Package 'varPro'

September 22, 2026

Version 3.3.0

Date 2026-09-22

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 2026-09-22 15:50:02 UTC

Contents

`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` *Select VarPro Predictors and Assess Selection by Cross-Validation*

Description

Selects predictors by comparing the prediction error of random forests built from candidate VarPro variable sets. Returns the minimum-error selection and conservative and liberal alternatives. Optional outer cross-validation assesses predictive performance and the stability of variable selection.

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,
          cv.folds = 0, foldid = NULL, ...)
```

Arguments

`formula`	Formula specifying the response and predictors. The supported outcome families are those of `varpro`: regression, multivariate regression, classification, and right-censored survival. See Details for multivariate scoring.
`data`	Training data frame. Incomplete observations are removed before ranking predictors and evaluating candidate models.
`nvar`	Maximum number of processed predictor columns retained by the preliminary split-weight screening in `varpro`. Candidate models are subsequently formed on the original-variable scale. Ignored by the screening step when computed and custom split-weights are both absent.
`ntree`	Number of trees in the VarPro rule forest, each candidate prediction forest, and each outer assessment forest.
`local.std`	Should locally standardized rule comparisons be used to calculate importance? See `varpro`.
`zcut`	Numeric vector of positive importance cutoffs. Predictors with `z >= zcut` enter a candidate model. Cutoffs are sorted into increasing order, and each distinct variable set is evaluated once.
`nblocks`	Positive integer controlling the subdivision of each candidate forest for estimating prediction error and its variation. Larger values give smaller subforests. Use `nblocks = 1` to evaluate the whole forest. See Details.
`split.weight`	Should preliminary split-weights guide the VarPro screening and rule generation? See `varpro`.
`split.weight.method`	Preliminary weighting methods passed to `varpro`, for example `c("lasso", "tree")`. The default chooses the combination automatically.

sparse	Should preliminary weighting concentrate more strongly on promising predictors? Passed to varpro.
nodesize	Minimum terminal node size for the VarPro forest and, with fast = FALSE, the candidate prediction forests. Chosen automatically when NULL. Fast candidate forests use the settings of rfsrc.fast.
max.rules.tree, max.tree	Rule-extraction limits passed to varpro. They determine the comparisons available for the importance ranking.
verbose	Should progress and candidate-model performance summaries be printed?
seed	Seed for VarPro, candidate forests, and optional outer cross-validation. Also call set.seed before cv.varpro to control all random sampling.
fast	Should candidate prediction forests use randomForestSRC::rfsrc.fast? This uses smaller per-tree samples and a shared holdout for evaluation when sufficient data are available. See Details.
crps	For survival, should candidate models be evaluated using a time-normalized integrated censoring-weighted Brier score? With FALSE, the forest's default concordance error is used. See Details.
cv.folds	Number of outer cross-validation folds. The default, zero, uses the blocked-OOB procedure alone. A value of at least two adds held-out performance and selection stability summaries for all three rules. Generated folds are stratified by response class or survival event status, as appropriate. The fold count is reduced with a warning when the data cannot support the requested number. This is separate from nblocks and the preliminary lasso's nfolds option.
foldid	Optional vector of fold labels, one per input data row, to supply the outer partition. Retained observations must have consecutive integer labels starting at one. Entries for omitted observations are ignored. Supplying foldid enables outer CV even when cv.folds = 0; a positive cv.folds must match the number of supplied folds. Use common labels to keep records from the same subject together.
...	Additional named arguments passed to varpro, including parallel, cores, nfolds, rmst, and external-forest controls. The sampsize argument also controls candidate-forest sampling. See Details.

Details

Variable ranking and selection: VarPro importance summarizes response differences between rule regions and their near-miss sets; see varpro. At each cutoff in zcut, predictors with scores at or above the cutoff form a candidate variable set. Higher cutoffs retain fewer variables. A prediction forest is used to evaluate each distinct set.

Selection is reported for the original variables. For a hot-encoded categorical predictor, its score is the largest importance among its encoded columns. Multivariate regression also takes the maximum across responses; classification uses overall importance. Prediction forests use the original outcome and predictors, so selecting a categorical variable retains all its levels.

Prediction error and the three selection rules: By default, candidate models are compared using out-of-bag predictions. Each forest is evaluated in blocks of trees, giving a mean prediction

error and a standard deviation across blocks. The error path is used to choose an importance cutoff. With `nblocks = 1`, the whole forest is evaluated and the block deviation is zero.

The three rules offer different balances between prediction accuracy and the number of selected variables:

Minimum error The set with the lowest mean error, returned in `imp`.

Conservative The smallest set within the error tolerance whose error is also below 1, returned in `imp.conserve`.

Liberal The largest set within the error tolerance, returned in `imp.liberal`.

The tolerance is the minimum mean error plus the average block standard deviation across candidate models. This allows smaller and larger sets with similar predictive performance.

In OOB-only mode, `imp.conserve` is `NULL` when no model meets the conservative criterion. With outer cross-validation, an empty selection uses the highest-ranked variable. If no candidate model has a finite error, a warning is issued and all three rules return the unfiltered ranking with cutoffs set to zero.

Performance measures: Lower error is better for every supported family.

Regression Mean squared prediction error divided by the response variance. Multivariate regression uses the first response for prediction error and combines responses for the importance ranking.

Classification Normalized Brier prediction error. For imbalanced two-class data, RFQ uses geometric-mean error, $1-\sqrt{\text{sensitivity} \times \text{specificity}}$. The controls `use.rfq` and `iratio.threshold` apply to both variable ranking and candidate-model evaluation.

Survival, `crps = FALSE` Concordance error, $1 - C$, from the survival forest.

Survival, `crps = TRUE` A censoring-weighted integrated Brier score for survival probabilities, divided by the maximum evaluation time.

Supplying `rmst` changes the survival summary used for the VarPro ranking. Prediction error is evaluated against the original survival outcome using concordance error or CRPS.

Outer cross-validation: Set `cv.folds = 5`, for example, to assess how well the complete variable-selection procedure predicts held-out observations. Variable selection and model training use the remaining folds. All three selection rules are assessed using the same outcome-specific scoring as the main analysis.

The returned `imp`, `imp.conserve`, and `imp.liberal` tables contain the full-data selections and importance scores. Additional columns show how often each variable was selected and how its importance varied across training samples. Held-out prediction errors are available in each table's `cv` attribute; see Value and the `peakVO2` example.

Outer cross-validation requires additional computation because the analysis is repeated for each training sample. Supply `foldid` to use a particular partition, for example to keep records from the same subject together.

Computational controls: `fast = TRUE` uses `rfsrc.fast` for candidate prediction forests. Evaluation uses a shared holdout when sufficient data are available and out-of-bag predictions otherwise. Variable ranking uses all observations available to that selection analysis. Supply `sampsize` to control per-tree sample size.

The `parallel` and `cores` arguments control the preliminary lasso calculations. Forest threading is controlled by **randomForestSRC**.

Value

An object of class "cv.varpro" containing:

imp — Original-variable importance table for the minimum-error selection, with columns variable and z.

imp.conserve — Corresponding conservative selection table.

imp.liberal — Corresponding liberal selection table.

err — Prediction-error path with columns zcut, nvar, err, and sd: the importance cutoff, number of selected original predictors, mean block error, and standard deviation across tree blocks.

zcut, zcut.conserve, zcut.liberal — Cutoffs for the three full-data selections. See Details for fallback behavior.

With outer cross-validation, variable and z retain their full-data values and four columns are appended to each importance table:

cv.select Percentage of outer folds in which the variable was selected under that table's rule, including top-variable fallbacks. A variable excluded during preliminary screening counts as unselected.

cv.z.mean Mean importance score across outer training samples. Uses all available scores, including those from folds where the variable was not selected.

cv.z.sd Sample standard deviation of those importance scores. Larger values indicate greater variation in importance across training samples. It is NA when fewer than two scores are available.

cv.z.n Number of training-fold importance scores contributing to cv.z.mean and cv.z.sd. This can be smaller than the number of folds when preliminary screening excludes a variable or its score is unavailable.

These importance summaries use finite scores before applying the selection cutoff. With five folds, a variable scored in every fold and selected in two has cv.z.n = 5 and cv.select = 40. The mean is NA when no score is available.

Each importance table has a cv attribute containing its mean held-out error (err), standard deviation of fold errors (sd), performance measure (perf.type), and per-fold results (folds). It also records selection percentages for all original predictors (selection.frequency) and whether the full-data selection used the top-variable fallback (fallback). For example, use attr(o$imp, "cv") to inspect the minimum-error rule's assessment. Fold assignments are available through attr(o, "cv")$foldid, in input-row order, with NA for omitted observations.

Attributes also retain the processed-column importance object (imp.org), processed and original predictor names (xvar.names, xvar.org.names), and the VarPro forest family (family). Use get.vimp(o) to inspect processed-column scores and get.orgvimp(o, pretty = FALSE) for original-variable scores with unselected variables filled by zero.

The print method displays the selections and a compact outer-CV performance summary when available. To construct a final prediction model, train the chosen learner using a returned variable set, as shown in Examples.

Note

For serial lasso calculations, supply `parallel = FALSE`. Outer error summaries are `NA` when a fold error is undefined or folds use different performance measures. Inspect `attr(o$imp, "cv")$folds` for the individual fold results.

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:2409.09003. doi:10.48550/arXiv.2409.09003.

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

See Also

`varpro`, `importance.varpro`, `predict.varpro`, `randomForestSRC::rfsrc`, `randomForestSRC::rfsrc.fast`.

Examples

