Package 'varPro'

August 23, 2026

Version 3.2.0

Date 2026-08-22

Title Model-Independent Variable Selection via the Rule-Based Variable Priority

Author Min Lu [aut],
Aster K. Shear [aut],
Udaya B. Kogalur [aut, cre],
Hemant Ishwaran [aut]

Maintainer Udaya B. Kogalur <ubk@kogalur.com>

BugReports https://github.com/kogalur/varPro/issues/

Depends R (>= 4.3.0),

Imports randomForestSRC (>= 3.4.5), glmnet, parallel, foreach, gbm, BART, survival

Suggests mlbench, doMC, caret, MASS, igraph

SystemRequirements OpenMP

Description A new framework of variable selection, which instead of generating artificial covariates such as permutation importance and knockoffs, creates release rules to examine the affect on the response for each covariate where the conditional distribution of the response variable can be arbitrary and unknown.

License GPL (>= 3)

URL https://www.varprotools.org/ https://www.luminwin.net/ https://ishwaran.org/

NeedsCompilation yes

Repository CRAN

Date/Publication 2026-08-22 23:20:08 UTC

Contents

alzheimers 2
cv.varpro 4

gliomas 8
hrrecov 9
importance 12
isopro 15
ivarpro 19
outpro 25
partialpro 29
plot.ivarpro 36
plot.partialpro 40
predict.isopro 43
predict.ivarpro 45
predict.uvarpro 48
predict.varpro 50
uvarpro 52
varpro 58
varpro.news 65
varpro.strength 66

Index **68**

`alzheimers` *Alzheimer's Disease Dataset*

Description

Health, lifestyle, and clinical data for 2,149 individuals used for studying Alzheimer's Disease. Variables include demographics, cognitive assessments, medical conditions, and symptoms.

Usage

```
data(alzheimers)
```

Format

A data frame with 2,149 observations on the following variables:

Age: Age in years (60 to 90).

Gender: Gender (0 = Male, 1 = Female).

Ethnicity: Ethnicity (0 = Caucasian, 1 = African American, 2 = Asian, 3 = Other).

EducationLevel: Education level (0 = None, 1 = High School, 2 = Bachelor's, 3 = Higher).

BMI: Body Mass Index (15 to 40).

Smoking: Smoking status (0 = No, 1 = Yes).

AlcoholConsumption: Weekly alcohol consumption in units (0 to 20).

PhysicalActivity: Weekly physical activity in hours (0 to 10).

DietQuality: Diet quality score (0 to 10).

SleepQuality: Sleep quality score (4 to 10).

FamilyHistoryAlzheimers: Family history of Alzheimer's (0 = No, 1 = Yes).

CardiovascularDisease: Cardiovascular disease (0 = No, 1 = Yes).

Diabetes: Diabetes (0 = No, 1 = Yes).

Depression: Depression (0 = No, 1 = Yes).

HeadInjury: History of head injury (0 = No, 1 = Yes).

Hypertension: Hypertension (0 = No, 1 = Yes).

SystolicBP: Systolic blood pressure (90 to 180 mmHg).

DiastolicBP: Diastolic blood pressure (60 to 120 mmHg).

CholesterolTotal: Total cholesterol (150 to 300 mg/dL).

CholesterolLDL: LDL cholesterol (50 to 200 mg/dL).

CholesterolHDL: HDL cholesterol (20 to 100 mg/dL).

CholesterolTriglycerides: Triglycerides (50 to 400 mg/dL).

MMSE: Mini-Mental State Examination score (0 to 30). Lower is worse.

FunctionalAssessment: Functional score (0 to 10). Lower is worse.

MemoryComplaints: Memory complaints (0 = No, 1 = Yes).

BehavioralProblems: Behavioral problems (0 = No, 1 = Yes).

ADL: Activities of Daily Living score (0 to 10). Lower is worse.

Confusion: Presence of confusion (0 = No, 1 = Yes).

Disorientation: Presence of disorientation (0 = No, 1 = Yes).

PersonalityChanges: Presence of personality changes (0 = No, 1 = Yes).

DifficultyCompletingTasks: Difficulty completing tasks (0 = No, 1 = Yes).

Forgetfulness: Forgetfulness (0 = No, 1 = Yes).

Diagnosis: Alzheimer's diagnosis (No, Yes).

Details

This dataset is suitable for modeling Alzheimer's risk, performing exploratory analysis, and evaluating statistical and machine learning algorithms. All individuals are uniquely identified and evaluated on a standardized set of clinical and behavioral measures.

Source

Rabie El Kharoua (2024). Alzheimer's Disease Dataset. Available from Kaggle at https://www.kaggle.com/datasets/rabieelkharoua/alzheimers-disease-dataset

Examples

```
## load the data
data(alzheimers, package = "varPro")
o <- varpro(Diagnosis~.,alzheimers)
imp <- importance(o)
print(imp)
```

cv.varpro *Cross-Validated Cutoff Value for Variable Priority (VarPro)*

Description

Selects Cutoff Value for Variable Priority (VarPro).

Usage

```
cv.varpro(formula, data, nvar = 30, ntree = 150,
       local.std = TRUE, zcut = seq(0.1, 2, length = 50), nblocks = 10,
       split.weight = TRUE, split.weight.method = NULL, sparse = TRUE,
       nodesize = NULL, max.rules.tree = 150, max.tree = min(150, ntree),
       verbose = FALSE, seed = NULL, fast = FALSE, crps = FALSE, ...)
```

Arguments

formula	Model formula specifying the outcome and predictors.
data	Training data set (data frame).
nvar	Maximum number of variables to return.
ntree	Number of trees to grow.
local.std	Use locally standardized importance values?
zcut	Grid of positive cutoff values used for selecting top variables.
nblocks	Number of blocks (folds) for cross-validation.
split.weight	Use guided tree-splitting? Variables are selected for splitting with probability proportional to split-weights, obtained by default from a preliminary lasso+tree step.
split.weight.method	Character string or vector specifying how split-weights are generated. Defaults to lasso+tree.
sparse	Use sparse split-weights?
nodesize	Minimum terminal node size. If not specified, an internal function sets the value based on sample size and data dimension.
max.rules.tree	Maximum number of rules per tree.
max.tree	Maximum number of trees used for rule extraction.
verbose	Print verbose output?
seed	Seed for reproducibility.
fast	Use rfsrc.fast in place of rfsrc? May improve speed at the cost of accuracy.
crps	Use CRPS (continuous ranked probability score) instead of Harrell's C-index for evaluating survival performance? Applies only to survival families.
...	Additional arguments passed to varpro.

Details

Applies VarPro and then selects from a grid of cutoff values the cutoff value for identifying variables that minimizes out-of-sample performance (error rate) of a random forest where the forest is fit to the top variables identified by the given cutoff value.

Additionally, a "conservative" and "liberal" list of variables are returned using a one standard deviation rule. The conservative list comprises variables using the largest cutoff with error rate within one standard deviation from the optimal cutoff error rate, whereas the liberal list uses the smallest cutoff value with error rate within one standard deviation of the optimal cutoff error rate.

For class imbalanced settings (two class problems where relative frequency of labels is skewed towards one class) the code automatically switches to random forest quantile classification (RFQ; see O'Brien and Ishwaran, 2019) under the gmean (geometric mean) performance metric.

Value

Output containing importance values for the optimized cutoff value. A conservative and liberal list of variables is also returned.

Note that importance values are returned in terms of the original features and not their hot-encodings. For importance in terms of hot-encodings, use the built-in wrapper `get.vimp` (see example below).

Author(s)

Min Lu and Hemant Ishwaran

References

Lu, M. and Ishwaran, H. (2024). Model-independent variable selection via the rule-based variable priority. arXiv e-prints, pp.arXiv-2409.

O'Brien R. and Ishwaran H. (2019). A random forests quantile classifier for class imbalanced data. *Pattern Recognition*, 90, 232-249.

See Also

`importance.varpro` `uvarpro` `varpro`

Examples

```
## ------------------------------------------------------------
## van de Vijver microarray breast cancer survival data
## high dimensional example
## ------------------------------------------------------------

data(vdv, package = "randomForestSRC")
o <- cv.varpro(Surv(Time, Censoring) ~ ., vdv)
print(o)

## ------------------------------------------------------------
## boston housing
## ------------------------------------------------------------
```

```
data(BostonHousing, package = "mlbench")
print(cv.varpro(medv~., BostonHousing))

## ------------------------------------------------------------
## boston housing - original/hot-encoded vimp
## ------------------------------------------------------------

## load the data
data(BostonHousing, package = "mlbench")

## convert some of the features to factors
Boston <- BostonHousing
Boston$zn <- factor(Boston$zn)
Boston$chas <- factor(Boston$chas)
Boston$lstat <- factor(round(0.2 * Boston$lstat))
Boston$nox <- factor(round(20 * Boston$nox))
Boston$rm <- factor(round(Boston$rm))

## make cv call
o <-cv.varpro(medv~., Boston)
print(o)

## importance original variables (default)
print(get.orgvimp(o, pretty = FALSE))

## importance for hot-encoded variables
print(get.vimp(o, pretty = FALSE))

## ------------------------------------------------------------
## multivariate regression example: boston housing
## vimp is collapsed across the outcomes
## ------------------------------------------------------------

data(BostonHousing, package = "mlbench")
print(cv.varpro(cbind(lstat, nox) ~., BostonHousing))

## ------------------------------------------------------------
## iris
## ------------------------------------------------------------

print(cv.varpro(Species~., iris))

## ------------------------------------------------------------
## friedman 1
## ------------------------------------------------------------

print(cv.varpro(y~., data.frame(mlbench::mlbench.friedman1(1000))))

##------------------------------------------------------------
##  class imbalanced problem
##
## - simulation example using the caret R-package
## - creates imbalanced data by randomly sampling the class 1 values
```

```
##
##------------------------------------------------------------

if (library("caret", logical.return = TRUE)) {

  ## experimental settings
  n <- 5000
  q <- 20
  ir <- 6
  f <- as.formula(Class ~ .)

  ## simulate the data, create minority class data
  d <- twoClassSim(n, linearVars = 15, noiseVars = q)
  d$Class <- factor(as.numeric(d$Class) - 1)
  idx.0 <- which(d$Class == 0)
  idx.1 <- sample(which(d$Class == 1), sum(d$Class == 1) / ir , replace = FALSE)
  d <- d[c(idx.0,idx.1),, drop = FALSE]
  d <- d[sample(1:nrow(d)), ]

  ## cv.varpro call
  print(cv.varpro(f, d))

}


## ------------------------------------------------------------
## pbc survival with rmst vector
## note that vimp is collapsed across the rmst values
## similar to mv-regression
## ------------------------------------------------------------

data(pbc, package = "randomForestSRC")
print(cv.varpro(Surv(days, status)~., pbc, rmst = c(500, 1000)))


## ------------------------------------------------------------
## peak VO2 with cutoff selected using fast option
## (a) C-index (default) (b) CRPS performance metric
## ------------------------------------------------------------

data(peakVO2, package = "randomForestSRC")
f <- as.formula(Surv(ttodead, died)~.)

## Harrel's C-index (default)
print(cv.varpro(f, peakVO2, ntree = 100, fast = TRUE))

## Harrel's C-index with smaller bootstrap
print(cv.varpro(f, peakVO2, ntree = 100, fast = TRUE, sampsize = 100))

## CRPS with smaller bootstrap
print(cv.varpro(f, peakVO2, crps = TRUE, ntree = 100, fast = TRUE, sampsize = 100))

## ------------------------------------------------------------
```

```
## largish data set: illustrates various options to speed up calculations
## ------------------------------------------------------------

## roughly impute the data
data(housing, package = "randomForestSRC")
housing2 <- roughfix(housing)

## use bigger nodesize
print(cv.varpro(SalePrice~., housing2, fast = TRUE, ntree = 50, nodesize = 150))

## use smaller bootstrap
print(cv.varpro(SalePrice~., housing2, fast = TRUE, ntree = 50, nodesize = 150, sampsize = 250))
```

---

gliomas *Diffuse Adult Glioma*

Description

Subset of the data used in Ceccarelli et al. (2016) for molecular profiling of adult diffuse gliomas. As part of the analysis, the authors developed a supervised analysis using DNA methylation data. Their original dataset was collected from a core set of 25,978 CpG probes which was reduced to eliminate sites that were methylated. This reduced set of 1206 probes from 880 tissues makes up part of the features of this data. Also included are clinical data and other molecular data collected for the samples. The outcome is a supervised class label developed in the study with labels: Classic-like, Codel, G-CIMP-high, G-CIMP-low, LGm6-GBM, Mesenchymal-like and PA-like.

References

Ceccarelli, M., Barthel, F.P., Malta, T.M., Sabedot, T.S., Salama, S.R., Murray, B.A., Morozova, O., Newton, Y., Radenbaugh, A., Pagnotta, S.M. et al. (2016). Molecular profiling reveals biologically discrete subsets and pathways of progression in diffuse glioma. Cell, 164, 550-563.

Examples

```
data(glioma, package = "varPro")
o <- varpro(y~., glioma, nodesize=2, max.tree=250)
imp <- importance(o)
print(head(imp$unconditional))
print(imp$conditional.z)
```

hrrecov *Exercise Heart Rate Recovery and Mortality*

Description

A survival dataset from a cohort of patients referred for symptom-limited exercise testing, originally used to study exercise heart rate recovery as a predictor of all-cause mortality.

Usage

```
data(hrrecov)
```

Format

A data frame with 23701 observations on 83 variables.

`ttodead` Follow-up time to death or censoring, in years (rounded to 2 decimals).

`died` Event indicator (1 = death, 0 = right-censored).

`hrrecov` Heart rate recovery (beats/min): peak heart rate minus heart rate 1 minute into recovery.

`lowrec` Indicator for low/abnormal heart rate recovery.

`lowcri` Indicator for low chronotropic response index/chronotropic incompetence.

`peak_hr` Peak heart rate during exercise (beats/min).

`peak_met` Peak workload achieved (metabolic equivalents, METs).

`fitness` Fitness category (ordinal 1-5; coding as in source data).

`heart_ra` Resting heart rate prior to testing (beats/min).

`sbprest` Resting systolic blood pressure (mm Hg).

`dbprest` Resting diastolic blood pressure (mm Hg).

`age` Age at exercise test (years).

`gender` Sex indicator (1=men).

`race` Race category (integer code; coding as in source data).

`black` Indicator for Black race (0/1).

`height` Height (m).

`weight` Weight (kg).

`bmi` Body mass index (kg/m^2).

`bsa` Body surface area.

`wtht` Weight-to-height ratio (weight/height; kg/m).

`obese` Indicator for obesity (0/1).

`priorcad` Known or suspected coronary artery disease prior to test (0/1).

`mihist` History of myocardial infarction (0/1).

`pcabg` Prior coronary artery bypass grafting (CABG) (0/1).

