `Sign`.

Note

Inspired by the caret's varImp function.

Source

Johan Bring (1994) How to Standardize Regression Coefficients, The American Statistician, 48:3, 209-213, DOI: 10.1080/00031305.1994.10476059.

Examples

```
## Not run:
# Basic example using imputed titanic data set
t3 <- titanic_mice[[1L]]

# Fit a simple model
set.seed(1449)  # for reproducibility
bst <- lightgbm::lightgbm(
  data = data.matrix(subset(t3, select = -survived)),
  label = ifelse(t3$survived == "yes", 1, 0),
  params = list("objective" = "binary", "force_row_wise" = TRUE),
  verbose = 0
)

# Compute VI scores
vi(bst)  # defaults to `method = "model"`
vi_model(bst)  # same as above

# Same as above (since default is `method = "model"`), but returns a plot
vip(bst, geom = "point")
vi_model(bst, type = "cover")
vi_model(bst, type = "cover", percentage = FALSE)

# Compare to
lightgbm::lgb.importance(bst)

## End(Not run)
```

`vi_permute` *Permutation-based variable importance*

Description

Compute permutation-based variable importance scores for the predictors in a model; for details on the algorithm, see Greenwell and Boehmke (2020).

Usage

```
vi_permute(object, ...)

## Default S3 method:
vi_permute(
  object,
  feature_names = NULL,
  train = NULL,
  target = NULL,
  metric = NULL,
  smaller_is_better = NULL,
  type = c("difference", "ratio"),
  nsim = 1,
  keep = TRUE,
  sample_size = NULL,
  sample_frac = NULL,
  reference_class = NULL,
  event_level = NULL,
  pred_wrapper = NULL,
  verbose = FALSE,
  parallel = FALSE,
  parallelize_by = c("features", "repetitions"),
  ...
)
```

Arguments

`object`	A fitted model object (e.g., a randomForest object).
`...`	Additional optional arguments to be passed on to foreach (e.g., `.packages` or `.export`).
`feature_names`	Character string giving the names of the predictor variables (i.e., features) of interest. If `NULL` (the default) then they will be inferred from the `train` and `target` arguments (see below). It is good practice to always specify this argument.
`train`	A matrix-like R object (e.g., a data frame or matrix) containing the training data. If `NULL` (the default) then the internal `get_training_data()` function will be called to try and extract it automatically. It is good practice to always specify this argument.
`target`	Either a character string giving the name (or position) of the target column in `train` or, if `train` only contains feature columns, a vector containing the target values used to train `object`.
`metric`	Either a function or character string specifying the performance metric to use in computing model performance (e.g., RMSE for regression or accuracy for binary classification). If `metric` is a function, then it requires two arguments, `actual` and `predicted`, and should return a single, numeric value. Ideally, this should be the same metric that was used to train `object`. See `list_metrics()` for a list of built-in metrics.
`smaller_is_better`	Logical indicating whether or not a smaller value of `metric` is better. Default is `NULL`. Must be supplied if `metric` is a user-supplied function.
`type`	Character string specifying how to compare the baseline and permuted performance metrics. Current options are `"difference"` (the default) and `"ratio"`.
`nsim`	Integer specifying the number of Monte Carlo replications to perform. Default is 1. If `nsim` > 1, the results from each replication are simply averaged together (the standard deviation will also be returned).

`keep`	Logical indicating whether or not to keep the individual permutation scores for all `nsim` repetitions. If `TRUE` (the default) then the individual variable importance scores will be stored in an attribute called `"raw_scores"`. (Only used when `nsim > 1`.)
`sample_size`	Integer specifying the size of the random sample to use for each Monte Carlo repetition. Default is `NULL` (i.e., use all of the available training data). Cannot be specified with `sample_frac`. Can be used to reduce computation time with large data sets.
`sample_frac`	Proportion specifying the size of the random sample to use for each Monte Carlo repetition. Default is `NULL` (i.e., use all of the available training data). Cannot be specified with `sample_size`. Can be used to reduce computation time with large data sets.
`reference_class`	Deprecated, use `event_level` instead.
`event_level`	String specifying which factor level of `truth` to consider as the "event". Options are `"first"` (the default) or `"second"`. This argument is only applicable for binary classification when `metric` is one of `"roc_auc"`, `"pr_auc"`, or `"youden"`. This argument is passed on to the corresponding yardstick metric.
`pred_wrapper`	Prediction function that requires two arguments, `object` and `newdata`. The output of this function should be determined by the `metric` being used: • Regression - A numeric vector of predicted outcomes. • Binary classification - A vector of predicted class labels (e.g., if using misclassification error) or a vector of predicted class probabilities for the reference class (e.g., if using log loss or AUC). • Multiclass classification - A vector of predicted class labels (e.g., if using misclassification error) or a A matrix/data frame of predicted class probabilities for each class (e.g., if using log loss or AUC).
`verbose`	Logical indicating whether or not to print information during the construction of variable importance scores. Default is `FALSE`.
`parallel`	Logical indicating whether or not to run `vi_permute()` in parallel (using a backend provided by the foreach package). Default is `FALSE`. If `TRUE`, a foreach-compatible backend must be provided by must be provided. Note that `set.seed()` will not not work with foreach's parellelized for loops; for a workaround, see this solution.
`parallelize_by`	Character string specifying whether to parallelize across features (`parallelize_by = "features"`) or repetitions (`parallelize_by = "reps"`); the latter is only useful whenever `nsim > 1`. Default is `"features"`.