```
## Nonlinear regression with a categorical effect and known signal set.
set.seed(137)
n <- 500
x <- matrix(runif(n * 10), nrow = n)
colnames(x) <- sprintf("x%02d", seq_len(ncol(x)))
group <- factor(sample(c("low", "middle", "high"), n, replace = TRUE),
                levels = c("low", "middle", "high"))
mu <- 10 * sin(pi * x[, 1] * x[, 2]) +
      20 * (x[, 3] - 0.5)^2 + 10 * x[, 4] + 5 * x[, 5] +
      3 * (group == "high")
d <- data.frame(y = mu + rnorm(n), x, group)
signal <- c(colnames(x)[1:5], "group")

set.seed(137)
o <- cv.varpro(y ~ ., d, ntree = 80,
               zcut = seq(0.1, 3, length.out = 10), nblocks = 6,
               max.tree = 60, max.rules.tree = 40,
               nfolds = 5, parallel = FALSE)

## Compare the three full-data selections.
print(o$imp)
print(o$imp.conserve)
print(o$imp.liberal)

print(data.frame(
  rule = c("minimum", "conservative", "liberal"),
  cutoff = c(o$zcut, o$zcut.conserve, o$zcut.liberal),
  nvar = c(NROW(o$imp), NROW(o$imp.conserve), NROW(o$imp.liberal))))
print(data.frame(variable = o$imp$variable, z = o$imp$z,
```

```
                signal = o$imp$variable %in% signal))

## Original-variable scores and encoded-column scores at each cutoff.
print(get.orgvimp(o, pretty = FALSE))
print(get.vimp(o, pretty = FALSE))

## Plot the error path. Bars show one tree-block standard deviation.
e <- as.data.frame(o$err)
e <- e[is.finite(e$err) & is.finite(e$sd), , drop = FALSE]
if (nrow(e) > 0) {
  ylim <- range(c(pmax(0, e$err - e$sd), e$err + e$sd))
  plot(e$zcut, e$err, type = "b", pch = 19, ylim = ylim,
       xlab = "Importance cutoff",
       ylab = "Mean tree-block prediction error")
  arrows(e$zcut, pmax(0, e$err - e$sd),
         e$zcut, e$err + e$sd, angle = 90, code = 3, length = 0.04)
  abline(h = min(e$err) + mean(e$sd), lty = 2)
  abline(v = o$zcut, lty = 3)
}

## Train a final learner with the minimum-error variable set.
selected <- o$imp$variable
if (length(selected) > 0 && any(is.finite(o$err[, "err"]))) {
  final <- randomForestSRC::rfsrc(
    y ~ ., d[, c("y", selected), drop = FALSE], ntree = 300)
}

## Outer cross-validation: survival prediction with peakVO2.
## Each fold repeats ranking and cutoff selection on its training data.
library(survival)
data(peakVO2, package = "randomForestSRC")
peak <- na.omit(peakVO2)

set.seed(137)
peak.cv <- cv.varpro(Surv(ttodead, died) ~ ., peak,
                     cv.folds = 5, ntree = 100, seed = 137,
                     zcut = seq(0.1, 3, length.out = 12), nblocks = 6,
                     max.tree = 75, max.rules.tree = 40,
                     nfolds = 5, parallel = FALSE)

## The selections and z scores come from the full-data analysis.
## cv.select gives the percentage of outer folds selecting each variable.
print(peak.cv$imp)
print(peak.cv$imp.conserve)
print(peak.cv$imp.liberal)

## Compare held-out performance for the three rules.
## Error is 1 - C; lower is better. cv.sd is the deviation across folds.
## nvar.full counts predictors selected in the full-data analysis.
rules <- c("imp", "imp.conserve", "imp.liberal")
cv.summary <- do.call(rbind, lapply(rules, function(rule) {
  tab <- peak.cv[[rule]]
  assessment <- attr(tab, "cv")
```

```
  data.frame(rule = rule, nvar.full = nrow(tab),
             cv.err = assessment$err, cv.sd = assessment$sd,
             perf.type = assessment$perf.type)
}))
print(cv.summary)

## Inspect fold errors, variable counts, cutoffs, and fallback status.
print(attr(peak.cv$imp, "cv")$folds)

## Compare selection percentages for all original predictors,
## including those absent from the full-data selections.
cv.select <- sapply(rules, function(rule) {
  attr(peak.cv[[rule]], "cv")$selection.frequency
})
print(cv.select[order(cv.select[, "imp"], decreasing = TRUE),
                , drop = FALSE])

## Check the sizes of the actual outer folds.
print(table(attr(peak.cv, "cv")$foldid))

## Survival: RMST defines the ranking target; CRPS evaluates the
## resulting survival prediction models on their OOB observations.
library(survival)
data(pbc, package = "randomForestSRC")
pbc <- na.omit(pbc)
set.seed(137)
s <- cv.varpro(Surv(days, status) ~ ., pbc,
               rmst = 1000, crps = TRUE, fast = FALSE,
               ntree = 100, nblocks = 6,
               zcut = seq(0.1, 3, length.out = 12),
               max.tree = 75, max.rules.tree = 40,
               nfolds = 5, parallel = FALSE)
print(s$imp)
print(s$err)
```

`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).

`reststst` 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: Case-Specific Local Gradients*

Description

Estimates how a prediction target changes with each predictor near individual training observations. Forest rules define the local comparisons, and local slopes provide signed, case-specific importance scores. Supports regression, classification, and survival analyses.

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	A varpro object, or a supervised rfsrc grow object with a single outcome and numeric predictors. Use a varpro object for categorical predictors and multivariate regression.
adaptive	Automatically limit the largest candidate neighborhood according to the sample size? Used when cut is not supplied.
cut	Optional vector of nonnegative neighborhood widths for continuous predictors, measured in standard deviations of the released-region predictor values. Supplied values are sorted and duplicates removed; adaptive, cut.max, and ncut are then ignored. Binary 0/1 predictors use a comparison of the two levels.
cut.max	Upper bound on candidate neighborhood width when cut is not supplied. Smaller values restrict estimation to more local observations.
ncut	Number of candidate widths when cut is not supplied.
nmin	Minimum number of usable observations for a local estimate.
nmax	Maximum number of observations used for a local estimate. The effective maximum is also limited according to the training sample size and is at least nmin.
y.external	Optional numeric vector or matrix containing the target values to explain. Rows must correspond, in order, to the processed training observations in object. A matrix supplies a separate target in each column; column names identify the targets. The default uses the rule-generating forest's out-of-bag predictions.
noise.na	Represent unavailable local estimates by NA? With TRUE, available rule estimates are averaged. With FALSE, unavailable rule estimates contribute zero, and cases with no applicable rules receive zero.
max.rules.tree	Maximum number of rules examined per tree. Defaults to the setting in a varpro object, or 150 for direct rfsrc input.
max.tree	Maximum number of trees used to obtain rules. Defaults to the setting in a varpro object, or 150 for direct rfsrc input.
use.loo	Select the continuous-predictor neighborhood by leave-one-out prediction error from the local regressions? With FALSE, use the largest valid candidate neighborhood, subject to nmin and nmax.
use.abs	Average absolute rule-level gradients? The default FALSE retains their signs.

`path.store.membership`
: Retain rule and near-miss membership indices for downstream diagnostics? Setting `FALSE` reduces storage while preserving the returned gradients and ordinary plots.

`save.data`
: Store processed predictors and prediction targets for plotting?

`save.model`
: Store the supplied model for downstream use?

`scale`
: Predictor scaling for the local slopes: `"local"` uses the selected neighborhood's spread, `"global"` uses the predictor's training-sample standard deviation, and `"none"` returns slopes per unit of the predictor. See Details.

Details

From forest rules to individual importance: A variable's influence can change across observations. iVarPro summarizes this variation through local slopes of a prediction target, using forest rules to identify relevant observations.

For a rule region R, releasing predictor s removes its restrictions and retains the restrictions on the other predictors. This gives the enlarged region $R^{(s)}$. The additional observations form the near-miss set $C_s = R^{(s)} \setminus R$. The local regression uses observations from both R and C_s, pooling them within the released region.

For a continuous predictor, estimation uses a neighborhood around its mean among the rule's out-of-bag members. A local regression of the target on that predictor supplies a slope. For a binary 0/1 predictor, the slope is the difference in mean target values between levels 1 and 0. Each case receives the average gradient from rules in whose original region it is an out-of-bag member and for which that predictor is released.

Reading the scores: A positive score indicates that larger predictor values locally accompany larger target values; a negative score indicates the opposite direction. Larger magnitudes represent stronger local changes on the chosen scale. Setting `use.abs = TRUE` summarizes strength by averaging magnitudes before opposing rule slopes can cancel.

Scores are reported for the processed predictors in `object$xvar.names`. A hot-encoded factor can therefore have several indicator columns. An unavailable score means that the case has no usable local estimate for that predictor, for example because it lacks applicable rules or sufficient local variation.

Prediction targets: By default, regression explains out-of-bag predicted responses, and classification explains out-of-bag class probabilities. Binary classification returns scores for the first probability column; `attr(x, "target")` identifies the target in the returned object. Multiclass classification returns a named list with one gradient table per class.

The target follows the forest stored in the model. A survival forest supplies mortality predictions. When VarPro uses a multivariate regression forest for RMST targets, each predicted target has its own gradient table. More generally, multivariate VarPro regression returns one table per predicted response.

Use `y.external` to explain another numeric target, such as predictions from a different model or a particular class probability. A supplied matrix retains all of its target columns. Target values must be aligned with the observations retained by the original analysis.

Scaling and locality: With `scale = "local"`, the slope is multiplied by the selected predictor values' spread around the rule-region center. With `scale = "global"`, the multiplier is the predictor's standard deviation across the training observations. Both express the gradient in target

units. With `scale = "none"`, the slope is in target units per predictor unit; for a binary predictor, it is the change from level 0 to level 1.

Neighborhood size controls how locally the slope is estimated. The defaults choose among candidate neighborhoods using local leave-one-out prediction error. Reducing `cut.max` restricts the available neighborhoods; increasing `nmin` requires more observations for each estimate. For binary predictors, both levels are represented in a sample of the released-region observations, subject to the sample-size controls.

Displaying local importance: `shap.ivarpro(x)` gives a beeswarm-style summary of the local gradients. Position shows the gradient and color shows the predictor value. `plot(x, var = "x1")` displays one predictor's gradients against its values. Use `col.var` to color by another variable and examine how the local relationship varies across groups.

For a list of target-specific tables, supply `target` by name or index to either plotting function. Without it, the first target is used with a warning. Stored data are used automatically; with `save.data = FALSE`, supply the corresponding processed data through `dat` to `shap.ivarpro` or `data` to `plot`.

New observations: Use `predict(x, newdata = test)` to obtain gradients for new cases from the stored rule estimates. Supply predictors in their original form; the training hot-encoding is reused. Omitting `newdata` recovers out-of-bag training scores. See `predict.ivarpro` for an example.

Value

An object of class `ivarpro`. A single target produces a numeric data frame with one row per processed training observation and one column per predictor in `object$xvar.names`. Row names retain the processed training-row identifiers. Multiple targets produce a named list of these data frames. `print(x)` summarizes a list; `print(x, full = TRUE)` displays all of its tables.

The `target` attribute identifies the target or targets. When requested, the `data` attribute contains processed predictors and the target values, and the `model` attribute contains the supplied model. Target columns in the stored data are named `y` or `y.<target>`, with suffixes added to avoid name collisions; their names are recorded in `attr(attr(x, "data"), "response.names")`.

The `ivarpro.path` attribute retains neighborhood settings and diagnostics for the selected rule estimates. Membership indices are included when `path.store.membership = TRUE`. Ordinary use and plotting require only the gradient tables and, for plots, their data.

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`, `shap.ivarpro`, `predict.ivarpro`

Examples