`ppci` Prior percutaneous coronary intervention (PCI) (0/1).

`cva` History of cerebrovascular accident/stroke (0/1).

`tia` History of transient ischemic attack (0/1).

`pvd` Peripheral vascular disease (0/1).

`diabetes` History of diabetes mellitus (0/1).

`insulin` Insulin therapy (0/1).

`htn` History of hypertension (0/1).

`htnrx` Antihypertensive treatment (0/1).

`hichol` History of high cholesterol/hyperlipidemia (0/1).

`smknow` Smoking history/status indicator.

`asthma` History of asthma (0/1).

`copd` History of chronic obstructive pulmonary disease (0/1).

`esrd` End-stage renal disease (0/1).

`lv_dysf` Left ventricular dysfunction (0/1).

`ecgmi` ECG evidence of myocardial infarction (0/1).

`ecglvh` ECG evidence of left ventricular hypertrophy (0/1).

`lbbb` Left bundle branch block (0/1).

`rbbb` Right bundle branch block (0/1).

`reststt` Resting ST-segment abnormality (0/1).

`stnond` Non-diagnostic ST-segment response (0/1).

`stabn` ST-segment abnormality during testing (0/1).

`stabnb` ST-segment abnormality subtype/flag.

`stabnv` ST-segment abnormality subtype/flag.

`rtachy` Tachycardia indicator.

`typcp` Typical chest pain indicator.

`ntangina` Angina history/symptom indicator.

`ttangina` Angina during treadmill test (0/1).

`ttclaud` Claudication during treadmill test (0/1).

`image` Exercise test performed with imaging (0/1).

`acei` Angiotensin-converting enzyme (ACE) inhibitor use (0/1).

`aspirin` Aspirin use (0/1).

`betablok` Beta-blocker use (0/1).

`dilver` Diltiazem/verapamil use (0/1).

`nifed` Nifedipine use (0/1).

`diuretic` Diuretic use (0/1).

`lipidrx` Lipid-lowering therapy use (0/1).

`nitrates` Nitrate therapy use (0/1).

`bdilat` Bronchodilator use (0/1).

`rs_ami` Reason for referral/testing: acute myocardial infarction (0/1).

`rs_mi` Reason for referral/testing: myocardial infarction (0/1).

`rs_ptca` Reason for referral/testing: PTCA/angioplasty (0/1).

`rs_cabg` Reason for referral/testing: CABG (0/1).

`rs_arrth` Reason for referral/testing: arrhythmia (0/1).

`rs_htran` Reason for referral/testing: heart transplant (0/1).

`bpvc_rst` Bigeminal premature ventricular contractions at rest (0/1).

`bpvc_ex` Bigeminal premature ventricular contractions during exercise (0/1).

`bpvc_rec` Bigeminal premature ventricular contractions during recovery (0/1).

`fpvc_rst` Frequent premature ventricular contractions at rest (0/1).

`fpvc_ex` Frequent premature ventricular contractions during exercise (0/1).

`fpvc_rec` Frequent premature ventricular contractions during recovery (0/1).

`nsvt_rst` Non-sustained ventricular tachycardia at rest (0/1).

`nsvt_ex` Non-sustained ventricular tachycardia during exercise (0/1).

`nsvt_rec` Non-sustained ventricular tachycardia during recovery (0/1).

`vtrp_rst` Ventricular triplets at rest (0/1).

`vtrp_ex` Ventricular triplets during exercise (0/1).

`vtrp_rec` Ventricular triplets during recovery (0/1).

`hbm2_rec` Heart rate-related recovery measure at 2 minutes.

`exec` Indicator variable (0/1); coding as in source data.

`aso` Indicator variable (0/1); coding as in source data.

Details

Heart rate recovery (`hrrecov`) is defined as peak heart rate minus the heart rate measured 1 minute into recovery. The survival outcome is all-cause mortality with right-censoring. Unless otherwise noted, indicator variables are coded 0/1 (0 = no/absent, 1 = yes/present).

References

Ishwaran, H., Blackstone, E. H., Pothier, C. E. and Lauer, M. S. (2004). Relative risk forests for exercise heart rate recovery as a predictor of mortality. *Journal of the American Statistical Association*, 99, 591-600.

Examples

```
data(hrrecov)
vp <- varpro(Surv(ttodead, died)~., hrrecov, ntree=50, split.weight=FALSE)
ivp <- ivarpro(vp)
plot(ivp, var = "peak_met", col.var = "fitness", size.var = "peak_hr",
       col.legend.n = 7, smooth.n = 7, x.dist = "auto")
```

`importance` *Calculate Importance for VarPro and UVarPro Objects*

Description

Calculates variable importance from compatible objects such as `varpro` and `uvarpro`.

Usage

```
importance(x, ...)

## S3 method for class 'varpro'
importance(x, local.std = TRUE, y.external = NULL,
  cutoff = 0.79, trim = 0.1, plot.it = FALSE, conf = TRUE, sort = TRUE,
  ylab = if (conf) "Importance" else "Standardized Importance",
  max.rules.tree, max.tree,
  ...)

## S3 method for class 'uvarpro'
importance(x, local.std = FALSE, y.external = NULL,
  cutoff = 0.79, trim = 0.1, plot.it = FALSE, conf = TRUE, sort = TRUE,
  ylab = if (conf) "Importance" else "Standardized Importance",
  max.rules.tree, max.tree,
  ...)

## S3 method for class 'rhf'
importance(x, local.std = TRUE, y.external = NULL,
  cutoff = 0.79, trim = 0.1, plot.it = FALSE, conf = TRUE, sort = TRUE,
  ylab = if (conf) "Importance" else "Standardized Importance",
  max.rules.tree, max.tree,
  ...)
```

Arguments

`x`	A `varpro`, `uvarpro` or `rhf` object.
`local.std`	Logical. If `TRUE`, uses locally standardized importance values. Ignored for `uvarpro` objects.
`y.external`	Optional user-supplied response vector. Must match the expected dimension and outcome family. Ignored for `uvarpro` objects.
`cutoff`	Threshold used to highlight significant variables in the importance plot. Applies only when `plot.it = TRUE`.
`trim`	Windsorization trim value used to robustify the mean and standard deviation calculations.

plot.it	Logical. If TRUE, generates a plot of importance values.
conf	Logical. If TRUE, displays importance values with standard errors as a boxplot (providing an informal confidence region). If FALSE, plots standardized importance values.
sort	Logical. If TRUE, sorts results in decreasing order of importance.
ylab	Character string specifying the y-axis label.
max.rules.tree	Optional. Maximum number of rules per tree. Defaults to the value stored in the fitted object if unspecified.
max.tree	Optional. Maximum number of trees used for rule extraction. Defaults to the value from the fitted object if unspecified.
...	Additional arguments passed to internal methods.

Details

This page documents the public importance() generic together with the methods for varpro and uvarpro objects.

The supervised varpro method calculates standardized importance values for identifying and ranking variables. Optionally, graphical output is provided, including confidence-style boxplots.

Value

Invisibly, a table summarizing the results. Contains mean importance mean, the standard deviation std, and standardized importance z.

For classification, conditional z tables are additionally provided, where the z standardized importance values are conditional on the class label.

See cv.varpro for a data-driven cross-validation method for selecting the cutoff value, cutoff, in supervised varpro analyses.

Author(s)

Min Lu and Hemant Ishwaran

References

Lu, M. and Ishwaran, H., (2024). Model-independent variable selection via the rule-based variable priority. arXiv e-prints, pp.arXiv-2409.

See Also

cv.varpro varpro uvarpro

Examples

```
## ------------------------------------------------------------
## toy example - needed to pass CRAN test
## ------------------------------------------------------------
```

```
## mtcars regression
o <- varpro(mpg ~ ., mtcars, ntree = 1)
imp <- importance(o, local.std = FALSE)
print(imp)



## ------------------------------------------------------------
## iris example
## ------------------------------------------------------------

## apply varpro to the iris data
o <- varpro(Species ~ ., iris, max.tree = 5)

## print/plot the results
imp <- importance(o, plot.it = TRUE)
print(imp)

## ------------------------------------------------------------
## boston housing: regression
## ------------------------------------------------------------

data(BostonHousing, package = "mlbench")

## call varpro
o <- varpro(medv~., BostonHousing)

## extract importance values
imp <- importance(o)
print(imp)

## plot the results
imp <- importance(o, plot.it = TRUE)
print(imp)



## ------------------------------------------------------------
## illustrates y-external: regression example
## ------------------------------------------------------------

## friedman1 - standard application of varpro
d <- data.frame(mlbench::mlbench.friedman1(250),noise=matrix(runif(250*10,-1,1),250))
o <- varpro(y~.,d)
print(importance(o))

## importance using external rf predictor
print(importance(o,y.external=randomForestSRC::rfsrc(y~.,d)$predicted.oob))

## importance using external lm predictor
print(importance(o,y.external=lm(y~.,d)$fitted))
```

```
## importance using external randomized predictor
print(importance(o,y.external=sample(o$y)))

## ------------------------------------------------------------
## illustrates y-external: classification example
## ------------------------------------------------------------

## iris - standard application of varpro
o <- varpro(Species~.,iris)
print(importance(o))

## importance using  external rf predictor
print(importance(o,y.external=randomForestSRC::rfsrc(Species~.,iris)$class.oob))

## importance using  external randomized predictor
print(importance(o,y.external=sample(o$y)))

## ------------------------------------------------------------
## illustrates y-external: survival
## ------------------------------------------------------------
data(pbc, package = "randomForestSRC")
o <- varpro(Surv(days, status)~., pbc)
print(importance(o))

## importance using  external rsf predictor
print(importance(o,y.external=randomForestSRC::rfsrc(Surv(days, status)~., pbc)$predicted.oob))

## importance using  external randomized predictor
print(importance(o,y.external=sample(o$y)))
```

`isopro` *Identify Anomalous Data*

Description

Use isolation forests to identify rare/anomalous data.

Usage

```
isopro(object,
       method = c("unsupv", "rnd", "auto"),
       sampsize = function(x){min(2^6, .632 * x)},
       ntree = 500, nodesize = 1,
       formula = NULL, data = NULL, ...)
```

Arguments

object	varpro object returned from a previous call.
method	Isolation forest method. Options are "unsupv" (unsupervised analysis, default), "rnd" (pure random splitting), and "auto" (auto-encoder, a type of multivariate forest).
sampsize	Function or numeric value specifying the sample size used for constructing each tree. Sampling is without replacement.
ntree	Number of trees to grow.
nodesize	Minimum terminal node size.
formula	Formula used for supervised isolation forest. Ignored if object is provided.
data	Data frame used to fit the isolation forest. Ignored if object is provided.
...	Additional arguments passed to rfsrc.

Details

Isolation Forest (Liu et al., 2008) is a random forest-based method for detecting anomalous observations. In its original form, trees are constructed using pure random splits, with each tree built from a small subsample of the data, typically much smaller than the standard 0.632 fraction used in random forests. The idea is that anomalous or rare observations are more likely to be isolated early, requiring fewer splits to reach terminal nodes. Thus, observations with relatively small depth values (i.e., shallow nodes) are considered anomalies.

There are several ways to apply the method:

- The default approach is to supply a formula and data to build a supervised isolation forest. If only data is provided (i.e., no response), an unsupervised analysis is performed. In this case, the method option is used to specify the type of isolation forest (e.g., "unsupv", "rnd", or "auto").
- If both a formula and data are provided, a supervised model is fit. In this case, method is ignored. While less conventional, this approach may be useful in certain applications.
- Alternatively, a varpro object may be supplied, but other configurations are also supported. In this setting, isolation forest is applied to the reduced feature matrix extracted from theobject. This is similar to using the data option alone but with the advantage of prior dimension reduction.

Users are encouraged to experiment with the choice of method, as the original isolation forest ("rnd") performs well in many scenarios but can be improved upon in others. For example, in some cases, "unsupv" or "auto" may yield better detection performance.

In terms of computational cost, "rnd" is the fastest, followed by "unsupv". The slowest is "auto", which is best suited for low-dimensional settings.

Value

Trained isolation forest and anomaly scores.

Author(s)

Min Lu and Hemant Ishwaran

References

Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. (2008). Isolation forest. 2008 Eighth IEEE International Conference on Data Mining. IEEE.

Ishwaran H. (2025). Multivariate Statistics: Classical Foundations and Modern Machine Learning, CRC (Chapman and Hall), in press.

See Also

predict.isopro uvarpro varpro

Examples

```
## ------------------------------------------------------------
##
## satellite data: convert some of the classes to "outliers"
## unsupervised isopro analysis
##
## ------------------------------------------------------------

## load data, make three of the classes into outliers
data(Satellite, package = "mlbench")
is.outlier <- is.element(Satellite$classes,
          c("damp grey soil", "cotton crop", "vegetation stubble"))

## remove class labels, make unsupervised data
x <- Satellite[, names(Satellite)[names(Satellite) != "classes"]]

## isopro calls
i.rnd <- isopro(data=x, method = "rnd", sampsize=32)
i.uns <- isopro(data=x, method = "unsupv", sampsize=32)
i.aut <- isopro(data=x, method = "auto", sampsize=32)

## AUC and precision recall (computed using true class label information)
perf <- cbind(get.iso.performance(is.outlier,i.rnd$howbad),
              get.iso.performance(is.outlier,i.uns$howbad),
              get.iso.performance(is.outlier,i.aut$howbad))
colnames(perf) <- c("rnd", "unsupv", "auto")
print(perf)


## ------------------------------------------------------------
##
## boston housing analysis
## isopro analysis using a previous VarPro (supervised) object
##
## ------------------------------------------------------------

data(BostonHousing, package = "mlbench")
```

```
## call varpro first and then isopro
o <- varpro(medv~., BostonHousing)
o.iso <- isopro(o)

## identify data with extreme percentiles
print(BostonHousing[o.iso$howbad <= quantile(o.iso$howbad, .01),])

## ------------------------------------------------------------
##
## boston housing analysis
## supervised isopro analysis - direct call using formula/data
##
## ------------------------------------------------------------

data(BostonHousing, package = "mlbench")

## direct approach uses formula and data options
o.iso <- isopro(formula=medv~., data=BostonHousing)

## identify data with extreme percentiles
print(BostonHousing[o.iso$howbad <= quantile(o.iso$howbad, .01),])


## ------------------------------------------------------------
##
## monte carlo experiment to study different methods
## unsupervised isopro analysis
##
## ------------------------------------------------------------

## monte carlo parameters
nrep <- 25
n <- 1000

## simulation function
twodimsim <- function(n=1000) {
  cluster1 <- data.frame(
    x = rnorm(n, -1, .4),
    y = rnorm(n, -1, .2)
  )
  cluster2 <- data.frame(
    x = rnorm(n, +1, .2),
    y = rnorm(n, +1, .4)
  )
  outlier <- data.frame(
    x = -1,
    y =  1
  )
  x <- data.frame(rbind(cluster1, cluster2, outlier))
  is.outlier <- c(rep(FALSE, 2 * n), TRUE)
  list(x=x, is.outlier=is.outlier)
}
```

```
## monte carlo loop
hbad <- do.call(rbind, lapply(1:nrep, function(b) {
  cat("iteration:", b, "\n")
  ## draw the data
  sim0 <- twodimsim(n)
  x <- sim0$x
  is.outlier <- sim0$is.outlier
  ## iso pro calls
  i.rnd <- isopro(data=x, method = "rnd")
  i.uns <- isopro(data=x, method = "unsupv")
  i.aut <- isopro(data=x, method = "auto")
  ## save results
  c(tail(i.rnd$howbad,1),
    tail(i.uns$howbad,1),
    tail(i.aut$howbad,1))
}))


## compare performance
colnames(hbad) <- c("rnd", "unsupv", "auto")
print(summary(hbad))
boxplot(hbad,col="blue",ylab="outlier percentile value")
```

---

ivarpro *Individual Variable Priority (iVarPro): Case-Specific Variable Importance*

Description

Individual Variable Priority (iVarPro) computes case-specific (individual-level) variable importance scores. For each observation in the data and for each predictor identified by the VarPro analysis, iVarPro returns a local gradient-based priority measure that quantifies how sensitive that case's prediction is to changes in that variable.