Value

A tidy data frame (i.e., a tibble object) with two columns:

- `Variable` - the corresponding feature name;
- `Importance` - the associated importance, computed as the average change in performance after a random permutation (or permutations, if `nsim > 1`) of the feature in question.

If `nsim > 1`, then an additional column (`StDev`) containing the standard deviation of the individual permutation scores for each feature is also returned; this helps assess the stability/variation of the individual permutation importance for each feature.

References

Brandon M. Greenwell and Bradley C. Boehmke, The R Journal (2020) 12:1, pages 343-366.

Examples

```
## Not run:
#
# Regression example
#

library(ranger)   # for fitting random forests

# Simulate data from Friedman 1 benchmark; only x1-x5 are important!
trn <- gen_friedman(500, seed = 101)  # ?vip::gen_friedman

# Prediction wrapper
pfun <- function(object, newdata) {
  # Needs to return vector of predictions from a ranger object; see
  # `ranger::predcit.ranger` for details on making predictions
  predict(object, data = newdata)$predictions
}

# Fit a (default) random forest
set.seed(0803) # for reproducibility
rfo <- ranger(y ~ ., data = trn)

# Compute permutation-based VI scores
set.seed(2021)  # for reproducibility
vis <- vi(rfo, method = "permute", target = "y", metric = "rsq",
          pred_wrapper = pfun, train = trn)
print(vis)

# Same as above, but using `vi_permute()` directly
set.seed(2021)  # for reproducibility
vi_permute(rfo, target = "y", metric = "rsq", pred_wrapper = pfun
           train = trn)

# Plot VI scores (could also replace `vi()` with `vip()` in above example)
vip(vis, include_type = TRUE)

# Mean absolute error
mae <- function(truth, estimate) {
  mean(abs(truth - estimate))
}

# Permutation-based VIP with user-defined MAE metric
set.seed(1101)  # for reproducibility
```

```
vi_permute(rfo, target = "y", metric = mae, smaller_is_better = TRUE,
           pred_wrapper = pfun, train = trn)

# Same as above, but using `yardstick` package instead of user-defined metric
set.seed(1101)  # for reproducibility
vi_permute(rfo, target = "y", metric = yardstick::mae_vec,
           smaller_is_better = TRUE, pred_wrapper = pfun, train = trn)

#
# Classification (binary) example
#

library(randomForest)  # another package for fitting random forests

# Complete (i.e., imputed version of titanic data); see `?vip::titanic_mice`
head(t1 <- titanic_mice[[1L]])
t1$pclass <- as.ordered(t1$pclass)  # makes more sense as an ordered factor

# Fit another (default) random forest
set.seed(2053)  # for reproducibility
(rfo2 <- randomForest(survived ~ ., data = t1))

# Define prediction wrapper for predicting class labels from a
# "randomForest" object
pfun_class <- function(object, newdata) {
  # Needs to return factor of classifications
  predict(object, newdata = newdata, type = "response")
}

# Sanity check
pfun_class(rfo2, newdata = head(t1))
##   1   2   3   4   5   6
## yes yes yes  no yes  no
## Levels: no yes

# Compute mean decrease in accuracy
set.seed(1359)  # for reproducibility
vi(rfo2,
   method = "permute",
   train = t1,
   target = "survived",
   metric = "accuracy",  # or pass in `yardstick::accuracy_vec` directly
   # smaller_is_better = FALSE,  # no need to set for built-in metrics
   pred_wrapper = pfun_class,
   nsim = 30  # use 30 repetitions
)
## # A tibble: 5 × 3
##   Variable Importance   StDev
##   <chr>         <dbl>   <dbl>
## 1 sex          0.228  0.0110
## 2 pclass       0.0825 0.00505
## 3 age          0.0721 0.00557
## 4 sibsp        0.0346 0.00430
## 5 parch        0.0183 0.00236

# Define prediction wrapper for predicting class probabilities from a
# "randomForest" object
pfun_prob <- function(object, newdata) {
  # Needs to return vector of class probabilities for event level of interest
  predict(object, newdata = newdata, type = "prob")[, "yes"]
}

# Sanity check
pfun_prob(rfo2, newdata = head(t1))  # estiated P(survived=yes | x)
##     1     2     3     4     5     6
## 0.990 0.864 0.486 0.282 0.630 0.078

# Compute mean increase in Brier score
set.seed(1411)  # for reproducibility
vi(rfo2,
   method = "permute",
   train = t1,
   target = "survived",
   metric = yardstick::brier_class_vec,  # or pass in `"brier"` directly
   smaller_is_better = FALSE,  # need to set when supplying a function
   pred_wrapper = pfun_prob,
   nsim = 30  # use 30 repetitions
)

## # A tibble: 5 × 3
## Variable Importance   StDev
##   <chr>         <dbl>   <dbl>
## 1 sex          0.210  0.00869
## 2 pclass       0.0992 0.00462
## 3 age          0.0970 0.00469
## 4 parch        0.0547 0.00273
## 5 sibsp        0.0422 0.00200

# Some metrics, like AUROC, treat one class as the "event" of interest. In
# such cases, it's important to make sure the event level (which typically
# defaults to which ever event class comes first in alphabetical order)
# matches the event class that corresponds to the prediction wrappers
# returned probabilities. To do this, you can (and should) set the
# `event_class` argument. For instance, our prediction wrapper specified
# `survived = "yes"` as the event of interest, but this is considered the
# second event:
levels(t1$survived)
## [1] "no"  "yes"

# So, we need to specify the second class as the event of interest via the
# `event_level` argument (otherwise, we would get the negative of the results
# we were hoping for; a telltale sign the event level and prediction wrapper
do not match)
set.seed(1413)  # for reproducibility
vi(rfo,
   method = "permute",
    train = t1,
    target = "survived",
    metric = "roc_auc",
    event_level = "second",  # use "yes" as class label/"event" of interest
    pred_wrapper = pfun_prob,
    nsim = 30  # use 30 repetitions
)

## # A tibble: 5 × 3
## Variable Importance   StDev
##   <chr>         <dbl>   <dbl>
## 1 sex          0.229  0.0137
## 2 pclass       0.0920 0.00533
## 3 age          0.0850 0.00477
## 4 sibsp        0.0283 0.00211
## 5 parch        0.0251 0.00351

## End(Not run)
```