```
## Survival: local changes in predicted mortality.
library(survival)
data(peakVO2, package = "randomForestSRC")
peak <- na.omit(peakVO2)

## Keep all predictors available for the local plots.
set.seed(137)
vp <- varpro(Surv(ttodead, died) ~ ., peak,
             split.weight = FALSE, ntree = 100,
             parallel = FALSE)
ivp <- ivarpro(vp)
print(head(ivp))
shap.ivarpro(ivp)

## Exercise time colors the local relationship with peak oxygen uptake.
plot(ivp, var = "peak.vo2", col.var = "interval")

## Specify a fixed maximum neighborhood width.
ivp.local <- ivarpro(vp, adaptive = FALSE, cut.max = 0.5)
plot(ivp.local, var = "peak.vo2", col.var = "interval")

## Interaction example: Model 3 of Lu and Ishwaran (2025).
## The signal is 6*x1*x2.
## Its slopes are 6*x2 for x1 and 6*x1 for x2.
set.seed(137)
n <- 500
X <- matrix(runif(n * 10), nrow = n)
colnames(X) <- paste0("x", seq_len(ncol(X)))
d <- data.frame(y = 6 * X[, 1] * X[, 2] + rnorm(n), X)
vp <- varpro(y ~ ., d, split.weight = FALSE, ntree = 100,
             parallel = FALSE)

set.seed(137)
iv.local <- ivarpro(vp)
set.seed(137)
iv.global <- ivarpro(vp, scale = "global")
set.seed(137)
iv.slope <- ivarpro(vp, scale = "none")

## Compare local magnitudes under the three scaling choices.
scale.summary <- data.frame(
  variable = colnames(iv.local),
  local = colMeans(abs(iv.local), na.rm = TRUE),
  global = colMeans(abs(iv.global), na.rm = TRUE),
  slope = colMeans(abs(iv.slope), na.rm = TRUE))
print(scale.summary)

## Map both slopes over the (x1, x2) plane, as in the paper's
## gradient displays. Each point is a case; color gives its slope.
## The top row shows the true slopes and the bottom row iVarPro estimates.
## scale = "none" puts the estimates in the same units as the derivatives.
```

```
slopes <- cbind("True slope for x1" = 6 * d$x2,
                "True slope for x2" = 6 * d$x1,
                "iVarPro slope for x1" = iv.slope$x1,
                "iVarPro slope for x2" = iv.slope$x2)

slope.map <- function(x1, x2, slopes) {
  ## Use one color scale for all four panels. Positive slopes run
  ## from white through orange to red; negative slopes are blue.
  values <- slopes[is.finite(slopes)]
  limit <- max(1, ceiling(max(abs(values))))
  pal <- c(grDevices::colorRampPalette(c("navy", "white"))(101)[-101],
           grDevices::colorRampPalette(c("white", "orange", "red"))(101))
  slope.col <- function(z) pal[1L + round(100 * (z / limit + 1))]

  op <- par(no.readonly = TRUE)
  on.exit(par(op), add = TRUE)
  layout(matrix(c(1, 2, 3, 4, 5, 5), ncol = 2, byrow = TRUE),
         heights = c(1, 1, 0.3))
  par(mar = c(3.5, 3.5, 2, 1), mgp = c(2, 0.6, 0))
  for (j in seq_len(ncol(slopes))) {
    ok <- is.finite(slopes[, j])
    plot(x1, x2, type = "n", xlim = c(0, 1), ylim = c(0, 1),
         xlab = "x1", ylab = "x2", main = colnames(slopes)[j], asp = 1)
    points(x1[ok], x2[ok], pch = 16, cex = 0.7,
           col = slope.col(slopes[ok, j]))
    points(x1[!ok], x2[!ok], pch = 4, cex = 0.6, col = "grey60")
  }
  par(mar = c(0, 0, 0, 0))
  plot.new()
  at <- seq(-limit, limit, length.out = 7)
  legend("center", legend = format(signif(at, 2), trim = TRUE),
         fill = slope.col(at), ncol = length(at), bty = "n", cex = 0.8,
         title = "Slope (grey crosses: unavailable)")
}
slope.map(d$x1, d$x2, slopes)

## With slopes on the vertical axis, color by the other predictor.
plot(iv.slope, var = "x1", col.var = "x2")
plot(iv.slope, var = "x2", col.var = "x1")