Usage

```
ivarpro(object,
        adaptive = TRUE,
        cut = NULL,
        cut.max = 1,
        ncut = 51,
        nmin = 20, nmax = 150,
        y.external = NULL,
        noise.na = TRUE,
        max.rules.tree = NULL,
        max.tree = NULL,
        use.loo = TRUE,
```

```
        use.abs = FALSE,
        path.store.membership = TRUE,
        save.data = TRUE,
        save.model = TRUE,
        scale = c("local", "global", "none"))
```

Arguments

object	varpro object from a previous call to varpro, or a rfsrc object.
adaptive	Logical. If FALSE and cut is not supplied, the cut grid is constructed as seq(0, cut.max, length.out = ncut). If TRUE (default) and cut is not supplied, a data-adaptive upper bound for the neighborhood scale is computed from the sample size using a simple bandwidth-style rule-of-thumb, and cut is constructed as a sequence from 0 to this data-adaptive maximum (subject to cut.max). This provides a convenient way to automatically sharpen the local neighborhood for case-specific gradients when the sample size is moderate to large.
cut	Optional user-supplied sequence of λ values used to relax the constraint region in the local linear regression model. For continuous release variables, each value in cut is calibrated so that cut = 1 corresponds to one standard deviation of the release coordinate. If cut is supplied, it is used as-is and the arguments cut.max, ncut, and adaptive are ignored. For binary or one-hot encoded release variables, the full released region is used and cut does not control neighborhood size.
cut.max	Maximum value of the λ grid used to define the local neighborhood for continuous release variables when cut is not supplied. By default, cut is constructed as seq(0, cut.max, length.out = ncut) (or up to a data-adaptive value if adaptive = TRUE). Smaller values of cut.max yield more local, sharper case-specific gradients, while larger values yield smoother, more global behavior.
ncut	Length of the cut grid when cut is not supplied. The grid is constructed as seq(0, cut.max, length.out = ncut) (or up to an adaptively chosen maximum if adaptive = TRUE).
nmin	Minimum number of observations required for fitting a local linear model.
nmax	Maximum number of observations allowed for fitting a local linear model. Internally, nmax is capped at 10% of the sample size.
y.external	Optional user-supplied response vector or matrix to use as the dependent variable in the local linear regression. Must have the same number of rows as the feature matrix and match the dimension and type expected for the outcome family.
noise.na	Logical. If TRUE (default), gradients for noisy or non-signal variables are set to NA; if FALSE, they are set to zero.
max.rules.tree	Maximum number of rules per tree. If unspecified, the value from the varpro object is used, while for rfsrc objects, a default value is used.
max.tree	Maximum number of trees used to extract rules. If unspecified, the value from the varpro object is used, while for rfsrc objects, a default value is used.

use.loo — Logical. If TRUE (default), leave-one-out cross-validation is used to select the best neighborhood size (i.e., the best value in cut) for each rule and release variable. If FALSE, the neighborhood is chosen to use the largest available sample that satisfies nmin and nmax.

use.abs — Use the absolute gradient for individual importance? Default is FALSE which uses the actual gradient.

path.store.membership — Store the rule membership indices (OOB case IDs) in the returned object for later ladder/band calculations? Setting FALSE can substantially reduce memory usage when the number of rules is large, but disables ladder-based bands in plot.ivarpro() and prevents ivarpro_band() from being used. Default is TRUE.

save.data — Save the x and y data (default is TRUE)? Used for downstream plots.

save.model — Save the original object (default is TRUE)? Used for predict.ivarpro to obtain gradient values of test data.

scale — Character. Controls how the local slope is scaled when forming the rule-level gradient. "local" (default) uses the neighborhood-based standardization described in Lu and Ishwaran(2025). "global" uses a fixed global scale (the overall standard deviation of the predictor in the training data) rather than a neighborhood-specific scale. "none" returns the unscaled slope in the original predictor units (or the unscaled finite difference for binary 0/1 release variables).

Details

Understanding individual-level (case-specific) variable importance is important in applications where decisions are made at the level of a single person, unit, or record. A predictor may have only a modest average effect, yet be highly influential for certain cases, or the direction of its effect may differ across individuals.

The VarPro framework summarizes population-level importance by defining feature-space regions using rule-based splitting and computing importance using only observed data. iVarPro (Lu and Ishwaran, 2025) extends this idea to the individual level by quantifying how sensitive each case's prediction is to small changes in a predictor identified by the VarPro rule set.

For each VarPro rule, iVarPro considers the corresponding rule-defined region and then *releases* the rule along the rule's release coordinate. Intuitively, releasing a region means keeping the other rule constraints in place while allowing additional variation in the released variable, which provides the information needed to estimate a local directional effect. A simple local linear regression is then fit on this released region, and the resulting slope is used as a local, gradient-based priority score. Case-specific scores are obtained by aggregating the relevant rule-level gradients over the rules that apply to each case.

Scaling / standardization. By default, iVarPro reports a *standardized* local gradient: the local linear regression is fit using a standardized release coordinate so that cut = 1 corresponds to one standard deviation of the release variable within the released region. This yields a dimensionless effect-size interpretation and helps make iVarPro values comparable across predictors with different units and variability (see Section~2.6 of Lu and Ishwaran, 2025). The argument scale controls how this scaling is applied: "local" uses the neighborhood-specific scale (default), "global" uses a fixed global scale for the predictor (overall SD in the training data), and "none" returns

the unscaled slope in the original predictor units (or the unscaled finite-difference for binary 0/1 release variables). When using iVarPro-derived gradients for interaction screening, comparing results across `scale` settings can help distinguish structural interactions in the prediction surface from scale-induced modulation due to neighborhood-dependent standardization.

Neighborhood size and `cut.max`. For continuous release variables, the size of the local neighborhood used for slope estimation is controlled by `cut` (constructed from `cut.max` and `ncut` when not supplied). Smaller neighborhoods produce more local behavior and can better reflect sharp changes, while larger neighborhoods produce smoother, more global behavior. When `use.loo = TRUE`, the neighborhood size is chosen in a data-driven way using a leave-one-out criterion; when `use.loo = FALSE`, the choice is based on meeting the requested sample-size bounds `nmin` and `nmax`. When `adaptive = TRUE` and `cut` is not supplied, an additional sample-size based rule is used to limit the maximum neighborhood scale (subject to `cut.max`).

For binary or one-hot encoded release variables, iVarPro interprets the local effect as a scaled finite difference between the two levels (0 and 1), conditional on the other rule constraints; in this case `cut` does not control the neighborhood along the binary coordinate.

Cut.max ladder (neighborhood sensitivity). Because the choice of neighborhood scale can affect the estimated local gradients, iVarPro also records a *ladder* of rule-level gradient estimates across the candidate neighborhood sizes defined by the `cut` grid. These ladder values can be summarized (e.g., ranges or quantiles) and used to visualize how sensitive case-specific gradients are to the neighborhood choice, without repeatedly refitting iVarPro for many different `cut.max` values. Ladder-based case summaries require rule membership information; set `path.store.membership = TRUE` to enable ladder bands and related summaries, or leave it `FALSE` to reduce memory usage when the number of rules is very large. See examples below.

Settings that are currently handled. The flexibility of this framework makes it suitable for quantifying case-specific variable importance in regression, classification, and survival settings. Currently, multivariate forests are not handled.

Value

For univariate outcomes (and two-class classification treated as a single score), a numeric `data.frame` of dimension $n \times p$ containing case-specific (individual-level) variable priority values, where n is the number of observations and p is the number of predictors in `object$xvar.names`.

- Each row corresponds to a case (observation) in the original data.
- Each column corresponds to a predictor variable in `object$xvar.names`.

The entry in row i and column j is the iVarPro importance score for variable j for case i, measuring the local sensitivity of that case's prediction to changes in that variable. Predictors that are never used as release variables in the VarPro rule set may appear with constant `NA` values (when `noise.na = TRUE`) or constant zero values (when `noise.na = FALSE`).

Ladder/path information. The returned object carries an attribute `"ivarpro.path"` containing additional information used for ladder-based summaries and plotting. In particular:

`cut` The full `cut` grid used to evaluate candidate local neighborhoods.

`cut.ladder` The interior values of `cut` (excluding the endpoints) used for the cut.max ladder path.

`scale` The scaling option used to compute rule-level gradients (`"local"`, `"global"`, or `"none"`).

`rule.imp.ladder` A numeric matrix of dimension $R \times L$ storing rule-level gradients selected under each ladder truncation, where R is the number of retained rules and $L = length(cut.ladder)$.

`rule.variable` Integer vector of length R giving the release-variable index for each retained rule.

`oobMembership` Optional list (length R) giving the OOB comparison/released membership indices for each retained rule; included only when `path.store.membership = TRUE`.

Additional tuning flags and rule metadata (e.g., `use.loo`, `adaptive`, and tree/branch identifiers) may also be included for diagnostics.

Author(s)

Min Lu and Hemant Ishwaran

References

Lu, M. and Ishwaran, H. (2025). Individual variable priority: a model-independent local gradient method for variable importance. *Artificial Intelligence Review*, 58:407.

See Also

`varpro` `plot.ivarpro`

Examples

```
## ------------------------------------------------------------
##
## survival example with shap-like plot
##
## ------------------------------------------------------------

data(peakVO2, package = "randomForestSRC")
o <- varpro(Surv(ttodead, died)~., peakVO2, ntree = 50)

## canonical standard analysis
imp1 <- ivarpro(o)
shap.ivarpro(imp1)

## non-adaptive analysis
imp2 <- ivarpro(o, adaptive = FALSE)
shap.ivarpro(imp2)

## non-adaptive using a small cut.max
imp3 <- ivarpro(o, cut.max = 0.5, adaptive = FALSE)
shap.ivarpro(imp3)


## ------------------------------------------------------------
##
## synthetic regression example with plot
##
```

```
## ------------------------------------------------------------

## true regression function
true.function <- function(which.simulation) {
  if (which.simulation == 1) {
    function(x1, x2) { 1 * (x2 <= .25) +
      15 * x2 * (x1 <= .5 & x2 > .25) +
      (7 * x1 + 7 * x2) * (x1 > .5 & x2 > .25) }
  }
  else if (which.simulation == 2) {
    function(x1, x2) { r <- x1^2 + x2^2; 5 * r * (r <= .5) }
  }
  else {
    function(x1, x2) { 6 * x1 * x2 }
  }
}

## simulation function
simfunction <- function(n = 1000, true.function, d = 20, sd = 1) {
  d <- max(2, d)
  X <- matrix(runif(n * d, 0, 1), ncol = d)
  dta <- data.frame(list(
    x = X,
    y = true.function(X[, 1], X[, 2]) + rnorm(n, sd = sd)
  ))
  colnames(dta)[1:d] <- paste("x", 1:d, sep = "")
  dta
}

## simulate the data
which.simulation <- 1
df <- simfunction(n = 500, true.function(which.simulation))

## varpro analysis
vp <- varpro(y ~ ., df)

## ivarpro analysis
imp <- ivarpro(vp)
## ivarpro analysis using a fixed global scale (optional)
imp.global <- ivarpro(vp, scale = "global")

## ivarpro analysis returning unscaled slopes (optional)
imp.none <- ivarpro(vp, scale = "none")


## plot of x2
plot(imp, var="x2")

## plot of x2 without ladder band
plot(imp, var="x2", ladder=FALSE)

## optional: use only a subset of ladder cuts
plot(imp, var="x2", ladder=TRUE, ladder.cuts=1:8)
```

```
## plot with color/size using x1 (color) and y (size)
plot(imp, var="x2", col.var="x1", size.var="y")

## ------------------------------------------------------------
##
## survival example with plot
##
## ------------------------------------------------------------

data(peakVO2, package = "randomForestSRC")

## varpro/importance call
vp <- varpro(Surv(ttodead, died)~., peakVO2)
ivp <- ivarpro(vp, adaptive = FALSE, cut.max = 2)

## plot of peak vo2
## color displays interval (a measure of exercise time)
## size displays "y" which is predicted mortality in survival
plot(ivp, var="peak.vo2", col.var="interval", size.var="y")

## same but using beta blockers for color
plot(ivp, var="peak.vo2", col.var="betablok", size.var="y")
```

`outpro` *Subspace-aware out-of-distribution scoring with outPro*

Description

`outpro` computes an out-of-distribution (OOD) distance for new inputs using the predictor representation and variable priorities from a fitted `varpro` object, or the predictor representation from an `rfsrc` grow object. The score is subspace aware: departures are measured in selected coordinates, optionally weighted by model-derived variable priority, rather than by an unweighted global distance over all features.