`vi_shap` *SHAP-based variable importance*

Description

Compute SHAP-based VI scores for the predictors in a model. See details below.

Usage

```
vi_shap(object, ...)

## Default S3 method:
vi_shap(object, feature_names = NULL, train = NULL, ...)
```

Arguments

`object`	A fitted model object (e.g., a randomForest object).
`...`	Additional arguments to be passed on to `fastshap::explain()` (e.g., `nsim = 30`, `adjust = TRUE`, or avprediction wrapper via the `pred_wrapper` argument); see `?fastshap::explain` for details on these and other useful arguments.
`feature_names`	Character string giving the names of the predictor variables (i.e., features) of interest. If `NULL` (the default) then they will be inferred from the `train` and `target` arguments (see below). It is good practice to always specify this argument.
`train`	A matrix-like R object (e.g., a data frame or matrix) containing the training data. If `NULL` (the default) then the internal `get_training_data()` function will be called to try and extract it automatically. It is good practice to always specify this argument.

Details

This approach to computing VI scores is based on the mean absolute value of the SHAP values for each feature; see, for example, https://github.com/shap/shap and the references therein.

Strumbelj, E., and Kononenko, I. Explaining prediction models and individual predictions with feature contributions. Knowledge and information systems 41.3 (2014): 647-665.

Value

A tidy data frame (i.e., a tibble object) with two columns:

- `Variable` - the corresponding feature name;
- `Importance` - the associated importance, computed as the mean absolute Shapley value.

Examples

```
## Not run:
library(ggplot2)  # for theme_light() function
library(xgboost)

# Simulate training data
trn <- gen_friedman(500, sigma = 1, seed = 101)  # ?vip::gen_friedman

# Feature matrix
X <- data.matrix(subset(trn, select = -y))  # matrix of feature values

# Fit an XGBoost model; hyperparameters were tuned using 5-fold CV
set.seed(859)  # for reproducibility
bst <- xgboost(X, label = trn$y, nrounds = 338, max_depth = 3, eta = 0.1,
               verbose = 0)

# Construct VIP using "exact" SHAP values from XGBoost's internal Tree SHAP
# functionality
vip(bst, method = "shap", train = X, exact = TRUE, include_type = TRUE,
    geom = "point", horizontal = FALSE,
    aesthetics = list(color = "forestgreen", shape = 17, size = 5)) +
  theme_light()

# Use Monte-Carlo approach, which works for any model; requires prediction
# wrapper
pfun_prob <- function(object, newdata) {  # prediction wrapper
  # For Shapley explanations, this should ALWAYS return a numeric vector
  predict(object, newdata = newdata, type = "prob")[, "yes"]
}

# Compute Shapley-based VI scores
set.seed(853)  # for reproducibility
vi_shap(rfo, train = subset(t1, select = -survived), pred_wrapper = pfun_prob,
        nsim = 30)
## # A tibble: 5 × 2
## Variable Importance
##   <chr>         <dbl>
## 1 pclass       0.104
## 2 age          0.0649
## 3 sex          0.272
## 4 sibsp        0.0260
## 5 parch        0.0291

## End(Not run)
```

Index