## Explain two supplied targets using the same forest rules.
## Name the columns so targets can be selected explicitly in plots.
targets <- cbind(interaction = 6 * d$x1 * d$x2,
                 linear = 2 * d$x1 - d$x2)
iv.targets <- ivarpro(vp, y.external = targets, scale = "none")
print(iv.targets)
print(head(iv.targets[["linear"]]))
plot(iv.targets, var = "x1", target = "interaction", col.var = "x2")
shap.ivarpro(iv.targets, target = "linear")
```

---

outpro *Out-of-Distribution Scoring with outPro*

Description

outpro assesses how well observations are supported by the training data in a selected set of predictors. With a varpro object, variable priorities guide the selection and weighting of these predictors. Larger distances indicate greater departure from the training data. outpro.null provides a reference distribution for interpreting the distances.

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 varpro object, or an rfsrc grow object with classes c("rfsrc", "grow").
newdata	Data frame of observations to score. For a varpro object, supply predictors in their original form; factor predictors are hot-encoded and aligned to the training predictors automatically. For an rfsrc object, supply the same predictor columns as in the training data; predictors selected for distance calculation must be numeric. The response is not needed. If omitted, the training observations are scored.
neighbor	Number of training neighbors used for each observation. The default is min(n / 10, 5000), where n is the number of training observations. The value is rounded and restricted to the available number of training observations. With distancef = "knn", these are nearest neighbors in the selected predictor subspace. Other distance functions use forest-derived neighbors.
distancef	Distance function. The default, "knn", averages weighted Manhattan distances to the nearest training observations. The alternatives "prod", "euclidean", "mahalanobis", "manhattan", "minkowski", and "kernel" use forest-derived neighborhoods. See Details.

`reduce` Variable selection and weighting. Use `TRUE` for automatic selection using varPro variable priorities, `FALSE` for all predictors with equal weights, a character vector for a specified set of predictors with equal weights, or a named numeric vector to specify predictors and their relative weights. For an `rfsrc` object, `TRUE` uses all predictors with equal weights. See Details for how priorities and supplied weights determine the distance.

`cutoff` Minimum variable-priority `z` value for automatic selection when `object` is a `varpro` object and `reduce = TRUE`. The default is `0.79` when the training predictor representation has at most 250 columns and `0` otherwise. If fewer than two variables meet the threshold, all variables in the priority table are used. This argument controls variable selection; reference percentiles from `outpro.null` can be used to identify unusually large distances.

`max.rules.tree` Maximum number of rules per tree used to construct forest-derived neighborhoods. Ignored when `distancef = "knn"`.

`max.tree` Maximum number of trees used to construct forest-derived neighborhoods. Ignored when `distancef = "knn"`.

`knn.chunk.size` Positive integer giving the number of observations processed together for `distancef = "knn"`. Smaller values reduce temporary memory use. This changes how the calculation is organized, while leaving the score unchanged. Ignored by the other distance functions.

`newdata.xscale` Advanced option for `varpro` objects. Leave `FALSE` for data supplied in their original form. Set `TRUE` only when `newdata` already contains the encoded training predictor columns, as in `object$x`. Training-based standardization is still applied when distances are computed.

`nulldata` Optional data frame representing the population regarded as in distribution. These observations are scored against the training data to obtain a reference distribution of distances. Supply predictors in the same form as `newdata`. If omitted, the training observations provide the reference distances.

`nulldata.xscale`
Advanced option for `varpro` objects. Has the same meaning for `nulldata` as `newdata.xscale` has for `newdata`.

Details

Method:

outPro assesses whether an observation is close to the training data in a predictor subspace chosen for the modeling problem. For a `varpro` object, variable priorities guide both predictor selection and weighting. The resulting distance emphasizes departures in predictors that varPro identifies as important.

The default method, `distancef = "knn"`, standardizes the selected predictors using their training means and standard deviations. It then finds the `neighbor` training observations closest to each scored observation, using weighted Manhattan distance. The score is the mean distance to these neighbors. Smaller scores indicate closer training support; larger scores indicate greater separation. Smaller neighborhoods emphasize immediate local support, while larger neighborhoods average over a broader region of the training data.

Choosing predictors and weights:

Use `reduce = TRUE` for automatic varPro selection based on `cutoff`, or `reduce = FALSE` to assess departures across all predictors with equal weights. A character vector, such as `reduce = c("x1", "x2")`, restricts scoring to those predictors with equal weights.

A named numeric vector specifies predictors and relative weights. Priorities and supplied weights are squared and normalized to sum to one for the distance calculation. For example, `reduce = c(x1 = 2, x2 = 1)` gives distance weights of `0.8` and `0.2`. Supply nonnegative relative weights, with at least one positive value.

Names supplied through `reduce` must match the training predictor columns, including hot-encoded names in `object$x` for `varpro` objects. Predictors with zero or nonfinite training standard deviation are omitted. Inspect `selected.variables` and `distance.args$weights.used` in the result to see the predictors and weights used.

Scoring observations:

Use `outpro(object, newdata = newdata)` to score new observations. The returned `distance` vector contains one score per row, in the same order as `newdata`.

To score the training observations, omit `newdata`. The default KNN method then excludes each observation from its own neighborhood, leaving at most `n - 1` neighbors. Supplying the training data explicitly as `newdata` retains self matches.

Interpreting distances using a reference sample:

`outpro.null` provides an empirical reference distribution for interpreting distances. Supply `nulldata` to use a separate sample representing the population regarded as in distribution, or omit it to use the training observations. The model's training data remain the source of neighbors; `nulldata` supplies the observations whose distances form the reference distribution.

For a reference result `ref` and a scoring result `op`, `ref$cdf(op$distance)` gives each observation's reference percentile. A value of `0.99` means that 99 percent of the reference distances are no larger than that observation's distance. Large percentiles therefore identify unusually weak training support relative to the reference sample. A rule such as `ref$cdf(op$distance) > 0.95` flags observations in the upper tail of this distribution.

The corresponding support score, `1 - ref$cdf(op$distance)`, is the fraction of reference distances greater than the scored distance. Smaller values indicate less support. These summaries describe position within the empirical reference distribution.

Use the same model, predictors, weights, neighborhood size, and distance function for reference and new observations. For forest-based distances, also keep `max.tree` and `max.rules.tree` the same. The examples illustrate the default KNN calibration workflow.

Forest-based distances:

The other `distancef` options use neighborhoods defined by the forest. Each score summarizes standardized differences between the observation and its forest-derived neighbors:

`"prod"` Mean across neighbors of the weighted geometric mean of absolute coordinate differences, with a small positive offset.

`"euclidean"` Mean weighted Euclidean distance.

`"manhattan"` Mean weighted Manhattan distance. This uses the same pairwise metric as `"knn"`, but with forest-derived neighbors.

`"minkowski"` Mean weighted Minkowski distance of order 4.

"mahalanobis" Mean Mahalanobis-type distance based on weighted absolute coordinate differences and the covariance of the standardized training predictors.

"kernel" One minus the mean Gaussian similarity to the neighbors, based on weighted squared Euclidean distances. The bandwidth is estimated separately for each scoring call, so scores from separate calls can have different scales.

Value

outpro returns a list with the following components:

distance Numeric vector of distances, one per scored observation. Larger values indicate greater departure from the training data in the selected predictor subspace.

selected.variables
Predictor names used in scoring, after removing variables with zero or nonfinite training standard deviation.

selected.weights
Relative predictor weights after rescaling and squaring. These precede the final normalization to sum to one; distance.args$weights.used contains the weights used in the distance calculation.

distance.args Distance settings used, including distancef and weights.used. For "knn", this also contains the effective neighbor count (knn.neighbor.used), whether self matches were excluded (knn.self.excluded), and knn.chunk.size.

neighbor Requested neighborhood size after rounding and restriction to the training sample size. For KNN training scores, distance.args$knn.neighbor.used records any further reduction to exclude self matches.

cutoff Variable-priority threshold setting. It affects selection only for varpro objects with reduce = TRUE.

means, sds Training means and standard deviations for the predictors used in scoring.

dropped.zero.sd.variables
Names of predictors removed because their training standard deviations were zero or nonfinite.

distance.object
A list retaining the distance information, including standardized training and scored predictors (xorg.scale and xnew.scale), selected predictor names and relative weights (xvar.names and xvar.wt), training means and standard deviations, and neighborhood information. For forest-based distances, dist.xvar contains absolute standardized coordinate differences to the neighbors; it is NULL for "knn".

score Forest-derived neighborhood information. NULL for distancef = "knn".

oob.bits Scoring-mode indicator: 0 when newdata is omitted and 1 when it is supplied.

newdata.xscale Whether supplied data were treated as already expressed in the model's encoded predictor columns.

call The matched function call.

outpro.null returns the same components, together with:

cdf	The empirical cumulative distribution function of the reference distances. Apply this function to new distances to obtain their reference percentiles.
quantile	The empirical cumulative probability of each reference observation's distance, computed as cdf(distance).

See Also

varpro, rfsrc.

Examples

```
## Simulate a regression problem with two signal and two noise variables.
set.seed(123)
n <- 800
dta <- data.frame(x1 = rnorm(n), x2 = rnorm(n),
                  noise1 = rnorm(n), noise2 = rnorm(n))
dta$y <- 2 * dta$x1 + dta$x2^2 + rnorm(n)

## Use separate training, reference, and test samples.
vp <- varpro(y ~ ., data = dta[1:500, ])
reference.data <- dta[501:650, ]
test.data <- dta[651:800, ]

## Shift one signal variable in half of the test observations.
shifted <- seq_len(nrow(test.data)) > 75
test.data$x1[shifted] <- test.data$x1[shifted] + 6

## Score test observations using the default KNN method.
op <- outpro(vp, newdata = test.data)
print(op$selected.variables)
print(setNames(op$distance.args$weights.used, op$selected.variables))

## Compare the test distances with an in-distribution reference sample.
ref <- outpro.null(vp, nulldata = reference.data)
percentile <- ref$cdf(op$distance)
support <- 1 - percentile
flag <- percentile > 0.95

print(head(data.frame(distance = op$distance, percentile = percentile,
                      support = support, flag = flag)))
print(tapply(percentile, shifted, median))
print(table(shifted = shifted, flagged = flag))

## Alternatively, use training distances as the reference distribution.
## Omit nulldata so each training observation excludes itself in KNN scoring.
ref.train <- outpro.null(vp)
print(head(ref.train$cdf(op$distance)))

## Use all predictors with equal weights.
op.all <- outpro(vp, newdata = test.data, reduce = FALSE)
```

```
## Specify a subspace with equal weights.
op.subspace <- outpro(vp, newdata = test.data,
                      reduce = c("x1", "x2"))

## Specify relative weights. Squaring and normalization give 0.8 and 0.2.
op.weighted <- outpro(vp, newdata = test.data,
                      reduce = c(x1 = 2, x2 = 1))
print(op.weighted$distance.args$weights.used)

## Use forest-derived neighborhoods with a product distance.
op.prod <- outpro(vp, newdata = test.data, distancef = "prod")
print(head(op.prod$distance))

## Recompute the reference distribution when changing scoring settings.
ref.prod <- outpro.null(vp, nulldata = reference.data, distancef = "prod")
print(head(ref.prod$cdf(op.prod$distance)))

## Compare all five methods on the same test observations.
## AUC measures discrimination using raw distances, with larger values
## identifying shifted observations. Tied distances receive average ranks.
## At a 0.95 reference-percentile cutoff, report the false-positive rate
## (FPR) among unshifted observations and the true-positive rate (TPR)
## among shifted observations, using each reference strategy.
compare.performance <- function(op, ref, ref.train) {
  distance <- op$distance
  n1 <- sum(shifted)
  n0 <- sum(!shifted)
  auc <- (sum(rank(distance, ties.method = "average")[shifted]) -
          n1 * (n1 + 1) / 2) / (n1 * n0)

  flag.reference <- ref$cdf(distance) > 0.95
  flag.training <- ref.train$cdf(distance) > 0.95

  c(AUC = auc,
    FPR.reference = mean(flag.reference[!shifted]),
    TPR.reference = mean(flag.reference[shifted]),
    FPR.training = mean(flag.training[!shifted]),
    TPR.training = mean(flag.training[shifted]))
}

## Each reference distribution uses the same scoring settings as its
## corresponding test scores. Reuse the references already computed.
performance <- rbind(
  KNN.varPro = compare.performance(op, ref, ref.train),
  KNN.all = compare.performance(
    op.all,
    outpro.null(vp, nulldata = reference.data, reduce = FALSE),
    outpro.null(vp, reduce = FALSE)),
  KNN.signal = compare.performance(
    op.subspace,
    outpro.null(vp, nulldata = reference.data,
                reduce = c("x1", "x2")),
    outpro.null(vp, reduce = c("x1", "x2"))),
```

```
  KNN.weighted = compare.performance(
    op.weighted,
    outpro.null(vp, nulldata = reference.data,
                reduce = c(x1 = 2, x2 = 1)),
    outpro.null(vp, reduce = c(x1 = 2, x2 = 1))),
  Forest.product = compare.performance(
    op.prod, ref.prod, outpro.null(vp, distancef = "prod"))
)
print(round(performance, 3))
```

---

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 dropped with a warning listing the unavailable names. This check is applied after the nvar limit.
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. Unnamed options, unrecognized names, and duplicate names cause an error.

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` for prediction; it applies only to regression/survival-style VarPro objects.

All three learner helpers require uniquely named options and reject unrecognized names before fitting. For `rf.learner`, `formula`, `data`, `xvar.wt`, and `perf.type` are set internally and cannot be supplied through `...`.

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 Individual Variable Priority*

Description

Display case-specific iVarPro gradients for one predictor, or compare their distributions across predictors in a beeswarm-style summary.

Usage