Usage

```
outpro(object,
       newdata,
       neighbor = NULL,
       distancef = "knn",
       reduce = TRUE,
       cutoff = NULL,
       max.rules.tree = 150,
       max.tree = 150,
       knn.chunk.size = 100L,
       newdata.xscale = FALSE)
```

```
outpro.null(object,
            nulldata = NULL,
            neighbor = NULL,
            distancef = "knn",
            reduce = TRUE,
            cutoff = NULL,
            max.rules.tree = 150,
            max.tree = 150,
            knn.chunk.size = 100L,
            nulldata.xscale = FALSE)
```

Arguments

object	A fitted varpro object or an rfsrc object with classes c("rfsrc", "grow").
newdata	New data to score. If omitted, the training design matrix is scored. For varpro objects and newdata.xscale = FALSE, newdata should be supplied on the original data scale and is aligned to the fitted training design by the package's hot-encoding helper. If newdata.xscale = TRUE, newdata is assumed to already be on the fitted varpro x-scale.
neighbor	Number of training neighbors per case. If NULL, the default is min(n/10, 5000), where n is the number of training rows. For distancef = "knn", this is the number of ordinary nearest neighbors used in the selected standardized subspace. For the other distance functions, this is the number of model-derived neighbors requested from varpro.strength.
distancef	Distance function. One of "knn", "prod", "euclidean", "mahalanobis", "manhattan", "minkowski", or "kernel". The default is "knn". The "knn" option uses an ordinary weighted Manhattan nearest-neighbor distance in the selected subspace and does not call varpro.strength. The remaining options use model-derived neighbor frames and then aggregate coordinate-wise deviations across those neighbors.
reduce	Controls variable selection and weighting. If TRUE with a varpro object, variables are selected using model-based variable priority and the threshold cutoff. If TRUE with an rfsrc object, all predictors are used with unit weights. A character vector selects variables by name. A named numeric vector selects variables by name and supplies their weights. Any other value uses all predictors with unit weights.
cutoff	Threshold used with varpro variable-priority z values when reduce = TRUE. If NULL, a default based on the number of fitted predictor columns is used: .79 for moderate dimension and 0 for high dimension. This is not an OOD acceptance threshold; it controls only the reduced scoring subspace.
max.rules.tree	Maximum number of rules per tree used by varpro.strength for model-derived neighbor extraction. Ignored when distancef = "knn".
max.tree	Maximum number of trees used by varpro.strength for model-derived neighbor extraction. Ignored when distancef = "knn".
knn.chunk.size	Positive integer giving the number of scored rows processed at one time by the "knn" distance calculation. Smaller values use less memory; larger values can be faster. Ignored by the non-KNN distance functions.

`newdata.xscale` Logical value used only for `varpro` objects. If `FALSE`, the default, `newdata` is treated as raw user data and hot-encoded/aligned to the fitted training design. If `TRUE`, `newdata` is assumed to already contain the fitted x-scale columns, such as a matrix created internally from `object$x`. This option is mainly intended for internal package use, for example when scoring virtual data already constructed on the fitted `varpro` x-scale.

`nulldata` For `outpro.null`, optional in-distribution reference data used to form a null/reference distance distribution. If omitted, the training design matrix is used.

`nulldata.xscale`
Logical value used only for `varpro` objects when `nulldata` is supplied. It has the same meaning for `nulldata` as `newdata.xscale` has for `newdata`.

Details

For a `varpro` object, the training design `object$x` is the canonical fitted x-scale. When the original data include factors, `object$x` stores the hot-encoded representation used by the fitted forest. Ordinary user calls should normally leave `newdata.xscale = FALSE`, in which case new data are hot-encoded and aligned to the fitted training columns. Internal code can set `newdata.xscale = TRUE` when the data to score have already been constructed on the same fitted x-scale.

The default distance function is `"knn"`. It standardizes the selected variables using training means and standard deviations, computes weighted Manhattan distances from each scored input to the training inputs in the selected subspace, and returns the average distance to the `neighbor` closest training points. When `newdata` is omitted and the training data are scored against themselves, the exact self match is excluded from the nearest-neighbor set.

The non-KNN distance functions use model-derived reference neighborhoods. For these metrics, `outpro` calls `varpro.strength` on the fitted forest, extracts local neighbor frames, computes standardized coordinate-wise absolute deviations from each scored case to its neighbors, and aggregates those deviations by the requested metric. Available metrics are multiplicative product, weighted Euclidean, Mahalanobis, Manhattan, Minkowski, and kernel distance.

For `varpro` objects with `reduce = TRUE`, the selected subspace is determined from variable-priority `z` values using `cutoff`. If one or fewer variables pass the cutoff, all variables from the priority table are used. A character `reduce` value bypasses this step and directly specifies the scoring variables. A named numeric `reduce` value directly specifies both the scoring variables and their relative weights.

All distances are computed after standardizing selected variables with training means and scales. Variables with zero training standard deviation are dropped automatically. Larger `distance` values mean that the scored case is farther from the in-distribution reference set in the selected subspace.

`outpro.null` calls `outpro` on a reference sample and augments the result with the empirical CDF of the reference distances. This is useful for calibrating a distance to a reference quantile or for constructing a support score such as `1 - F_0(distance)`.

Value

`outpro` returns a list with components:

- `distance`: numeric vector with one OOD distance per scored case.
- `distance.object`: ingredients used for distance computation, including
  - `score`: neighbor frames returned by `varpro.strength`; `NULL` for `distancef = "knn"`.

  - neighbor: neighbor count used.
  - oob.bits: indicator of whether scoring was done on training rows or new data.
  - xvar.names: selected variable names after zero standard-deviation removal.
  - xvar.wt: variable weights after internal normalization and squaring.
  - dist.xvar: list of absolute coordinate-difference matrices in standardized units; NULL for distancef = "knn".
  - xorg.scale, xnew.scale: standardized training and scored matrices for the selected variables.
  - means, sds: training means and scales for the selected variables.
  - dropped.zero.sd.variables: variables removed because their training standard deviation was zero.
- distance.args: list of metric arguments actually used. For distancef = "knn", this includes the effective neighbor count, whether self matches were excluded, and knn.chunk.size.
- score: the neighbor information returned by varpro.strength; NULL for distancef = "knn".
- neighbor: neighbor setting used.
- cutoff: cutoff used for variable-priority reduction.
- oob.bits: indicator of whether scoring was done on training rows or new data.
- selected.variables: variables used in scoring after all filters.
- selected.weights: final selected-variable weights.
- dropped.zero.sd.variables: variables removed due to zero standard deviation.
- means, sds: training means and scales for convenience.
- newdata.xscale: logical value indicating whether supplied newdata was treated as already on the fitted x-scale.
- call: the matched call.

outpro.null returns the same list with two additional components:

- cdf: the empirical distribution function of distance.
- quantile: the empirical cumulative probability for each reference case.

Background

The method follows a model-centered view of OOD detection. The goal is not to estimate a full feature density, but to measure whether an input is well supported in the feature directions used by the fitted prediction rule. The default KNN score provides a fast selected-subspace support distance, while the non-KNN metrics use the forest structure to define model-derived local neighborhoods before aggregating coordinate deviations.

See Also

varpro, rfsrc.

Examples

```
## ------------------------------------------------

## fit a varPro model
data(BostonHousing, package = "mlbench")
smp <- sample(1:nrow(BostonHousing), size = nrow(BostonHousing) * .75)
train.data <- BostonHousing[smp,]
test.data <- BostonHousing[-smp,]
vp <- varpro(medv ~ ., data = train.data)

## Score new data with the default KNN metric
op <- outpro(vp, newdata = test.data)
head(op$distance)

## Forest-neighborhood multiplicative distance
op.prod <- outpro(vp, newdata = test.data, distancef = "prod")
head(op.prod$distance)

## Calibrate a null/reference distribution using training data
op.null <- outpro.null(vp)
head(op.null$quantile)
```

`partialpro` *Case-local Partial Profiles for VarPro Variables*

Description

Estimate and display case-local partial effect profiles for selected variables from a fitted `varpro` object. The method constructs virtual twins by varying one feature at a time, evaluates a prediction learner on those virtual records, and optionally removes virtual records that are far from the observed covariate support using Unlimited Virtual Twins (UVT). UVT filtering can be based on `isopro` isolation forest support or on `outpro` OOD distances.

Usage

```
partialpro(object, xvar.names, nvar,
           target, learner, newdata, method = c("unsupv", "rnd", "auto"),
           verbose = FALSE, vt.filter = c("isopro", "outpro", "none"),
           ...)
```

Arguments

`object`	A `varpro` object returned by `varpro`.

`xvar.names` Optional character vector of variables for which partial profiles are requested. If omitted, the variables returned by `get.topvars(object)` are used. Names not found in `object$xvar.names` are silently dropped.

`nvar` Optional integer limiting the number of variables used from `xvar.names`. If supplied, only the first `nvar` requested variables are used. If `xvar.names` is omitted, this limits the number of top VarPro variables.

`target` For classification, the class for which the partial profile is computed. This can be either an integer column index or a character class label. The default is the last class level. For regression and survival outcomes this argument is ignored and the first prediction column is used.

`learner` Optional prediction function. If omitted, `partialpro` uses the forest stored in `object$rf`. A supplied learner must accept a data frame of feature values with the same columns as `object$x`. It should return a numeric vector for regression or survival outcomes, or a matrix/data frame of class probabilities for classification outcomes. For classification, the column selected by `target` is used.

`newdata` Optional data frame of cases on which the partial profiles are conditioned. If omitted, up to `nsmp` training cases are sampled from `object$x` separately for each requested variable. If supplied, all rows of `newdata` are used. The data must already be on the fitted VarPro x-scale, i.e. it must contain the columns of `object$x`. This is automatic when using rows from `object$x`, for example `newdata = object$x[1:6, ]`. Extra columns are ignored after alignment to `colnames(object$x)`.

`method` Isolation forest method used by `isopro` when `vt.filter = "isopro"` and `cut != 0`. Options are `"unsupv"`, `"rnd"`, and `"auto"`. The default is `"unsupv"`. If there is only one top variable and `method = "unsupv"`, the code switches internally to `"rnd"`. This argument is ignored when `vt.filter = "outpro"` or `vt.filter = "none"`.

`verbose` Logical. If `TRUE`, print the variable currently being processed.

`vt.filter` Virtual twin filtering engine. The default `"isopro"` uses isolation forests through `isopro`. The option `"outpro"` uses out-of-distribution distances from `outpro`, calibrated through `outpro.null`. The option `"none"` disables virtual twin filtering, equivalent to setting `cut = 0`.

`...` Advanced options controlling the virtual grid, UVT filtering, and local smoothing. These options are intentionally not part of the main argument list, but they are user-adjustable. See **Advanced options** in `Details`.

Details

Method. For a requested variable `xnm`, `partialpro` creates a sequence of virtual values `xvirtual`. For each conditioning case, it constructs virtual twins by replacing `xnm` with each value in `xvirtual` while holding all other features fixed at that case's observed values. The prediction learner is then evaluated on the resulting virtual data.

The virtual grid is constructed as follows. Factors and variables with at most 10 unique observed values use their observed levels/values. Continuous variables with more than 10 unique values are trimmed to the central `alpha` to `1 - alpha` empirical range and then represented by up to `nvirtual`

quantile values. The actual number of grid points can be smaller than nvirtual if quantiles coincide.

If cut != 0 and vt.filter != "none", UVT filtering is applied before estimating the local profile. With vt.filter = "isopro", an isolation forest is fit by isopro using the top VarPro variables. Each virtual twin receives an isopro support score, and only virtual twins with score at least cut are treated as admissible.

With vt.filter = "outpro", virtual twins are scored by outpro on the fitted VarPro x-scale. The public outpro function normally hot-encodes raw newdata; here the virtual matrix is already aligned to object$x, so partialpro calls outpro with newdata.xscale = TRUE. The raw outpro distance is calibrated against the null/reference distance distribution from outpro.null and converted to a support score 1 - F0(distance). The same rule is then used: a virtual twin is admissible when this support score is at least cut. Thus cut = 0 disables UVT filtering, while larger values of cut retain fewer virtual twins. The default outpro distance for partialpro is "knn", which uses a standardized nearest-neighbor support calculation and avoids the forest-neighborhood computations used by several other outpro distances.

For regression and survival outcomes, learner predictions are used on their native prediction scale. For classification, the selected class probability is converted to a clipped log-odds scale using probabilities truncated to the interval [0.001, 0.999].

For continuous variables, a local polynomial profile is fit separately for each conditioning case using lm.fit and the design 1, x, x^2, ..., x^df. A case is retained if it has at least min(nmin, nxorg / 2) admissible virtual twins, where nxorg is the number of unique observed values of the variable. When UVT filtering is active and enough admissible training virtual twins are available, the code fits both a UVT-restricted polynomial and an unrestricted polynomial on a case-specific train split of virtual values. The unrestricted fit is used only if its held-out standardized MSE is lower than the UVT-restricted MSE by more than mse.tolerance; otherwise the UVT-restricted fit is used.

For binary variables, no polynomial extrapolation is used. The profile is computed directly from the mean predicted value at the two virtual values, and both virtual values must be admissible for a non-missing case-specific profile.

The component named yhat.causal is the case-specific profile centered at the first virtual value, i.e. the estimated contrast relative to xvirtual[1]. The name is retained for compatibility with the VarPro terminology, but causal interpretation requires the usual substantive and design assumptions; the function itself estimates model-based partial profiles and contrasts.

Advanced options supplied through ...

cut UVT threshold. Virtual twins with support score less than cut are excluded. For vt.filter = "isopro", the support score is the isopro score. For vt.filter = "outpro", the support score is 1 - F0(distance), where F0 is the empirical null/reference CDF from outpro.null. Default is 0.1. Set cut = 0, or use vt.filter = "none", to turn off UVT filtering.

nsmp Number of training cases sampled when newdata is omitted. The actual number is min(nrow(object$x), nsmp). Default is 250. This option is ignored when newdata is supplied.

nvirtual Requested number of virtual grid values for a continuous variable with more than 10 unique values. Default is 100. The actual grid can be shorter if quantile values are tied. Factors and variables with at most 10 unique values use all observed levels/values instead.

nmin Minimum number of admissible virtual twins required for a continuous-variable case profile, after applying the cap min(nmin, nxorg / 2). Default is 15.

`alpha` Tail trimming used to define the continuous virtual grid. The grid is formed from values between the empirical `alpha` and `1 - alpha` quantiles. Default is `0.025`.

`df` Degree of the local polynomial used for continuous variables. The value is rounded and constrained to be at least 1. Default is 2, giving a quadratic local profile.

`sampsize` Sample-size function passed to `isopro` when `vt.filter = "isopro"` and UVT filtering is active. The default is `function(x) min(2^8, 0.632 * x)`.

`ntree` Number of trees used by the `isopro` isolation forest when `vt.filter = "isopro"` and UVT filtering is active. Default is `500`.

`nodesize` Terminal node size used by `isopro` when `vt.filter = "isopro"` and UVT filtering is active. Default is `1`.

`mse.tolerance` Tolerance in the held-out comparison between the UVT-restricted and unrestricted polynomial fits. The unrestricted fit is selected only when its standardized MSE is smaller by more than this amount. Default is `0`.

`out.distancef` Distance used by `outpro` when `vt.filter = "outpro"`. The default is `"knn"`. Other accepted values are those supported by `outpro`, including `"prod"`, `"euclidean"`, `"mahalanobis"`, `"manhattan"`, `"minkowski"`, and `"kernel"`.

`out.neighbor` Neighbor count passed to `outpro` and `outpro.null`. The default `NULL` lets `outpro` choose its own value.