```
## S3 method for class 'ivarpro'
plot(x, var, col.var = NULL, size.var = NULL,
  data = NULL, target = NULL, pch = 16, cex = 0.8,
  cex.range = c(0.5, 2), main = NULL, xlab = NULL,
  ylab = "iVarPro gradient", legend = TRUE, ...)

shap.ivarpro(ivar, dat = NULL, feature_names = NULL,
  max.points = 5000, max.points.per.feature = NULL,
  point.alpha = 1.0, point.size = 0.35, point.pch = 16,
  scale.value = TRUE, style = c("blobby", "jitter"),
  blobby.separation = 3, target = NULL)
```

Arguments

`x`, `ivar`	An iVarPro object, or a numeric data frame or matrix of case-specific gradients.
`var`	Predictor to display, given by its column name or index in the gradient table. Use the processed predictor name for a hot-encoded factor.
`col.var`	Optional column of the plotting data used to color points. A factor gives group colors; a continuous variable gives a color gradient.
`size.var`	Optional column of the plotting data used to scale point sizes.

data, dat	Plotting data with rows in the same order as the gradient table. By default, use the data stored by ivarpro or predict.ivarpro. The plotted predictor columns must be numeric; the stored data provide the appropriate hot-encoding.
target	For a multi-target object, the response or class name, or its integer position. If omitted, use the first target with a warning.
pch, cex	Point symbol and size for the single-predictor plot. Group-specific point styles can override pch.
cex.range	Range of point sizes when size.var is supplied.
main, xlab, ylab	Plot title and axis labels.
legend	Display the color legend in the single-predictor plot?
...	Graphical arguments and optional display controls for plot; see Details.
feature_names	Names for an unnamed gradient matrix in the summary plot. Existing column names take precedence.
max.points	Maximum total number of points in the summary plot. Larger collections are subsampled.
max.points.per.feature	Optional additional limit on the number of displayed points per predictor.
point.alpha, point.size, point.pch	Point opacity, size and symbol for the summary plot.
scale.value	Rescale each predictor's displayed values to the range zero to one for coloring? Default is TRUE. This changes the colors, not the gradient scores.
style	Summary-point arrangement: "blobby" separates overlapping points into a beeswarm-like display; "jitter" uses vertical jitter.
blobby.separation	Spacing between points for style = "blobby". Larger values increase separation.

Details

Single-predictor plot: plot(x, var = "x1") places predictor values on the horizontal axis and their local gradients on the vertical axis. A zero reference line helps distinguish positive and negative relationships. Coloring by another predictor can reveal how the relationship varies across cases.

Smooth curves are drawn by default, with separate curves for color groups. Useful controls passed through ... include smooth = FALSE to suppress curves, smooth.span to adjust smoothing, and jitter = FALSE to suppress horizontal jitter. jitter.seed makes jitter reproducible.

Use x.dist = "rug", "hist", or "density" to show the predictor distribution along the horizontal axis. Combinations such as x.dist = c("hist", "rug") are allowed. zero.line = FALSE suppresses the reference line. col.style selects "auto", "solid", "outline", or "binary" point styling.

Summary plot: shap.ivarpro(x) displays the iVarPro gradients for all predictors. Horizontal position shows a gradient's sign and magnitude; color shows the predictor value. Predictors are ordered by mean absolute gradient. Columns containing only zero or nonfinite scores are omitted.

The gradient scale is set by the original ivarpro call. In particular, scale = "global" expresses changes on a common training-standard-deviation scale within each predictor, while scale = "none" shows slopes in original predictor units. See ivarpro for the three scaling choices.

Value

Called for its plotting side effect. `plot.ivarpro` returns `TRUE` invisibly; `shap.ivarpro` returns `NULL` invisibly.

See Also

`ivarpro`, `predict.ivarpro`

Examples

```
## Compare local mortality gradients and inspect peak oxygen uptake.
library(survival)
data(peakVO2, package = "randomForestSRC")
peak <- na.omit(peakVO2)
set.seed(137)
vp <- varpro(Surv(ttodead, died) ~ ., peak,
             split.weight = FALSE, ntree = 100,
             parallel = FALSE)
ivp <- ivarpro(vp)
print(head(ivp))
shap.ivarpro(ivp)
plot(ivp, var = "peak.vo2", col.var = "interval")
## scale points to "y" (here equal to mortality)
plot(ivp, var = "peak.vo2", col.var = "interval", size.var = "y",
     x.dist = c("hist", "rug"), jitter = FALSE)
```

---

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 Individual Variable Priority for New Cases*

Description

Apply an iVarPro analysis to new observations, or recover its out-of-bag training scores. The result gives a local gradient for each case and processed predictor.

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`.
`newdata`	A data frame containing the predictors for new cases. Outcome columns are optional. For a VarPro model, supply the original predictor columns; the training hot-encoding is applied automatically. If `NULL`, recover the original training scores using out-of-bag rule membership.

model	The VarPro object or random forest used to obtain object. By default, use the model stored by ivarpro(save.model = TRUE). Supply it here if it was not stored.
noise.na	How to average unavailable rule scores: TRUE omits them and returns NA where no usable score is available; FALSE includes them as zeros. By default, inherit the setting from object.
path.store.membership	Retain the case-to-rule membership indices with the result? These are optional diagnostic information.
save.data	Store the processed predictor data for subsequent plots? Default is TRUE.
...	Additional arguments passed to the forest prediction method when memberships are obtained.

Details

New cases are routed through the original forest. For each predictor, their scores average the stored gradients of applicable rules that release that predictor. The local slopes and neighborhood choices remain those learned in the original iVarPro analysis.

Scores retain the target, scaling and signed or absolute convention chosen in ivarpro. Multiple targets produce a named list of score tables. Use plot or shap.ivarpro to inspect the result, with target selecting a response or class when needed.

Omitting newdata recovers training scores using out-of-bag membership. Passing the training predictors explicitly treats those observations as new cases and uses all applicable forest rules, so the two calls can give different scores.

Value

An object of class ivarpro, with one row per prediction case and one column per processed predictor. A single target produces a data frame; multiple targets produce a named list of data frames. Row names identify the prediction cases.

The result retains the target labels, original model and rule information needed for further prediction. With save.data = TRUE, its data attribute contains aligned plotting data. For new cases this contains processed predictors; the original explanation targets are not recalculated. See ivarpro for score interpretation.

See Also

ivarpro, plot.ivarpro, shap.ivarpro

Examples

```
## Train on one sample and explain new observations.
set.seed(137)
n <- 400
sim <- data.frame(x1 = runif(n, -1, 1), x2 = runif(n, -1, 1))
sim$y <- 6 * sim$x1 * sim$x2 + rnorm(n, sd = 0.3)
trn <- sample.int(n, 300)
vp <- varpro(y ~ ., sim[trn, , drop = FALSE],
```

```
             split.weight = FALSE, ntree = 100,
             parallel = FALSE)
iv <- ivarpro(vp, scale = "none", nmin = 10)

## Outcome values are not needed for new-case gradients.
test <- sim[-trn, c("x1", "x2"), drop = FALSE]
iv.test <- predict(iv, newdata = test)
print(head(iv.test))
plot(iv.test, var = "x1", col.var = "x2")

## Recover the original out-of-bag training scores.
iv.oob <- predict(iv)
print(all.equal(as.matrix(iv.oob), as.matrix(iv),
                check.attributes = FALSE))
```

`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 and Dependence Graphs using UVarPro*

Description

Performs unsupervised variable selection using the Variable Priority framework. uvarpro constructs forest rules and region-release contrasts from unlabeled data. get.beta.entropy uses local lasso regressions to summarize which variables distinguish these contrasts. sdependent displays the resulting importance matrix as a variable-dependence graph and identifies candidate signal variables.

Usage

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

get.beta.entropy(o,
                 second.stage = FALSE,
                 pre.filter = TRUE,
                 papply = mclapply,
                 vimp.min = 0,
                 nfolds = 10,
                 maxit = 2500,
                 thresh = 1e-3,
                 parallel = FALSE,
                 use.cv = TRUE,
                 lambda.sel = c("lambda.1se", "lambda.min"),
                 nlambda = NULL,
                 lambda.min.ratio = NULL,
                 nlambda.lasso = nlambda,
                 lambda.min.ratio.lasso = lambda.min.ratio)
```

```
sdependent(I,
           threshold = .25,
           layout = "grid",
           q.signal = .75,
           directed = TRUE,
           min.degree = NULL,
           title = "s-Dependent Variable Detection",
           plot = TRUE)
```

Arguments

data	Data frame containing the features to be analyzed, without an outcome or class-label column. Rows with missing values are omitted, unused factor levels are dropped, and categorical features are hot-encoded. See Details.
method	Forest construction method. "auto" uses a multivariate forest autoencoder, "unsupv" uses unsupervised splitting, and "rnd" uses random splitting.
ntree	Number of trees to grow.
nodesize	Minimum terminal node size. If NULL, the requested size is n / 10 for n < 100, and max(n / 10, 20) otherwise, where n is the number of complete observations.
max.rules.tree	Maximum number of rules sampled per tree for the region-release analysis.
max.tree	Maximum number of trees used to extract rules.
verbose	Currently unused.
seed	Seed argument for uvarpro. Seed the lasso cross-validation separately; see Note.
...	Additional named forest arguments recognized by rfsrc, such as mtry or ytry, and a custom scoring function supplied as entropy; see Details. Supply lasso controls to get.beta.entropy.
o	An object returned by uvarpro, containing the processed feature data in x and the region membership lists in entropy.
second.stage	If FALSE, return the mean absolute logistic-lasso coefficient matrix. If TRUE, perform an additional local linear lasso using the predictors selected by the logistic stage, and return its mean absolute coefficients divided by the standard deviation of the released variable. See Details.
pre.filter	Should preliminary importance from uvarpro restrict the lasso analysis to variables with get.vimp(o, pretty = FALSE) > vimp.min? The filter applies to both predictors and release variables. It is skipped when preliminary importance is unavailable.
papply	An lapply-like function for processing released variables. Use lapply for serial execution. The default mclapply is resolved through the package parallel backend using getOption("mc.cores", 1L). The parallel argument controls parallelism across cross-validation folds.
vimp.min	Strict lower cutoff for preliminary importance when pre.filter = TRUE. The default retains values greater than zero. Ignored when pre.filter = FALSE.