`out.reduce` Reduced subspace passed to `outpro` when `vt.filter = "outpro"`. The default `NULL` uses the top VarPro variables together with the focal variable currently being profiled. A character vector selects named variables; a named numeric vector selects variables and supplies their weights. In these two cases, the focal variable is added if absent. Other values, such as `TRUE` or `FALSE`, are passed through to `outpro`.

`out.cutoff` Cutoff passed to `outpro` when `out.reduce = TRUE`. The default `NULL` lets `outpro` use its dimension-dependent default. This is not the virtual twin threshold; the virtual twin threshold is `cut`.

`out.max.rules.tree` Maximum number of rules per tree passed to `outpro` for forest-neighborhood distances. Default is `150`. This option has little effect for the default `out.distancef = "knn"`.

`out.max.tree` Maximum number of trees passed to `outpro` for forest-neighborhood distances. Default is `150`. This option has little effect for the default `out.distancef = "knn"`.

`out.knn.chunk.size` Chunk size used by the `outpro` KNN distance calculation. Default is `100L`. Increasing this value can be faster but uses more memory.

`out.null` Optional precomputed `outpro.null` object, or a named list of such objects keyed by profiled variable name. This is mainly useful when repeatedly calling `partialpro` with `vt.filter = "outpro"` and the same calibration should be reused. If omitted, null calibration is computed internally and cached by reduced subspace.

Custom learner helpers. The package also supplies convenience learner constructors used in the examples. `rf.learner(o, ...)` re-fits a random forest using the top VarPro variables as a gentler variable-weighting scheme and passes recognized random-forest arguments through `...`. `gbm.learner(o, ...)` fits a gbm model with defaults `n.trees = 500`, `shrinkage = 0.1`, `interaction.depth = 3`, `cv.folds = 5`, and `n.cores = get.mc.cores()`; it uses Gaussian loss for regression/survival scores and Bernoulli loss for two-class classification. `bart.learner(o, ...)` fits a BART regression learner and accepts `mc.cores`; it applies only to regression/survival-style VarPro objects.

Because `partialpro` samples cases and creates case-specific train/test splits of the virtual grid, use `set.seed` before calling the function when reproducible profiles are needed.

Value

An invisible named list with class "partialpro". There is one named component for each requested variable that remains after matching to object$xvar.names. A component can be NULL if no usable virtual profiles are available for that variable. Otherwise each component is a list with the following elements:

case Indices of the sampled training cases, or row numbers of newdata, retained for the variable.

xorg Original observed values of the variable in object$x.

xvirtual Virtual grid values used for the variable.

goodvt Matrix with one row per retained case and one column per virtual value. Entries are 1 for admissible virtual twins and NA for virtual twins removed by UVT.

yhat.par Case-specific fitted partial profile matrix on the prediction scale.

yhat.nonpar Case-specific fitted partial profile matrix on the prediction scale. For the current continuous-variable implementation, this is the local fitted profile with a common global intercept added back.

yhat.causal Case-specific contrast matrix centered at the first virtual value.

Author(s)

Min Lu and Hemant Ishwaran

References

Ishwaran H. (2025). Multivariate Statistics: Classical Foundations and Modern Machine Learning, CRC (Chapman and Hall), in press.

See Also

varpro, isopro, outpro

Examples

```
##------------------------------------------------------------------
##
## Boston housing
##
##------------------------------------------------------------------

library(mlbench)
data(BostonHousing)
oldpar <- par(mfrow=c(2,3))
plot((oo.boston<-partialpro(varpro(medv~.,BostonHousing),nvar=6)))
par(oldpar)

##------------------------------------------------------------------
##
## Boston housing using newdata option
```

```
##
##
##-----------------------------------------------------------------

library(mlbench)
data(BostonHousing)
o <- varpro(medv~.,BostonHousing)
oldpar <- par(mfrow=c(2,3))
plot(partialpro(o,nvar=3))
## same analysis using outpro/KNN filtering
plot(partialpro(o,nvar=3,vt.filter="outpro"))
## same but using newdata (set to first 6 cases of the training data)
plot(partialpro(o,newdata=o$x[1:6,],nvar=3))
par(oldpar)

##-----------------------------------------------------------------
##
## Boston housing with externally constructed rf learner
##
##-----------------------------------------------------------------

## varpro analysis
library(mlbench)
data(BostonHousing)
o <- varpro(medv~.,BostonHousing)

## default partial pro call
pro <- partialpro(o, nvar=3)

## partial pro call using built in rf learner
mypro <- partialpro(o, nvar=3, learner=rf.learner(o))

## compare the two
oldpar <- par(mfrow=c(2,3))
plot(pro)
plot(mypro, ylab="external rf learner")
par(oldpar)

##-----------------------------------------------------------------
##
## Boston housing:  tree gradient boosting learner, bart learner
##
##-----------------------------------------------------------------

if (library("gbm", logical.return=TRUE) &&
    library("BART", logical.return=TRUE)) {

## varpro analysis
library(parallel)
library(mlbench)
data(BostonHousing)
o <- varpro(medv~.,BostonHousing)
```

```
## default partial pro call
pro <- partialpro(o, nvar=3)

## partial pro call using built in gradient boosting learner
mypro <- partialpro(o, nvar=3, learner=gbm.learner(o, n.trees=1000, n.cores=get.mc.cores()))

## partial pro call using built in bart learner
mypro2 <- partialpro(o, nvar=3, learner=bart.learner(o, mc.cores=get.mc.cores()))

## compare the learners
oldpar <- par(mfrow=c(3,3))
plot(pro)
plot(mypro, ylab="external boosting learner")
plot(mypro2, ylab="external bart learner")
par(oldpar)
}

##------------------------------------------------------------------
##
## peak vo2 with 5 year rmst
##
##------------------------------------------------------------------

data(peakVO2, package = "randomForestSRC")
oldpar <- par(mfrow=c(2,3))
plot((oo.peak<-partialpro(varpro(Surv(ttodead,died)~.,peakVO2,rmst=5),nvar=6)))
par(oldpar)

##------------------------------------------------------------------
##
## veteran data set with celltype as a factor
##
##------------------------------------------------------------------

data(veteran, package = "randomForestSRC")
dta <- veteran
dta$celltype <- factor(dta$celltype)
oldpar <- par(mfrow=c(2,3))
plot((oo.veteran<-partialpro(varpro(Surv(time, status)~., dta), nvar=6)))
par(oldpar)

##------------------------------------------------------------------
##
## iris: classification analysis showing partial effects for all classes
##
##------------------------------------------------------------------

o.iris <- varpro(Species~.,iris)
yl <- paste("log-odds", levels(iris$Species))
oldpar <- par(mfrow=c(3,2))
plot((oo.iris.1 <- partialpro(o.iris, target=1, nvar=2)),ylab=yl[1])
plot((oo.iris.2 <- partialpro(o.iris, target=2, nvar=2)),ylab=yl[2])
plot((oo.iris.3 <- partialpro(o.iris, target=3, nvar=2)),ylab=yl[3])
```

```
par(oldpar)

##------------------------------------------------------------------
##
## iowa housing data
##
##------------------------------------------------------------------

## quickly impute the data; log transform the outcome
data(housing, package = "randomForestSRC")
housing <- randomForestSRC::impute(SalePrice~., housing, splitrule="random", nimpute=1)
dta <- data.frame(data.matrix(housing))
dta$y <- log(housing$SalePrice)
dta$SalePrice <- NULL

## partial effects analysis
o.housing <- varpro(y~., dta, nvar=Inf)
oo.housing <- partialpro(o.housing,nvar=15)
oldpar <- par(mfrow=c(3,5))
plot(oo.housing)
par(oldpar)
```

`plot.ivarpro` *Plot iVarPro Gradients for a Single Variable*

Description

`plot.ivarpro()` draws a base-graphics scatterplot of case-specific iVarPro gradients for one predictor: the x-axis is the predictor value and the y-axis is the iVarPro local gradient. Points can be colored (`col.var`), sized (`size.var`), optionally jittered, and optionally overlaid with loess smooths stratified by `col.var`.

Usage

```
## S3 method for class 'ivarpro'
plot(x,
    var,
    col.var = NULL,
    size.var = NULL,
    data = NULL,
    target = NULL,
    ladder = FALSE,
    ladder.cuts = NULL,
    ladder.max.segments = 3000,
    pch = 16,
    cex = 0.8,
    cex.range = c(0.5, 2),
```

```
    main = NULL,
    xlab = NULL,
    ylab = "iVarPro gradient",
    legend = TRUE,
    ...)
```

Arguments

x	An object returned by ivarpro or predict.ivarpro. If data is not supplied, plot.ivarpro() uses attr(x, "data") when available.
var	Variable to plot (name or column index). Must exist in both the iVarPro object and data.
col.var	Optional variable (column name in data) used for coloring. If treated as categorical, colors are assigned per level; if continuous, a color ramp is used and the legend shows selected quantiles.
size.var	Optional variable (column name in data) used to scale point sizes to cex.range.
data	Optional data.frame or matrix of feature values used for the x-axis variable and any auxiliary variables referenced by col.var or size.var.
target	Optional target/outcome coordinate when x is a list-valued iVarPro result (for example multiclass or multivariate output). May be specified as a single name or index. If omitted, the first element is used.
ladder	Logical. If TRUE, attempt to draw a ladder-based band (vertical segments spanning the range across ladder.cuts) when path membership information is present.
ladder.cuts	Optional subset of ladder indices/values used for the band.
ladder.max.segments	Maximum number of ladder segments to draw.
pch	Point character for the default point style.
cex	Base point size (used when size.var is not supplied).
cex.range	Min/max point sizes used when size.var is supplied.
main	Main title (defaults to paste0(var, " vs iVarPro gradient")).
xlab	X-axis label (defaults to var).
ylab	Y-axis label (defaults to "iVarPro gradient").
legend	Logical. If TRUE and col.var is supplied, draw a legend describing the color mapping.
...	Additional arguments passed to graphics::plot(), plus these optional controls: smooth Logical. Draw loess smooth curves. Default TRUE. For categorical col.var, one curve per level. For continuous col.var, curves are drawn for strata defined by quantiles. smooth.span Loess span (default 0.75). Other loess controls: smooth.degree, smooth.family, smooth.lwd, smooth.lty, smooth.alpha, smooth.min.n, smooth.n.grid.

`jitter` Logical or numeric. Default `TRUE`. Adds horizontal jitter to the x-values to reduce overplotting. Related options: `jitter.amount`, `jitter.fraction`, `jitter.seed`.

`x.dist` Character vector controlling an x-axis distribution strip. Default `"none"`. Supported values include `"rug"`, `"hist"`, `"density"`, and `"auto"` (defaults to `c("hist","rug")`). Use `x.dist = c("hist","rug")` (or `c("density","rug")`) to combine.
Related options: `x.dist.side`, `x.dist.height`, `x.dist.pad`, `x.dist.col`, `x.dist.border`, `x.dist.lwd`, `x.dist.lty`, `x.dist.bins`, `x.dist.adjust`, `x.dist.n`, `x.dist.rug.col`, `x.dist.rug.lwd`, `x.dist.rug.ticksize`, `x.dist.rug.max`. By default, an outline is drawn to keep the strip visible; set `x.dist.lwd = 0` to suppress the outline.

`col.legend.probs`, `col.legend.n` Quantiles shown in the legend when `col.var` is continuous. Default is 5 quantiles: `c(0.05, 0.25, 0.5, 0.75, 0.95)`.

`smooth.probs`, `smooth.n` Quantiles defining strata for smooth curves when `col.var` is continuous. Default matches the legend quantiles.

`zero.line` Logical. Default `TRUE`. Adds a dashed reference line at gradient 0 (`y = 0`). Related options: `zero.line.col`, `zero.line.lty`, `zero.line.lwd`.

`col.var.discrete.max` If `col.var` is numeric with at most this many distinct values, treat it as categorical (default `10`).

`col.style` How categorical colors are rendered: `"auto"` (default), `"solid"`, `"outline"`, `"binary"`. Related options include `col.outline`, `col.binary.pch`, `col.dodge`.

Details

Jitter is applied for visualization only; loess smooths and the optional x-axis distribution strip (`x.dist`) are computed using the unjittered x-values.

Value

Invisibly returns `TRUE`.

See Also

`ivarpro`, `varpro`, `shap.ivarpro`

Examples

```
## ------------------------------------------------------------
##
## Survival example: peakVO2 plot
##
## ------------------------------------------------------------

data(peakVO2, package = "randomForestSRC")

ipv <- ivarpro(varpro(Surv(ttodead, died) ~ ., peakVO2))
```

```
## Continuous col.var: legend + smooth strata default to 5 quantiles
plot(ipv, var = "peak.vo2", col.var = "interval", size.var = "y")

## Add an x-axis distribution strip (histogram + rug)
plot(ipv, var = "peak.vo2", col.var = "interval", size.var = "y",
     x.dist = c("hist", "rug"))

## Increase legend/smooth strata (e.g., 7 quantiles)
plot(ipv, var = "peak.vo2", col.var = "interval", size.var = "y",
     col.legend.n = 7, smooth.n = 7)

## Classic 3-quantile view (5%, 50%, 95%)
plot(ipv, var = "peak.vo2", col.var = "interval", size.var = "y",
     col.legend.probs = c(0.05, 0.5, 0.95),
     smooth.probs = c(0.05, 0.5, 0.95))

## Factor col.var example: one smooth per level
plot(ipv, var = "peak.vo2", col.var = "betablok", size.var = "y")

## ------------------------------------------------------------
##
## multiclass example: iris (use target= to choose a class)
##
## ------------------------------------------------------------

data(iris)

vp.ir  <- varpro(Species ~ ., iris, ntree = 50)
ivp.ir <- ivarpro(vp.ir)

## Plot gradients for the "setosa" class (target selects the list element)
plot(ivp.ir, var = "Petal.Length", target = "setosa",
     col.var = "Species", data = iris)

## Alternatively, color by the predicted class probability stored in ivp.ir
plot(ivp.ir, var = "Petal.Length", target = "setosa",
     col.var = "y.setosa")


## ------------------------------------------------------------
##
## multiclass example: wine (advanced example)
##
## ------------------------------------------------------------

data(wine, package = "randomForestSRC")

## Give the class labels nicer names than "3", "4", ..., "9"
wine$quality <- factor(wine$quality)
levels(wine$quality) <- paste0("Q", levels(wine$quality))  # "Q3" "Q4" ... "Q9"

vp  <- varpro(quality ~ ., wine, ntree = 50)
```

```
ivp <- ivarpro(vp)

## Available targets correspond to class probability columns
names(ivp)

## Build a plotting data.frame that contains:
##  - predictors (from attr(ivp,"data"))
##  - predicted class probabilities (y.Q3, y.Q4, ..., y.Q9)
##  - the observed class label (quality)
xdat <- attr(ivp, "data")
xdat$quality <- wine$quality

## Plot gradients for the "Q7" class:
## y-axis: d P(Y="Q7" | x) / d alcohol  (iVarPro gradient)
## x-axis: alcohol
## color: observed class label
plot(ivp, var = "alcohol", target = "Q7",
     data = xdat, col.var = "quality")

## Alternatively, color by the model's predicted probability for the same class.
## (These columns come from attr(ivp,"data") as y.<class>)
plot(ivp, var = "alcohol", target = "Q7", col.var = "y.Q7")
```

`plot.partialpro` *Plot method for partialpro objects*

Description

Plot partial effects from a previous `partialpro` analysis.

Usage

```
## S3 method for class 'partialpro'
plot(x, xvar.names, nvar,
  parametric = FALSE, se = TRUE,
  causal = FALSE, subset = NULL, plot.it = TRUE, ...)
```

Arguments

`x`	A `partialpro` object returned from a previous call to `partialpro`.
`xvar.names`	Names (or integer indices) of the x-variables to plot. Defaults to all variables.
`nvar`	Number of variables to plot. Defaults to all variables.
`parametric`	Logical. Set to `TRUE` only if the partial effect is believed to follow a polynomial form.
`se`	Display standard errors?
`causal`	Display causal estimator?

subset	Optional conditioning factor. Not applicable if parametric = TRUE. May also be a logical or integer vector to subset the analysis.
plot.it	If FALSE, no plot is produced; instead, the internal plotting objects are returned.
...	Additional arguments passed to plot.

Details

Generates smoothed partial-effect plots for continuous variables. The solid black line represents the estimated partial effect; dashed red lines show an approximate plus-minus standard error band. These standard errors are intended as heuristic guides and should be interpreted cautiously.

Partial effects are estimated nonparametrically using locally fitted polynomial models. This is the default behavior and is recommended when effects are expected to be nonlinear. Use parametric = TRUE if the underlying effect is believed to follow a global polynomial form.

For binary variables, partial effects are shown as boxplots, with whiskers reflecting variability analogous to standard error.

The causal estimator, when requested, displays the baseline-subtracted local effect.

Conditioning is supported via the subset option. When supplied as a factor (with length equal to the original data), the plot is stratified by its levels. Alternatively, subset can be a logical or integer vector indicating the cases to include in the analysis.

Value

If plot.it = TRUE, the method is called for its side effect of producing plots and returns NULL.

If plot.it = FALSE, the method returns a named list of internal plot objects, one per requested variable, containing the partial-effect curves and associated summaries used for plotting.

Author(s)

Min Lu and Hemant Ishwaran

References

Ishwaran H. (2025). Multivariate Statistics: Classical Foundations and Modern Machine Learning, CRC (Chapman and Hall), in press.

See Also

partialpro

Examples

```
##------------------------------------------------------------------
##
## Boston housing
##
##------------------------------------------------------------------

library(mlbench)
```

```
data(BostonHousing)
o.boston <- varpro(medv~., BostonHousing)
oo.boston <- partialpro(o.boston, nvar = 4, learner = rf.learner(o.boston))

oldpar <- par(mfrow = c(2, 4))

## parametric local estimation
plot(oo.boston, parametric = TRUE, ylab = "parametric est.")

## non-parametric local estimation (default)
plot(oo.boston, parametric = FALSE, ylab = "non-parametric est.")

par(oldpar)

##------------------------------------------------------------------
##
## Boston housing with subsetting
##
##------------------------------------------------------------------

library(mlbench)
data(BostonHousing)
o.boston <- varpro(medv~., BostonHousing)
oo.boston <- partialpro(o.boston, nvar = 3, learner = rf.learner(o.boston))

## subset analysis
price <- BostonHousing$medv
pricef <- factor(price > median(price), labels = c("low priced", "high priced"))
oldpar <- par(mfrow = c(1, 1))
plot(oo.boston, subset = pricef, nvar = 1)
par(oldpar)

##------------------------------------------------------------------
##
## veteran data with subsetting using celltype as a factor
##
##------------------------------------------------------------------

data(veteran, package = "randomForestSRC")
dta <- veteran
dta$celltype <- factor(dta$celltype)
o.vet <- varpro(Surv(time, status) ~ ., dta)
oo.vet <- partialpro(o.vet, nvar = 6, nsmp = Inf, learner = rf.learner(o.vet))

## partial effects, with subsetting
oldpar <- par(mfrow = c(2, 3))
plot(oo.vet, subset = dta$celltype)
par(oldpar)

## causal effects, with subsetting
oldpar <- par(mfrow = c(2, 3))
plot(oo.vet, subset = dta$celltype, causal = TRUE)
par(oldpar)
```

```
## retrieve plotting objects without drawing
obj <- plot(oo.vet, subset = dta$celltype, plot.it = FALSE)
str(obj, max.level = 1)
```

`predict.isopro` *Prediction for Isopro for Identifying Anomalous Data*

Description

Use isolation forests to identify rare/anomalous values using test data.

Usage

```
## S3 method for class 'isopro'
predict(object, newdata, quantiles = TRUE, ...)
```

Arguments

`object`	`isopro` object returned from a previous call.
`newdata`	Optional test data. If not provided, the training data is used.
`quantiles`	Logical. If `TRUE` (default), returns quantile values; if `FALSE`, returns case depth values.
`...`	Additional arguments passed to internal methods.

Details

Uses a previously constructed `isopro` object to assess anomalous observations in the test data. By default, returns quantile values representing the depth of each test observation relative to the original training data. Smaller values indicate greater outlyingness.

To return raw depth values instead of quantiles, set `quantiles = FALSE`.

Value

Anomaly scores for the test data (or training data).

Author(s)

Min Lu and Hemant Ishwaran

References

Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. (2008). Isolation forest. 2008 Eighth IEEE International Conference on Data Mining. IEEE.

Ishwaran H. (2025). Multivariate Statistics: Classical Foundations and Modern Machine Learning, CRC (Chapman and Hall), in press.

See Also

isopro uvarpro varpro

Examples

```
## ------------------------------------------------------------
##
## boston housing
## unsupervised isopro analysis
##
## ------------------------------------------------------------

## training
data(BostonHousing, package = "mlbench")
o <- isopro(data=BostonHousing)

## make fake data
fake <- do.call(rbind, lapply(1:nrow(BostonHousing), function(i) {
  fakei <- BostonHousing[i,]
  fakei$lstat <- quantile(BostonHousing$lstat, .99)
  fakei$nox <- quantile(BostonHousing$nox, .99)
  fakei
}))

## compare depth values for fake data to training data
depth.fake <- predict(o, fake)
depth.train <- predict(o)
depth.data <- rbind(data.frame(whichdata="fake", depth=depth.fake),
                    data.frame(whichdata="train", depth=depth.train))
boxplot(depth~whichdata, depth.data, xlab="data", ylab="depth quantiles")


## ------------------------------------------------------------
##
## boston housing
## isopro supervised analysis with different split rules
##
## ------------------------------------------------------------

data(BostonHousing, package="mlbench")

## supervised isopro analysis using different splitrules
o <- isopro(formula=medv~.,data=BostonHousing)
o.hvwt <- isopro(formula=medv~.,data=BostonHousing,splitrule="mse.hvwt")
o.unwt <- isopro(formula=medv~.,data=BostonHousing,splitrule="mse.unwt")

## make fake data
fake <- do.call(rbind, lapply(1:nrow(BostonHousing), function(i) {
  fakei <- BostonHousing[i,]
  fakei$lstat <- quantile(BostonHousing$lstat, .99)
  fakei$nox <- quantile(BostonHousing$nox, .99)
  fakei
```

```
}))

## compare depth values for fake data to training data
depth.train <- predict(o)
depth.hvwt.train <- predict(o.hvwt)
depth.unwt.train <- predict(o.unwt)
depth.fake <- predict(o, fake)
depth.hvwt.fake <- predict(o.hvwt, fake)
depth.unwt.fake <- predict(o.unwt, fake)
depth.data <- rbind(data.frame(whichdata="fake", depth=depth.fake),
                    data.frame(whichdata="fake.hvwt", depth=depth.hvwt.fake),
                    data.frame(whichdata="fake.unwt", depth=depth.unwt.fake),
                    data.frame(whichdata="train", depth=depth.train),
                    data.frame(whichdata="train.hvwt", depth=depth.hvwt.train),
                    data.frame(whichdata="train.unwt", depth=depth.unwt.train))
boxplot(depth~whichdata, depth.data, xlab="data", ylab="depth quantiles")
```

`predict.ivarpro` *Predict iVarPro Gradients on New Data*

Description

Calculates case-specific iVarPro gradients on a new feature matrix by reusing the rule-level gradients stored in a previously computed iVarPro object and recomputing rule membership for the new cases.

Usage

```
## S3 method for class 'ivarpro'
predict(object,
  newdata = NULL,
  model = NULL,
  noise.na = NULL,
  path.store.membership = FALSE,
  save.data = TRUE,
  ...)
```

Arguments

`object`	An object returned by `ivarpro` (a numeric `data.frame` for univariate outcomes, or a list of such `data.frames` for multivariate outcomes). The object must contain the `"ivarpro.path"` attribute with stored rule metadata and rule-level gradients.
`newdata`	Optional data.frame of predictors for which to compute case-specific gradients. If supplied, gradients are computed for `newdata` using full membership (all eligible trees). If `NULL` (default), the function attempts a restore prediction for the

training data, using OOB membership so that the returned gradients match the original `ivarpro()` output when possible.

`model` Optional model override. If `NULL` (default), the model is taken from `attr(object, "model")`. The stored model can be either a `varpro` object or a `randomForestSRC` `rfsrc` grow object.

`noise.na` Logical controlling how cells with no usable contributing rules are handled. If `NULL` (default), inherits the setting stored in `attr(object, "ivarpro.path")$noise.na`. If `TRUE`, such cells are set to `NA`; if `FALSE`, they are set to zero.

`path.store.membership`
Logical. If `TRUE`, store the rule membership indices (case IDs) for the prediction in `attr(out, "ivarpro.path")$oobMembership`. This enables ladder-based bands in `plot.ivarpro` and summaries via `ivarpro_band()` for the predicted gradients, at the cost of additional memory. Default is `FALSE`.

`save.data` Logical. If `TRUE` (default) and `newdata` is supplied, save the predictor data used for prediction as `attr(out, "data")`, enabling downstream plotting functions (e.g., `plot.ivarpro`) to retrieve `data` automatically.

`...` Additional arguments passed to `randomForestSRC::predict.rfsrc()`.

Details

A previously computed iVarPro object contains (i) a set of retained VarPro rules identified by tree and node/branch IDs, (ii) a release-variable index for each rule, and (iii) an estimated rule-level gradient. For new cases, the function obtains terminal node membership for each tree using `randomForestSRC::predict.rfsrc(membership = TRUE)` and determines which stored rules apply to each case by matching the case's terminal node to the stored rule node/branch ID for that tree. Case-specific gradients are then computed by aggregating the stored rule-level gradients over all rules that apply to the case and release the corresponding variable.

When `newdata` is not supplied, `predict()` attempts to return OOB case-specific gradients for the original training data. When the iVarPro object was created with membership information stored in its path (`path.store.membership = TRUE`, the default for `ivarpro()`), the restore prediction will reproduce the original iVarPro matrix.

If `save.data = TRUE` and `newdata` is supplied, `attr(out, "data")` contains the predictor matrix used for prediction, so downstream wrappers such as `plot.ivarpro` can be used directly. If you want ladder bands on predicted gradients, set `path.store.membership = TRUE` when predicting.

Value

Returns an object with the same shape as `object`:

- For univariate outcomes, a numeric `data.frame` of dimension $n_{test} \times p$ containing predicted case-specific gradients.
- For multivariate outcomes, a named list of such `data.frames`, one per outcome coordinate.

The returned object includes an `"ivarpro.path"` attribute containing rule metadata and (optionally) prediction-time membership indices.

See Also

ivarpro, plot.ivarpro, varpro

Examples

```
## ------------------------------------------------------------
##
## Restore mode example (training OOB gradients are reproduced)
##
## ------------------------------------------------------------

data(peakVO2, package = "randomForestSRC")

## Fit VarPro and iVarPro
vp  <- varpro(Surv(ttodead, died) ~ ., peakVO2, ntree = 50)
imp <- ivarpro(vp)

## Restore prediction: should match the original iVarPro matrix
p.restore <- predict(imp)
all.equal(p.restore, imp, check.attributes = FALSE)

## Downstream plotting using the restored gradients
plot(p.restore, var = "peak.vo2",
     col.var = "interval", size.var = "y",
     data = attr(imp, "data"))


## ------------------------------------------------------------
##
## Synthetic example (Friedman #1) with prediction on new data
##
## ------------------------------------------------------------

if (requireNamespace("mlbench", quietly = TRUE)) {

  set.seed(123)

  ## training data
  tr <- mlbench::mlbench.friedman1(500, sd = 1)
  train <- data.frame(tr$x, y = tr$y)
  colnames(train)[1:ncol(tr$x)] <- paste0("x", 1:ncol(tr$x))

  ## ivarpro fit on training data
  vp <- varpro(y ~ ., train, ntree = 100)
  imp <- ivarpro(vp)

  ## test data
  te <- mlbench::mlbench.friedman1(1e4, sd = 1)
  test <- data.frame(te$x)
  colnames(test) <- paste0("x", 1:ncol(te$x))
```

```
  ## predicted gradients on test data
  p.test <- predict(imp, newdata = test)

  ## partial plot directly on predicted object
  plot(p.test, var = "x1", col.var = "x2")


  ## push x1 outside the original Friedman support [0, 1]
  ## a heuristic way to test out-of-distribution (OOD)
  test.ood <- test
  test.ood$x1 <- runif(nrow(test), min = -0.5, max = 1.5)

  ## predicted gradients on OOD test data
  p.test.ood <- predict(imp, newdata = test.ood)

  ## partial plot showing support for x1
  plot(p.test.ood, var = "x1", col.var = "x2",
       x.dist = c("density", "rug"))
  ## reference lines for the original training support of x1
  abline(v = c(0, 1), lty = 2)

}
```

`predict.uvarpro` *Prediction on Test Data using Unsupervised VarPro*

Description

Obtain predicted values on test data for unsupervised forests.

Usage

```
## S3 method for class 'uvarpro'
predict(object, newdata, ...)
```

Arguments

`object`	Unsupervised VarPro object from a previous call to `uvarpro`. Only applies if `method = "auto"` was used.
`newdata`	Optional test data. If not provided, the training data is used.
`...`	Additional arguments passed to internal methods.

Details

Applies to unsupervised VarPro objects built using the autoencoder (`method = "auto"`). The object contains a multivariate random forest used to generate predictions for the test data.

Value

Returns a matrix of predicted values, where each column corresponds to a feature (with one-hot encoding applied). The result includes the following attributes:

1. `mse`: Standardized mean squared error averaged across features.
2. `mse.all`: Standardized mean squared error for each individual feature.