`nfolds` Number of cross-validation folds for each local logistic-lasso analysis. At least three folds are required. Ignored when `use.cv = FALSE`.

`maxit` Maximum number of iterations passed to `glmnet` in both lasso stages.

`thresh` Convergence tolerance passed to `glmnet`.

`parallel` Should `cv.glmnet` process its folds in parallel? A suitable `foreach` backend must be registered. Ignored when `use.cv = FALSE`. Account for available resources when enabling both levels of parallelism.

`use.cv` Should the logistic-lasso penalty be chosen by cross-validation? If `FALSE`, coefficients are taken at the last, smallest penalty on the computed `glmnet` path.

`lambda.sel` Cross-validation penalty choice for the logistic stage. `"lambda.1se"` selects the largest penalty whose cross-validation error is within one standard error of the minimum; `"lambda.min"` selects the penalty minimizing that error. Ignored when `use.cv = FALSE`.

`nlambda` Number of penalty values requested for the logistic stage. `NULL` leaves the `glmnet` default unchanged. The computed path can end before all requested values are reached.

`lambda.min.ratio`
Smallest requested logistic-stage penalty relative to the largest penalty on the path. `NULL` uses the `glmnet` default.

`nlambda.lasso`, `lambda.min.ratio.lasso`
Corresponding path controls for the optional second-stage linear lasso. They default to the logistic-stage settings and are used only when `second.stage = TRUE`. The second stage uses the last computed penalty.

`I` A finite, nonnegative numeric importance matrix, usually obtained from `get.beta.entropy`, with released variables as rows and predictors as columns. Supply unique, nonempty row and column names. Row names must belong to the column names and may occur in any order. Rows are aligned by name; missing release rows are filled with zeros.

`threshold` Positive edge cutoff on the scale of `I`. An off-diagonal entry greater than or equal to this value creates an edge. Increasing the cutoff removes edges; decreasing it can recover weaker connections.

`layout` Graph layout. Supported names are `"fr"`, `"dh"`, `"gem"`, `"kk"`, `"lgl"`, `"mds"`, `"sugiyama"`, `"graphopt"`, `"nicely"`, `"random"`, `"sphere"`, `"grid"`, `"circle"`, `"star"`, `"tree"`, and `"bipartite"`; unambiguous abbreviations are accepted. Layout-specific requirements of **igraph** still apply. Alternatively, supply a numeric coordinate matrix with one row per nonisolated vertex, in graph vertex order.

`q.signal` Quantile of the global importance scores among nonisolated vertices used for signal designation. Values range from zero to one; the default is the upper quartile. A signal must meet both this cutoff and the degree condition.

`directed` Should the graph be directed? In the directed graph, an edge goes from the released variable to the predictor used to distinguish its region-release contrasts. For an undirected display, supply a symmetric importance matrix; see Note.

`min.degree` Minimum degree for candidate signal designation. With directed graphs this is the out-degree. The default is 1 for directed graphs and 2 for undirected graphs.

title	Title for the graph.
plot	Should the graph be plotted? If FALSE, graph construction and signal selection are still performed. The **igraph** package is required in either case.

Details

Overview and forest construction: UVarPro studies dependence among features in unlabeled data. The population framework of Zhou et al. (2026) seeks a signal set that accounts for dependence among the remaining variables. Redundant or interchangeable variables can give rise to multiple signal sets.

The analysis has three steps: uvarpro constructs local region-release comparisons, get.beta.entropy estimates a lasso coefficient importance matrix, and sdependent displays its dependence graph.

For method = "auto", copies of the processed features serve as the multivariate response and the original features serve as predictors in rfsrc. Regressing the feature vector on itself produces a forest autoencoder that partitions feature space.

For method = "unsupv", the forest uses unsupervised splitting with internally selected pseudo-responses. The default ytry is min(ceiling(sqrt(p)), p - 1), where p is the number of processed features. For method = "rnd", an independent Gaussian response is generated and splitrule = "random" is used. All three methods use the same subsequent region-release and lasso calculations.

Complete observations are hot-encoded and saved as o$x. Membership indices refer to these observations; lasso rows and columns and graph vertices refer to the processed features. Each encoded column is analyzed separately.

Rules, release regions, and local classification: A tree rule defines a region R through restrictions on feature values. Releasing variable s removes its restrictions while retaining the others. Write the enlarged region as $R^{(s)}$ and its complementary, or near-miss, region as

$$C_s = R^{(s)} \setminus R.$$

The complementary region contains observations admitted by releasing s that still satisfy the other rule conditions.

uvarpro extracts original-region out-of-bag membership and complementary-region membership. Comparisons are retained when both sets are nonempty. The default o$entropy stores these index pairs by released variable, with comp first and oob second.

For each comparison, get.beta.entropy combines the complementary observations (class 0) and original observations (class 1). It predicts this membership label using the eligible features other than the released variable. Excluding the released variable prevents the classifier from simply recovering the rule restriction that separates the groups. The remaining predictors identify features that distinguish the local contrast. Repeating this analysis across rules allows different predictors to contribute in different parts of feature space.

Preliminary dispersion importance: The default entropy function computes a dispersion ratio. Let $d(M)$ be the average column standard deviation of a feature matrix M. For the original and complementary feature matrices X_O and X_C, the stored rule score is

$$H = \frac{d(X_C) + d(X_O)}{2d(X_C \cup X_O)}.$$

Here the union denotes row concatenation. The ratio compares average within-group dispersion with pooled dispersion. Before scoring, features are divided by their full-data standard deviations without centering; near-zero standard deviations are replaced by one. All features, including the released variable, enter this calculation.

`importance(o)` summarizes the rule scores. By default, `get.beta.entropy` uses preliminary importance to screen predictors and release variables before the local regressions. Set `pre.filter = FALSE` to include all processed features.

Logistic lasso and the importance matrix: Each local classification problem uses binomial lasso logistic regression from **glmnet**, with penalty parameter $\alpha = 1$. With `use.cv = TRUE`, `cv.glmnet` selects a penalty separately for each comparison using binomial deviance; the default choice is `lambda.1se`. With `use.cv = FALSE`, the last computed penalty on the regularization path is used.

For released variable s and retained comparison r, let $\widehat{\beta}_{srj}$ denote the coefficient of predictor j. The entries of the importance matrix are mean absolute coefficients across retained comparisons. The intercept is discarded; unselected predictors and the released coordinate receive zero. Writing $\mathcal{K}_s$ for the retained comparisons, the default matrix has entries

$$I_{sj} = \frac{1}{|\mathcal{K}_s|} \sum_{r \in \mathcal{K}_s} |\widehat{\beta}_{srj}|, \qquad j \neq s, \qquad I_{ss} = 0.$$

Rows identify released variables and columns identify predictors. Thus `I[s, j]` measures predictor `j`'s contribution when variable `s` is released. The matrix is generally asymmetric. Column sums aggregate each predictor's contributions across release tasks; column means give the same ranking.

A comparison is retained when fitting and coefficient extraction succeed and at least one finite, nonzero predictor coefficient remains. Failed fits and intercept-only solutions are excluded. Retained comparisons receive equal weight, including zeros for unselected predictors. A release row is omitted when it has no retained comparisons, so predictor columns can outnumber rows. The result is `NULL` when no rows remain.

Coefficient scaling and the optional second stage: Before each logistic fit, `scale(X, center = FALSE)` divides each predictor column by $\sqrt{\sum_i X_{ij}^2/(m-1)}$, where m is the local sample size. `glmnet` also applies its default internal standardization and returns coefficients on the scale of the matrix supplied to it.

With `second.stage = TRUE`, predictors selected by the logistic lasso enter a local Gaussian lasso with the released variable as its response. This regression uses the combined complementary and original observations, the same predictor scaling, and the response from `o$x`. Coefficients are taken at the last computed penalty on the linear-lasso path.

Absolute second-stage coefficients are averaged over the comparisons retained by the logistic stage, with zeros for unselected predictors. A second-stage fit that fails or selects no predictors contributes zeros to this average. Row s is then divided by the full-data standard deviation of the released variable:

$$I^{(2)}_{sj} = \frac{1}{\widehat{\sigma}_s |\mathcal{K}_s|} \sum_{r \in \mathcal{K}_s} |\widehat{\gamma}_{srj}|.$$

Here $\widehat{\gamma}_{srj}$ is a second-stage coefficient. A zero or nonfinite response standard deviation produces `NA` in that row.

`get.beta.entropy` returns this matrix when `second.stage = TRUE`. Choose the graph cutoff on the coefficient scale of the selected stage.

Adjacency matrix and interpretation of the graph: In UVarPro, s-dependence concerns which variables remain informative about the local contrast induced by releasing s. The graph summarizes their estimated contributions.

The theoretical motivation distinguishes releasing a noise variable, whose contrast is explained by associated signal variables, from releasing a signal variable, whose contrast can also involve associated noisy variables. This asymmetry motivates aggregating contributions across release tasks.

`sdependent` aligns rows to column names, adds zero rows for missing release variables, and then clears the diagonal. Column order determines vertex order before isolated vertices are removed. It calculates the column scores $G_j = \sum_s I_{sj}$ and constructs the binary adjacency matrix

$$A_{sj} = \mathbf{1}\{I_{sj} \geq t\}, \quad s \neq j, \qquad A_{ss} = 0,$$

where t is `threshold`. Each retained connection is one edge, with its plotted width determined by the corresponding importance entry.