Author(s)

Min Lu and Hemant Ishwaran

See Also

`uvarpro`

Examples

```
## ------------------------------------------------------------
##
## boston housing
## obtain predicted values for the training data
##
## ------------------------------------------------------------

## unsupervised varpro on boston housing
data(BostonHousing, package = "mlbench")
o <- uvarpro(data=BostonHousing)

## predicted values for the training features
print(head(predict(o)))

## ------------------------------------------------------------
##
## mtcars
## obtain predicted values for test data
## also illustrates hot-encoding working on test data
##
## ------------------------------------------------------------

## mtcars with some factors
d <- data.frame(mpg=mtcars$mpg,lapply(mtcars[, c("cyl", "vs", "carb")], as.factor))

## training
o <- uvarpro(d[1:20,])

## predicted values on test data
print(predict(o, d[-(1:20),]))

## predicted values on bad test data with strange factor values
dbad <- d[-(1:20),]
dbad$carb <- as.character(dbad$carb)
```

```
dbad$carb <-  sample(LETTERS, size = nrow(dbad))
print(predict(o, dbad))
```

`predict.varpro` *Prediction on Test Data using VarPro*

Description

Obtain predicted values on test data for VarPro object.

Usage

```
## S3 method for class 'varpro'
predict(object, newdata, ...)
```

Arguments

`object`	VarPro object returned from a previous call to `varpro`.
`newdata`	Optional test data. If not provided, predictions are computed using the training data (out-of-bag).
`...`	Additional arguments passed to internal methods.

Details

VarPro uses rules extracted from a random forest built using guided tree-splitting, where variables are selected based on split-weights computed in a preprocessing step.

Value

Returns predicted values for the input data. If `newdata` is provided, predictions are made on that data; otherwise, out-of-bag predictions for the training data are returned.

Author(s)

Min Lu and Hemant Ishwaran

References

Lu, M. and Ishwaran, H. (2024). Model-independent variable selection via the rule-based variable priority. arXiv e-prints, pp.arXiv-2409.

See Also

`varpro`

Examples

```
## ------------------------------------------------------------
## toy example - needed to pass CRAN test
## ------------------------------------------------------------

## train call
o <- varpro(mpg~., mtcars[1:20,], ntree = 1)

## predict call
print(predict(o, mtcars[-(1:20),]))



## ------------------------------------------------------------
##
## boston housing regression
## obtain predicted values for the training data
##
## ------------------------------------------------------------

## varpro applied to boston housing data
data(BostonHousing, package = "mlbench")
o <- varpro(medv~., BostonHousing)

## predicted values for the training features
print(head(predict(o)))

## ------------------------------------------------------------
##
## iris classification
## obtain predicted values for test data
##
## ------------------------------------------------------------

## varpro applied to iris data
trn <- sample(1:nrow(iris), size = 100, replace = FALSE)
o <- varpro(Species~., iris[trn,])

## predicted values on test data
print(data.frame(Species=iris[-trn, "Species"], predict(o, iris[-trn,])))

## ------------------------------------------------------------
##
## mtcars regression: illustration of hot-encoding on test data
##
## ------------------------------------------------------------

## mtcars with some factors
d <- data.frame(mpg=mtcars$mpg,lapply(mtcars[, c("cyl", "vs", "carb")], as.factor))

## varpro on training data
o <- varpro(mpg~., d[1:20,])
```

```
## predicted values on test data
print(predict(o, d[-(1:20),]))

## predicted values on bad test data with strange factor values
dbad <- d[-(1:20),]
dbad$carb <- as.character(dbad$carb)
dbad$carb <-  sample(LETTERS, size = nrow(dbad))
print(predict(o, dbad))
```

---

uvarpro *Unsupervised Variable Selection using Variable Priority (UVarPro)*

Description

Performs unsupervised variable selection by extending the VarPro framework to forests grown without labels. UVarPro identifies features that explain structure in the data through region-release contrasts, with importance assessed using entropy or lasso-based methods.

Usage

```
uvarpro(data,
        method = c("auto", "unsupv", "rnd"),
        ntree = 200, nodesize = NULL,
        max.rules.tree = 20, max.tree = 200,
        verbose = FALSE, seed = NULL,
        ...)
```

Arguments

data	Data frame containing the unsupervised data.
method	Type of forest used. Options are "auto" (auto-encoder), "unsupv" (unsupervised analysis), and "rnd" (pure random forest).
ntree	Number of trees to grow.
nodesize	Minimum terminal node size. If not specified, an internal function selects an appropriate value based on sample size and dimension.
max.rules.tree	Maximum number of rules per tree.
max.tree	Maximum number of trees used to extract rules.
verbose	Print verbose output?
seed	Seed for reproducibility.
...	Additional arguments passed to rfsrc.

Details

UVarPro performs unsupervised variable selection by applying the VarPro framework to random forests trained on unlabeled data (Zhou et al., 2025). The procedure has two components: (i) the construction of an unsupervised forest, and (ii) the evaluation of variable importance based on region-release contrasts, in direct analogy to the supervised setting in VarPro.

The forest construction is controlled by the `method` argument. By default, `method = "auto"` fits a random forest autoencoder, which regresses each selected variable on itself, a specialized form of multivariate forest modeling. Alternatives include `"unsupv"`, which uses pseudo-responses and multivariate splits to build an unsupervised forest (Tang and Ishwaran, 2017), and `"rnd"`, which uses completely random splits. For large datasets, the autoencoder may be slower, while `"unsupv"` and `"rnd"` are often much faster.

Variable importance is assessed using region-release contrasts formed by the forest. By default, the `importance` function returns an entropy-based criterion. This measure compares the variability within each region to the variability across the combined sample, effectively acting like a ratio of between to within sums of squares. Importance values are averaged across many region-release rules, providing a rough but fast estimate of how strongly a feature contributes to distinguishing regions. See examples below.

In addition to this default entropy measure, UVarPro supports custom user-defined entropy functions to create alternative importance metrics.

A more sophisticated procedure, described in Zhou et al. (2026), reframes each region-release contrast as a supervised classification task, with membership in the region serving as the class label. Variable effects are estimated using lasso-based logistic regression, and coefficients are aggregated over the collection of region-release tasks. This produces a sparser and often more interpretable assessment of importance compared to the entropy method. Although more computationally intensive, the lasso-driven approach can provide sharper separation of relevant and irrelevant features. See examples below for details.

Value

A uvarpro object.

Author(s)

Min Lu and Hemant Ishwaran

References

Tang F. and Ishwaran H. (2017). Random forest missing data algorithms. *Statistical Analysis and Data Mining*, 10:363-377.

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

See Also

`varpro`

Examples

```
## ------------------------------------------------------------
## toy example - needed to pass CRAN test
## ------------------------------------------------------------

## mtcars unsupervised regression
o <- uvarpro(mtcars, ntree = 1)



## ------------------------------------------------------------
## boston housing: default call
## ------------------------------------------------------------

data(BostonHousing, package = "mlbench")

## default call
o <- uvarpro(BostonHousing)
print(importance(o))

## ------------------------------------------------------------
## boston housing: using method="unsupv"
## ------------------------------------------------------------

data(BostonHousing, package = "mlbench")

## unsupervised splitting
o <- uvarpro(BostonHousing, method = "unsupv")
print(importance(o))

## ------------------------------------------------------------
## boston housing: illustrates hot-encoding
## ------------------------------------------------------------

## load the data
data(BostonHousing, package = "mlbench")

## convert some of the features to factors
Boston <- BostonHousing
Boston$zn <- factor(Boston$zn)
Boston$chas <- factor(Boston$chas)
Boston$lstat <- factor(round(0.2 * Boston$lstat))
Boston$nox <- factor(round(20 * Boston$nox))
Boston$rm <- factor(round(Boston$rm))

## call unsupervised varpro and print importance
print(importance(o <- uvarpro(Boston)))

## get top variables
get.topvars(o)
```

```
## map importance values back to original features
print(get.orgvimp(o))

## same as above ... but for all variables
print(get.orgvimp(o, pretty = FALSE))


## ------------------------------------------------------------
## latent variable simulation
## ------------------------------------------------------------

n <- 1000
w <- rnorm(n)
x <- rnorm(n)
y <- rnorm(n)
z <- rnorm(n)
ei <- matrix(rnorm(n * 20, sd = sqrt(.1)), ncol = 20)
e21 <- rnorm(n, sd = sqrt(.4))
e22 <- rnorm(n, sd = sqrt(.4))
wi <- w + ei[, 1:5]
xi <- x + ei[, 6:10]
yi <- y + ei[, 11:15]
zi <- z + ei[, 16:20]
h1 <- w + x + e21
h2 <- y + z + e22
dta <- data.frame(w=w,wi=wi,x=x,xi=xi,y=y,yi=yi,z=z,zi=zi,h1=h1,h2=h2)

## default call
print(importance(uvarpro(dta)))


## ------------------------------------------------------------
## glass (remove outcome)
## ------------------------------------------------------------

data(Glass, package = "mlbench")

## remove the outcome
Glass$Type <- NULL

## get importance
o <- uvarpro(Glass)
print(importance(o))

## compare to PCA
(biplot(prcomp(o$x, scale = TRUE)))

## ------------------------------------------------------------
## iowa housing - illustrates lasso importance
## ------------------------------------------------------------

## first we roughly impute the data
data(housing, package = "randomForestSRC")
```

```
## to speed up analysis, convert all factors to real values
iowa <- roughfix(housing)
iowa <- data.frame(data.matrix(iowa))

## canonical call
o <- uvarpro(iowa)

## standard importance
print(importance(o))

## lasso importance
beta <- get.beta.entropy(o)
print(beta)
print(sort(colMeans(beta, na.rm=TRUE), decreasing = TRUE))

## s-dependent graph
sdependent(beta)

## lasso importance without pre-filtering
## beta.nof <- get.beta.entropy(o, pre.filter = FALSE)
## print(beta.nof)
## print(sort(colMeans(beta.nof, na.rm=TRUE), decreasing = TRUE))

## lasso importance with second stage sparsity lasso
## beta.sparse <- get.beta.entropy(o, second.stage = TRUE)
## print(beta.sparse)


## ------------------------------------------------------------
##  custom importance
##  OPTION 1: use hidden entropy option
## ------------------------------------------------------------

my.entropy <- function(xC, xO, ...) {

  ## xC     x feature data from complementary region
  ## xO     x feature data from original region
  ## ...    used to pass aditional options (required)

  ## custom importance value
  wss <- mean(apply(rbind(xO, xC), 2, sd, na.rm = TRUE))
  bss <- (mean(apply(xC, 2, sd, na.rm = TRUE)) +
              mean(apply(xO, 2, sd, na.rm = TRUE)))
  imp <- 0.5 * bss / wss

  ## entropy value must contain complementary and original membership
  entropy <- list(comp = list(...)$compMembership,
                  oob = list(...)$oobMembership)

  ## return importance and in the second slot the entropy list
  list(imp = imp, entropy)
```

```
o <- uvarpro(BostonHousing, entropy=my.entropy)
print(importance(o))


## ------------------------------------------------------------
##  custom importance
##  OPTION 2: direct importance without hidden entropy option
## ------------------------------------------------------------

o <- uvarpro(BostonHousing, ntree=3, max.rules.tree=10)

## convert original/release region into two-class problem
## define importance as the lasso beta values

## For faster performance on Unix systems, consider using:
## library(parallel)
## imp <- do.call(rbind, mclapply(seq_along(o$entropy), function(j) { ... }))

imp <- do.call(rbind, lapply(seq_along(o$entropy), function(j) {
  rO <- do.call(rbind, lapply(o$entropy[[j]], function(r) {
    xC <- o$x[r[[1]],names(o$entropy),drop=FALSE]
    xO <- o$x[r[[2]],names(o$entropy),drop=FALSE]
    y <- factor(c(rep(0, nrow(xC)), rep(1, nrow(xO))))
    x <- rbind(xC, xO)
    x <- x[, colnames(x) != names(o$entropy)[j]]
    fit <- tryCatch(
      suppressWarnings(glmnet::cv.glmnet(as.matrix(x), y, family = "binomial")),
      error = function(e) NULL
    )
    if (!is.null(fit)) {
      beta <- setNames(rep(0, length(o$entropy)), names(o$entropy))
      bhat <- abs(coef(fit)[-1, 1])
      beta[names(bhat)] <- bhat
      beta
    } else {
      NULL
    }
  }))
  if (!is.null(rO)) {
    val <- colMeans(rO, na.rm = TRUE)
    names(val) <- colnames(rO)
    return(val)
  } else {
    return(NULL)
  }
}) |> setNames(names(o$entropy)))

print(imp)


## ------------------------------------------------------------
##  custom importance
##  OPTION 3: direct importance using built in lasso beta function
```

```
## ------------------------------------------------------------

o <- uvarpro(BostonHousing)
print((get.beta.entropy(o)))

}
```

---

varpro *Model-Independent Variable Selection via the Rule-based Variable Priority (VarPro)*

Description

Model-Independent Variable Selection via the Rule-based Variable Priority (VarPro) for Regression, Multivariate Regression, Classification and Survival.

Usage

```
varpro(formula, data, nvar = 30, ntree = 500,
       split.weight = TRUE, split.weight.method = NULL, sparse = TRUE,
       nodesize = NULL, max.rules.tree = 150, max.tree = min(150, ntree),
      parallel = TRUE, cores = get.mc.cores(), verbose = FALSE, seed = NULL, ...)
```

Arguments

formula	Formula specifying the model to be fit.
data	Data frame containing the training data.
nvar	Maximum number of variables to return.
ntree	Number of trees to grow.
split.weight	Use guided tree-splitting? Candidate variables for splitting are selected with probability proportional to a split-weight, obtained by default from a preliminary lasso+tree step.
split.weight.method	Character string (or vector) specifying method used to generate split-weights. Defaults to lasso+tree. See Details.
sparse	Use sparse split-weights?
nodesize	Minimum terminal node size. If not specified, value is set internally based on sample size and dimension.
max.rules.tree	Maximum number of rules per tree.
max.tree	Maximum number of trees used to extract rules.
parallel	Use parallel execution for lasso folds using **doMC**.
cores	Number of cores for parallel processing. Defaults to parallel::detectCores().
verbose	Print verbose output?
seed	Seed for reproducibility.
...	Additional arguments for advanced use.

Details

Rule-based models, such as decision rules, rule learning, trees, boosted trees, Bayesian additive regression trees, Bayesian forests, and random forests, are widely used for variable selection. These nonparametric methods require no model specification and accommodate various outcomes including regression, classification, survival, and longitudinal data.

Although permutation variable importance (VIMP) and knockoff methods have been extensively studied, their effectiveness can be limited in practice. Both approaches rely on the quality of artificially generated covariates, which may not perform well in complex or high-dimensional settings.