With `directed = TRUE`, $A_{sj} = 1$ creates the arrow $s \longrightarrow j$: predictor j contributes to distinguishing the comparisons formed by releasing s. Arrows describe predictive contributions. Reciprocal arrows and cycles are allowed.

The out-degree of s counts thresholded entries in its row, giving the number of predictors connected to its release tasks. The in-degree of j counts thresholded entries in its column, giving the number of release variables connected to that predictor. Global importance G_j sums the unthresholded magnitudes.

Isolated vertices are removed. Global scores are calculated before thresholding and removal, so they include subthreshold contributions and contributions from rows whose vertices are later removed. The signal-designation quantile uses the remaining vertices.

Signal designation and graphical appearance: A retained vertex enters `signal.vars` when its degree is at least `min.degree` and its global importance is at least the `q.signal` quantile of retained scores. For directed graphs, these conditions use out-degree and column-sum importance. The defaults require at least one outgoing edge and importance at or above the upper quartile of nonisolated vertices, including ties.

Signal vertices are blue; other retained vertices are gray. A directed edge is blue when its source is designated as signal and gray otherwise. Vertex size is `6 + 2 * log1p(imp.score)`. An edge with importance w has width $1 + 3w/w_{\max}$, where $w_{\max}$ is the largest plotted edge value. Widths are rescaled for each plot.

`threshold` controls the displayed connections; `q.signal` and `min.degree` control vertex highlighting. Changing the edge cutoff can also change the quantile reference set through vertex removal. Signal designations are exploratory.

Latent-variable example: The simulation in Examples contains four independent standard normal variables `w`, `x`, `y`, and `z`, each accompanied by five noisy copies with independent Gaussian errors of variance 0.1. Two further features are $h_1 = w + x + \epsilon_{21}$ and $h_2 = y + z + \epsilon_{22}$, with independent Gaussian errors of variance 0.4. All errors are independent of the four source variables. All 26 features, including the sources, are supplied to `uvarpro`.

The signal set is $\{w, x, y, z\}$: conditioning on these four variables makes the other 22 features mutually independent. Releasing any of `wi.1`, ..., `wi.5` has `w` as its s-dependent signal; the `xi`,

yi, and zi blocks similarly point to x, y, and z. Releasing h1 has s-dependent signals w and x, while releasing h2 has y and z.

When a source such as w is released, its noisy copies and h1 can help predict local membership; x can also contribute by accounting for its component of h1. The graph can therefore contain connections in both directions. Repeated contributions across the noisy-feature releases motivate the column-sum ranking of the four signals.

The example uses pre.filter = FALSE to keep all 26 features eligible as predictors and release variables. It compares the estimated importance entries and graph connections with this known dependence structure.

Custom scoring functions: A custom entropy function accepts xC, x0, and .... The first two arguments contain the scaled complementary and original features, with the released feature first. Additional information includes compMembership, oobMembership, the processed data, and xvar.names.

Return a numeric rule score, or a list containing the score first and auxiliary information second. For subsequent use of get.beta.entropy, supply list(comp = compMembership, oob = oobMembership) as the second element, preserving that order. See Examples.

Value

uvarpro returns an object of class "uvarpro". Its principal components are:

rf The forest used to generate rules.

results Rule-level information with columns tree, branch, variable, n.oob, and imp. The variable index refers to xvar.names. Unevaluated comparisons have NA importance.

x The complete-case, hot-encoded feature data, before the scaling used in scoring or local regressions.

xvar.names, xvar.org.names Processed and original feature names, respectively.

entropy With the default scoring function, membership pairs grouped by released variable. Each pair contains comp and oob observation indices into x. May be NULL when no usable memberships are available. Custom callbacks can change the auxiliary information stored here.

max.rules.tree, max.tree Rule extraction limits.

family The value "unsupv", irrespective of the forest construction method.

get.beta.entropy returns a named numeric matrix, with released variables as rows and eligible predictors as columns. Entries are mean absolute logistic-lasso coefficients when second.stage = FALSE, and response-scaled mean absolute linear-lasso coefficients when second.stage = TRUE. Rows with no retained comparisons are omitted. Returns NULL if no usable release rows remain.

sdependent invisibly returns a list with components:

signal.vars Names of vertices meeting both the degree and importance-quantile conditions. Can be empty even when the graph is nonempty.

imp.score Named global column-sum scores for nonisolated vertices, sorted in increasing order.

degree Named degrees for nonisolated vertices, in graph vertex order. These are out-degrees when directed = TRUE.

If no edges remain, sdependent returns a character diagnostic. See Note.

Note

For reproducible serial lasso calculations, call `set.seed` immediately before `get.beta.entropy(o, papply = lapply)` to control its randomized cross-validation folds. Seed forest construction separately. Small local classes, degenerate predictors, and excessive pre-filtering can leave no usable comparisons. Individual lasso errors are caught and warnings suppressed.

For an undirected display, supply a symmetric importance matrix. For square input with matching row and column order, `pmax(I, t(I))` retains either direction. Symmetrization can change global scores and signal designations. Undirected edge colors use the first endpoint in the internal edge list.

To admit weaker positive edges in an empty graph, decrease `threshold`. Keep the cutoff positive, since `>=` admits zero-valued entries at a zero cutoff. Check for `NULL` lasso output before graph construction.

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. doi:10.1016/j.patcog.2025.112727.

See Also

varpro, importance, cv.glmnet, glmnet, scale, graph_from_adjacency_matrix

Examples

```
## Small forest for a quick check
set.seed(1)
o <- uvarpro(mtcars, ntree = 1)


## ------------------------------------------------------------
## Latent-variable model: known signals and s-dependence
## ------------------------------------------------------------
set.seed(21)
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)
signal <- c("w", "x", "y", "z")

## All 26 columns are observed features. The remaining 22 are
## mutually independent conditional on the four signal variables.
o <- uvarpro(dta)
print(importance(o))

## Retain every feature for the local lasso analysis, including
## the noisy-copy and mixed-variable releases that reveal signals.
set.seed(22)
beta <- get.beta.entropy(o, pre.filter = FALSE,
                         papply = lapply, nfolds = 5)

if (!is.null(beta)) {
  ## Compare the column-sum ranking with the known signal set.
  score <- sort(colSums(beta), decreasing = TRUE)
  print(data.frame(variable = names(score),
                   importance = unname(score),
                   signal = names(score) %in% signal))

  ## Inspect one copy from each block and the two mixed variables.
  ## Population targets: each copy points to its source; h1 points
  ## to w and x, and h2 to y and z. See Details for source releases.
  release <- c("wi.1", "xi.1", "yi.1", "zi.1", "h1", "h2")
  release <- intersect(release, rownames(beta))
  print(round(beta[release, signal, drop = FALSE], 2))

  ## Inspect the corresponding adjacency block.
  ## A[s, j] = 1 gives the arrow s -> j.
  threshold <- 0.25
  A <- 1L * (beta[release, signal, drop = FALSE] >= threshold)
  print(A)

  if (all(is.finite(beta)) &&
      requireNamespace("igraph", quietly = TRUE)) {
    set.seed(23)
    ## Pass the lasso matrix directly; row alignment is automatic.
    graph.info <- sdependent(beta, threshold = threshold, layout = "fr",
                             title = "Latent model: s-dependence")
    if (is.list(graph.info)) {
      print(graph.info$signal.vars)
      print(graph.info$degree)
    } else {
      print(graph.info)
    }

    ## Return graph summaries without drawing the figure:
```

```
    ## graph.info <- sdependent(beta, threshold = threshold, plot = FALSE)
  }
}

## Optional second-stage lasso, keeping the same feature set.
## set.seed(22)
## beta.second <- get.beta.entropy(o, pre.filter = FALSE,
##                                 second.stage = TRUE,
##                                 papply = lapply, nfolds = 5)

## ------------------------------------------------------------
## Boston housing: alternative forest and categorical features
## ------------------------------------------------------------
if (requireNamespace("mlbench", quietly = TRUE)) {
  data(BostonHousing, package = "mlbench")

  ## Every column is treated as a feature; no response is specified.
  o.unsupv <- uvarpro(BostonHousing, method = "unsupv")
  print(importance(o.unsupv))

  ## Random splitting uses the same subsequent analysis.
  ## o.random <- uvarpro(BostonHousing, method = "rnd")

  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))

  o.factor <- uvarpro(Boston)
  print(importance(o.factor))
  print(get.orgvimp(o.factor))
  print(get.orgvimp(o.factor, pretty = FALSE))
}

## ------------------------------------------------------------
## Iowa housing: larger lasso analysis
## ------------------------------------------------------------
data(housing, package = "randomForestSRC")
iowa <- roughfix(housing)
## Numeric coding below is only a speed-oriented illustration;
## retain factors to use the usual hot-encoding instead.
iowa <- data.frame(data.matrix(iowa))
o.iowa <- uvarpro(iowa, ntree = 50, max.tree = 50,
                  max.rules.tree = 5)
set.seed(22)
beta.iowa <- get.beta.entropy(o.iowa, papply = lapply, nfolds = 5)
if (!is.null(beta.iowa)) {
  print(sort(colMeans(beta.iowa), decreasing = TRUE))
  ## sdependent(beta.iowa)
}
```

```
## ------------------------------------------------------------
## Custom scoring callback: reproduces the default dispersion ratio
## ------------------------------------------------------------
my.entropy <- function(xC, xO, ...) {
  mean.sd <- function(x) mean(apply(x, 2, sd, na.rm = TRUE))
  imp <- 0.5 * (mean.sd(xC) + mean.sd(xO)) /
    mean.sd(rbind(xC, xO))
  dots <- list(...)
  list(imp = imp,
       membership = list(comp = dots$compMembership,
                         oob = dots$oobMembership))
}

if (requireNamespace("mlbench", quietly = TRUE)) {
  data(BostonHousing, package = "mlbench")
  o.custom <- uvarpro(BostonHousing, entropy = my.entropy,
                      ntree = 50, max.tree = 50, max.rules.tree = 5)
  print(importance(o.custom))
  ## you can still run the lasso if you want
  ## beta.custom <- get.beta.entropy(o.custom, papply = lapply)
}
```

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

Description

Identifies predictors associated with a response by comparing forest rule regions with their near-miss sets. Supports regression, multivariate regression, classification, and right-censored survival. The returned object contains the rule-generating forest and rule-level scores; `importance` summarizes these scores and `cv.varpro` selects an importance cutoff using predictive performance.