To address these limitations, we introduce a new framework called variable priority (VarPro). Instead of generating synthetic covariates, VarPro constructs *release rules* to assess the impact of each covariate on the response. Neighborhoods of existing data are used for estimation, avoiding the need for artificial data generation. Like VIMP and knockoffs, VarPro imposes no assumptions on the conditional distribution of the response.

The VarPro algorithm proceeds as follows: A forest of `ntree` trees is grown using guided tree-splitting, where candidate variables for node splitting are selected with probability proportional to their split-weights. These split-weights are computed in a preprocessing step. A subset of `max.tree` trees is randomly selected from the forest, and `max.rules.tree` branches are sampled from each selected tree. The resulting rules form the basis of the VarPro importance estimator. The method supports regression, multivariate regression, multiclass classification, and survival outcomes.

Guided tree-splitting encourages rule construction to favor influential features. Thus, `split.weight` should generally remain `TRUE`, especially for high-dimensional problems. If disabled, it is recommended to increase `nodesize` to improve estimator precision.

By default, split-weights are computed via a lasso-plus-tree strategy. Specifically, the split-weight of a variable is defined as the product of the absolute standardized lasso coefficient and the variable's split frequency from a forest of shallow trees. If sample size and dimension are both moderate, this may be replaced by the variable's absolute permutation importance. Note: variables are one-hot encoded for use in lasso, and all inputs are converted to numeric values.

To customize split-weight construction, use the `split.weight.method` argument with one or more of the following strings: `"lasso"`, `"tree"`, or `"vimp"`. For example, `"lasso"` uses only lasso coefficients; `"lasso tree"` combines lasso and shallow trees; `"lasso vimp"` combines lasso with permutation importance. See examples below.

Variables are ranked by importance, with higher values indicating greater influence. Cross-validation can be used to determine a cutoff threshold. See `cv.varpro` for details.

Run time can be reduced by using smaller values of `ntree` or larger values of `nodesize`. Additional runtime tuning options are discussed in the examples.

In class-imbalanced two-class settings, the algorithm automatically switches to random forest quantile classification (RFQ; see O'Brien and Ishwaran, 2019) using the geometric mean (gmean) metric. This behavior can be overridden via the hidden option `use.rfq`.

Value

Output containing VarPro estimators used to calculate importance. See `importance.varpro`. Also see `cv.varpro` for automated variable selection.

Author(s)

Min Lu and Hemant Ishwaran

References

Ishwaran H. (2025). *Multivariate Statistics: Classical Foundations and Modern Machine Learning*. Chapman and Hall/CRC.

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

Lu, M. and Ishwaran, H. (2024). Model-independent variable selection via the rule-based variable priority. arXiv e-prints, pp.arXiv-2409.

Lu, M. and Ishwaran, H. (2025). Individual variable priority: a model-independent local gradient method for variable importance. *Artificial Intelligence Review*, 58:407.

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

See Also

alzheimers cv.varpro glioma hrrecov importance.varpro partialpro predict.varpro isopro ivarpro outpro uvarpro

Examples

```
## ------------------------------------------------------------
## toy example - needed to pass CRAN test
## ------------------------------------------------------------

## mtcars regression
o <- varpro(mpg ~ ., mtcars, ntree = 1)



## ------------------------------------------------------------
## classification example: iris
## ------------------------------------------------------------

## apply varpro to the iris data
o <- varpro(Species ~ ., iris, max.tree = 5)

## call the importance function and print the results
print(importance(o))

## ------------------------------------------------------------
## regression example: boston housing
## ------------------------------------------------------------

## load the data
data(BostonHousing, package = "mlbench")
```

```
## call varpro
o <- varpro(medv~., BostonHousing)

## extract and print importance values
imp <- importance(o)
print(imp)

## another way to extract and print importance values
print(get.vimp(o))
print(get.vimp(o, pretty = FALSE))

## plot importance values
importance(o, plot.it = TRUE)

## ------------------------------------------------------------
## regression example: boston housing illustrating hot-encoding
## ------------------------------------------------------------

## load the data
data(BostonHousing, package = "mlbench")

## convert some of the features to factors
Boston <- BostonHousing
Boston$zn <- factor(Boston$zn)
Boston$chas <- factor(Boston$chas)
Boston$lstat <- factor(round(0.2 * Boston$lstat))
Boston$nox <- factor(round(20 * Boston$nox))
Boston$rm <- factor(round(Boston$rm))

## call varpro and print the importance
print(importance(o <- varpro(medv~., Boston)))

## get top variables
get.topvars(o)

## map importance values back to original features
print(get.orgvimp(o))

## same as above ... but for all variables
print(get.orgvimp(o, pretty = FALSE))

## ------------------------------------------------------------
## regression example: friedman 1
## ------------------------------------------------------------

o <- varpro(y~., data.frame(mlbench::mlbench.friedman1(1000)))
print(importance(o))

## ------------------------------------------------------------
## example without guided tree-splitting
## ------------------------------------------------------------
```

```
o <- varpro(y~., data.frame(mlbench::mlbench.friedman2(1000)),
            nodesize = 10, split.weight = FALSE)
print(importance(o))

## ------------------------------------------------------------
## regression example: all noise
## ------------------------------------------------------------

x <- matrix(rnorm(100 * 50), 100, 50)
y <- rnorm(100)
o <- varpro(y~., data.frame(y = y, x = x))
print(importance(o))

## ------------------------------------------------------------
## multivariate regression example: boston housing
## ------------------------------------------------------------

data(BostonHousing, package = "mlbench")

## using rfsrc multivariate formula call
importance(varpro(Multivar(lstat, nox) ~., BostonHousing))

## using cbind multivariate formula call
importance(varpro(cbind(lstat, nox) ~., BostonHousing))

##------------------------------------------------------------------
##  class imbalanced problem
##
## - simulation example using the caret R-package
## - creates imbalanced data by randomly sampling the class 1 values
##
##------------------------------------------------------------------

if (library("caret", logical.return = TRUE)) {

  ## experimental settings
  n <- 5000
  q <- 20
  ir <- 6
  f <- as.formula(Class ~ .)

  ## simulate the data, create minority class data
  d <- twoClassSim(n, linearVars = 15, noiseVars = q)
  d$Class <- factor(as.numeric(d$Class) - 1)
  idx.0 <- which(d$Class == 0)
  idx.1 <- sample(which(d$Class == 1), sum(d$Class == 1) / ir , replace = FALSE)
  d <- d[c(idx.0,idx.1),, drop = FALSE]
  d <- d[sample(1:nrow(d)), ]

  ## varpro call
  print(importance(varpro(f, d)))

}
```

```
## ------------------------------------------------------------
## multiclass example: wine data
## ------------------------------------------------------------
data(wine, package = "randomForestSRC")
wine$quality <- factor(wine$quality)
o <- varpro(quality~., wine)
print(importance(o))

## ------------------------------------------------------------
## survival example: pbc
## ------------------------------------------------------------
data(pbc, package = "randomForestSRC")
o <- varpro(Surv(days, status)~., pbc)
print(importance(o))

## ------------------------------------------------------------
## pbc survival with rmst (restricted mean survival time)
## functional of interest is RMST at 500 days
## ------------------------------------------------------------
data(pbc, package = "randomForestSRC")
o <- varpro(Surv(days, status)~., pbc, rmst = 500)
print(importance(o))

## ------------------------------------------------------------
## pbc survival with rmst vector
## variable importance is a list for each rmst value
## ------------------------------------------------------------
data(pbc, package = "randomForestSRC")
o <- varpro(Surv(days, status)~., pbc, rmst = c(500, 1000))
print(importance(o))

## ------------------------------------------------------------
## survival example with more variables
## ------------------------------------------------------------
data(peakVO2, package = "randomForestSRC")
o <- varpro(Surv(ttodead, died)~., peakVO2)
imp <- importance(o, plot.it = TRUE)
print(imp)

## ------------------------------------------------------------
## high dimensional survival example
## ------------------------------------------------------------
data(vdv, package = "randomForestSRC")
o <- varpro(Surv(Time, Censoring)~., vdv)
print(importance(o))

## ------------------------------------------------------------
## high dimensional survival example without sparse option
## ------------------------------------------------------------
data(vdv, package = "randomForestSRC")
o <- varpro(Surv(Time, Censoring)~., vdv, sparse = FALSE)
print(importance(o))
```

```
## ------------------------------------------------------------------------
## high dimensional survival example using different split-weight methods
## ------------------------------------------------------------------------
data(vdv, package = "randomForestSRC")
f <- as.formula(Surv(Time, Censoring)~.)

## lasso only
print(importance(varpro(f, vdv, split.weight.method = "lasso")))

## lasso and vimp
print(importance(varpro(f, vdv, split.weight.method = "lasso vimp")))

## lasso, vimp and shallow trees
print(importance(varpro(f, vdv, split.weight.method = "lasso vimp tree")))

## ------------------------------------------------------------
## largish data (iowa housing data)
## to speed up calculations convert data to all real
## ------------------------------------------------------------

## first we roughly impute the data
data(housing, package = "randomForestSRC")
dta <- roughfix(housing)
dta <- data.frame(data.matrix(dta))

## varpro call
o <- varpro(SalePrice~., dta)
print(importance(o))

## ------------------------------------------------------------
## large data: illustrates different ways to improve speed
## ------------------------------------------------------------

n <- 25000
p <- 50
d <- data.frame(y = rnorm(n), x = matrix(rnorm(n * p), n))

## use large nodesize
print(system.time(o <- varpro(y~., d, ntree = 100, nodesize = 200)))
print(importance(o))

## use large nodesize, smaller bootstrap
print(system.time(o <- varpro(y~., d, ntree = 100, nodesize = 200,
           sampsize = 100)))
print(importance(o))


## ------------------------------------------------------------
## custom split-weights (hidden option)
## ------------------------------------------------------------

## load the data
```

```
data(BostonHousing, package = "mlbench")

## make some features into factors
Boston <- BostonHousing
Boston$zn <- factor(Boston$zn)
Boston$chas <- factor(Boston$chas)
Boston$lstat <- factor(round(0.2 * Boston$lstat))
Boston$nox <- factor(round(20 * Boston$nox))
Boston$rm <- factor(round(Boston$rm))

## get default custom split-weights: a named real vector
swt <- get.splitweight.custom(medv~.,Boston)

## define custom splits weight
swt <- swt[grepl("crim", names(swt)) |
           grepl("zn", names(swt)) |
           grepl("nox", names(swt)) |
           grepl("rm", names(swt)) |
           grepl("lstat", names(swt))]

swt[grepl("nox", names(swt))] <- 4
swt[grepl("lstat", names(swt))] <- 4

swt <- c(swt, strange=99999)

cat("custom split-weight\n")
print(swt)

## call varpro with the custom split-weights
o <- varpro(medv~.,Boston,split.weight.custom=swt,verbose=TRUE,sparse=FALSE)
cat("varpro result\n")
print(importance(o))
print(get.vimp(o, pretty=FALSE))
print(get.orgvimp(o, pretty=FALSE))
```

`varpro.news` *Show the NEWS file*

Description

Show the NEWS file of the **varPro** package.

Usage

```
varpro.news(...)
```

Arguments

`...`	Further arguments passed to or from other methods.

Value

None.

Author(s)

Min Lu and Hemant Ishwaran

`varpro.strength` *Obtain Strength Array and Other Values from a VarPro Object*

Description

Used to parse values from a VarPro object.

Usage

```
varpro.strength(object,
                newdata,
                m.target = NULL,
                max.rules.tree = 150,
                max.tree = 150,
                stat = c("importance", "complement", "oob", "none"),
                membership = FALSE,
                neighbor = 5,
                seed = NULL,
                do.trace = FALSE, ...)
```

Arguments

`object`	`rfsrc` object.
`newdata`	Optional test data. If provided, returns branch and complementary branch membership of the training data corresponding to the test cases.
`m.target`	Character string specifying the target outcome for multivariate families. If unspecified, a default is selected automatically.
`max.rules.tree`	Maximum number of rules extracted per tree.
`max.tree`	Maximum number of trees used for rule extraction.
`stat`	Statistic to output. Options include `"importance"`, `"complement mean"`, and `"oob mean"`.
`membership`	Return out-of-bag and complementary membership indices for each rule?
`neighbor`	Nearest neighbor parameter, used only when `newdata` is specified.
`seed`	Seed for reproducibility.
`do.trace`	Enable detailed trace output.
`...`	Additional arguments.

Details

Not intended for direct end-user use; primarily designed for internal package operations.

Value

Object coerced so as to work with other functions in the package.

Examples

```
## ------------------------------------------------------------
## regression example: boston housing
## ------------------------------------------------------------

## load the data
data(BostonHousing, package = "mlbench")

o <- randomForestSRC::rfsrc(medv~., BostonHousing, ntree=100)

## call varpro.strength
varpro.strength(object = o, max.rules.tree = 10, max.tree = 15)

## call varpro.strength with test data
varpro.strength(object = o, newdata = BostonHousing[1:3,], max.rules.tree = 10, max.tree = 15)
```

Index

∗ **cv.varpro**
  cv.varpro, 4
∗ **datasets**
  alzheimers, 2
  gliomas, 8
  hrrecov, 9
∗ **documentation**
  varpro.news, 65
∗ **hplot**
  plot.ivarpro, 36
∗ **importance**
  plot.ivarpro, 36
∗ **individual importance**
  ivarpro, 19
  predict.ivarpro, 45
∗ **outlier**
  isopro, 15
∗ **plot**
  partialpro, 29
  plot.partialpro, 40
∗ **predict outlier**
  predict.isopro, 43
∗ **predict uvarpro**
  predict.uvarpro, 48
∗ **predict varpro**
  predict.varpro, 50
∗ **uvarpro**
  uvarpro, 52
∗ **varpro.strength**
  varpro.strength, 66
∗ **varpro**
  importance, 12
  varpro, 58

alzheimers, 2, *60*

cv.varpro, 4, *13*, *60*

glioma, *60*
glioma *(*gliomas*)*, 8

gliomas, 8

hrrecov, 9, *60*

importance, 12
importance.varpro, *5*, *60*
isopro, 15, *33*, *44*, *60*
ivarpro, 19, *37, 38*, *45*, *47*, *60*

outpro, 25, *33*, *60*

partialpro, 29, *41*, *60*
plot.ivarpro, *23*, 36, *46, 47*
plot.partialpro, 40
predict.isopro, *17*, 43
predict.ivarpro, *37*, 45
predict.uvarpro, 48
predict.varpro, 50, *60*

rfsrc, *28*

shap.ivarpro, *38*

uvarpro, *5*, *13*, *17*, *44*, *49*, 52, *60*

varpro, *5*, *13*, *17*, *23*, *28*, *33*, *38*, *44*, *47*, *50*, *53*, 58
varpro.news, 65
varpro.strength, 66