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 response and predictors, such as `y ~ .`, `cbind(y1, y2) ~ .`, or `Surv(time, status) ~ .`. Use a factor response for classification and numeric responses for regression. Survival status is coded 0 for censoring and 1 for an event.

`data` Data frame containing the response and predictors. Incomplete observations are removed, unused factor levels are dropped, and categorical predictors are hot-encoded. If observations are omitted, a warning reports the input, omitted, and retained counts. See Details.

`nvar` Maximum number of processed predictor columns retained by split-weight screening before rule generation. A categorical predictor can contribute several columns. Applies when computed or custom split-weights are used; `split.weight = FALSE` without custom weights uses all processed predictors.

`ntree` Number of trees in the rule-generating forest.

`split.weight` Should preliminary split-weights guide rule generation? Positive-weight predictors are screened to at most `nvar` columns, and candidate split variables are sampled according to their retained weights.

`split.weight.method`
Character string or vector selecting preliminary weighting methods: `"lasso"`, `"tree"`, and `"vimp"`. Methods can be combined, for example `c("lasso", "tree")`. The default `NULL` selects a combination automatically. See Details.

`sparse` Should preliminary weighting concentrate more strongly on promising predictors? Useful when relatively few predictors are expected to carry signal. See Details.

`nodesize` Minimum terminal node size for the rule-generating forest. Larger values produce broader regions with more observations for the local comparisons. The default is chosen automatically from the data dimensions.

`max.rules.tree` Maximum number of tree branches sampled per selected tree. A branch can yield several comparisons, one for each variable released from its rule.

`max.tree` Maximum number of trees sampled for rule extraction. Increasing this value provides more trees for importance aggregation, up to the number grown.

`parallel` Should the preliminary lasso cross-validation folds run in parallel?

`cores` Number of cores for lasso-fold parallelism. The default respects `options(mc.cores = ...)` or `MC_CORES`. Forest parallelism is controlled separately by **randomForestSRC**.

`verbose` Should progress messages and preliminary split-weights be printed?

`seed` Seed supplied to the main forest calculations. Use `set.seed` before the call to also control R-level sampling, lasso folds, and auxiliary calculations. See Note.

`...` Additional named controls for split-weight construction, rule generation, and external survival estimation. The supported controls used most often are described in Details. Unnamed controls, unrecognized names, and duplicate names cause an error.

Details

What VarPro measures: VarPro studies a predictor's contribution by comparing responses in a rule region and its near-miss set. In regression, the target is the conditional mean of the response; in classification, it is the vector of conditional class probabilities. Survival analysis uses a forest estimate of mortality or restricted mean survival time as its response summary.

The workflow has three steps. `varpro` prepares the data, generates rules, and stores their release comparisons. `importance` combines the comparisons into variable-level scores. `cv.varpro` uses the scores to form candidate variable sets and chooses among them using forest prediction error.

Rules and near-miss comparisons: A root-to-terminal-node path defines a region R by combining its split conditions. Releasing predictor s removes every condition involving that predictor while retaining the conditions on the other predictors. This produces the enlarged release region $R^{(s)}$. The code compares responses in R with those in the near-miss set

$$C_s = R^{(s)} \setminus R = R^c \cap R^{(s)}.$$

For example, a rule $x_1 \leq 0.5,\ x_2 > 0.3$ becomes $x_2 > 0.3$ when x_1 is released. Its near-miss set contains observations with $x_1 > 0.5,\ x_2 > 0.3$. Comparing responses in the original region and the near-miss set measures the local contribution of x_1 while preserving the restriction on x_2.

Rules are extracted from the forest. At most `max.tree` trees and `max.rules.tree` branches per selected tree contribute to the analysis. Original-region observations are out-of-bag for the corresponding tree. Repeated comparisons allow a predictor to contribute differently in different regions of the feature space.

Guided rule generation and the lasso: Preliminary split-weights help the forest form informative rules. Three sources of information are available:

`"lasso"` Absolute coefficients from a cross-validated penalized regression on standardized predictors. This favors predictors with strong linear effects or categorical contrasts.

`"tree"` Split frequencies from a shallow forest, which can identify nonlinear effects and interactions.

`"vimp"` Positive permutation importance from a preliminary forest, measuring a predictor's contribution to prediction accuracy.

Combining methods uses complementary evidence when forming the weights. With `split.weight.method = NULL`, the combination is chosen automatically. Set an explicit method to control which sources are used.

The lasso penalty is selected by cross-validation. The weights use the more regularized solution within one standard error of the minimum error. Coefficient magnitudes are combined across responses or classes when needed. For survival, the lasso uses the response summary described below.

With `sparse = TRUE`, stronger preliminary weights receive greater emphasis. Use `FALSE` to spread the emphasis more broadly. At most `nvar` positive-weight processed columns are retained for rule generation. Increase `nvar` to explore more predictors, or use `split.weight = FALSE` to generate rules using all processed predictors without preliminary weighting.

The retained weights guide candidate-variable sampling in the forest. Final VarPro importance is calculated from the resulting release comparisons.

Importance scores and selection: Call `importance(o)` to summarize a `varpro` object. Its default `local.std = TRUE` recomputes locally standardized comparisons from the region memberships. For regression, the score comparing R and its near-miss set C_s is the absolute Welch statistic

$$T = \frac{|\bar{y}_R - \bar{y}_{C_s}|}{\sqrt{s_R^2/n_R + s_{C_s}^2/n_{C_s}}}.$$

The same calculation is applied separately to each numeric response, including external survival summaries. For classification, the overall comparison is the square root of a Pearson chi-squared statistic divided by its degrees of freedom, with continuity correction for two-class tables. Class-specific comparisons use absolute standardized differences in proportions.

Within each tree, comparisons for a predictor are averaged with weights equal to their original-region sample sizes. Class-specific summaries use the corresponding class counts. The tree-level scores are then averaged after winsorization at the lower and upper 10 percent quantiles by default. With local standardization, this mean is the reported `z` score. With `local.std = FALSE`, `importance` summarizes the stored rule scores and divides their winsorized mean by their winsorized standard deviation across trees.

Larger scores indicate stronger response changes under release. The default `importance` cutoff of 0.79 is used to highlight variables in its plots. To choose a cutoff from predictive performance, use `cv.varpro`. Its optional `cv.folds` argument assesses the complete selection procedure on held-out folds and appends stability summaries to the returned importance tables. See `importance.varpro` for the summary and plotting controls.

Categorical predictors and outcome families: Numeric predictors retain their values. Two-level categorical predictors become binary columns; predictors with more levels become indicator columns. Screening, forest rules, and `importance(o)` operate on these processed columns. `o$x` stores the complete processed predictor data, while `o$xvar.names` identifies the columns retained for rule generation.

`get.orgvimp(o)` summarizes scores at the original-variable level using the largest score among a predictor's encoded columns. For multivariate regression it also takes the maximum across outcomes. `cv.varpro` uses this original-variable summary for its candidate sets. `get.vimp(o)` returns named scores for the represented processed columns. With `pretty = FALSE`, `get.vimp` and `get.orgvimp` return all processed or original predictors, respectively, filling absent scores with zero.

In binary classification, the working response labels are 0 for the majority class and 1 for the minority class. For imbalanced classes, the default handling uses AUC splitting and RFQ with geometric-mean performance for the preliminary permutation-importance calculation. The original response labels remain in `y.org` and the rule-generating forest. The controls `use.rfq` and `iratio.threshold` modify this behavior.

Survival targets: A preliminary survival forest estimates a numeric response summary for each training observation. By default, this is forest mortality. Supplying `rmst = tau` requests restricted mean survival time, obtained by integrating the estimated survival curve up to `tau`. A vector of horizons supplies one numeric response per horizon.

These full-ensemble training summaries are used for preliminary weighting and the default locally standardized importance. With mortality or one RMST horizon, rules are generated using the original survival outcome. With multiple RMST horizons, rule generation uses multivariate regression on the RMST responses. Thus the horizons determine the survival features being prioritized. The controls `ntree.external`, `nodesize.external`, and `ntime.external` govern the preliminary survival forest.

Additional controls and computation: Useful arguments supplied through `...` include `nfolds` for lasso cross-validation, `rmst` for the survival target, and `ntree.external` for the preliminary survival forest.

Custom split-weights can be supplied as a named numeric vector through `split.weight.custom`. Names must refer to processed predictor columns; omitted columns receive zero weight. Use

get.splitweight.custom(formula, data) to obtain a correctly named starting vector, then assign positive weights to the predictors of interest.

Reducing ntree, max.tree, or max.rules.tree reduces computation. Increasing nodesize produces larger comparison groups with less local detail. The parallel and cores arguments control lasso-fold parallelism; forest threading is controlled by **randomForestSRC**.

Value

An object of class "varpro" with the following components:

rf	The rule-generating rfsrc forest.
split.weight	Named weights for the retained predictor columns, or NULL when computed and custom weighting are both absent.
split.weight.raw	List of untransformed preliminary components actually calculated, named lasso, tree, or vimp. Empty when preliminary estimation is skipped.
results	Rule-level data frame with tree and branch identifiers, released-variable index, original-region sample size n.oob, and importance columns. Indices refer to xvar.names. Classification includes class-specific counts and scores; multivariate regression includes one score per response. importance supplies the variable-level summary.
x	Complete-case processed predictor data, including columns removed by preliminary screening. Encoding information is stored as attributes.
xvar.names	Processed predictor names retained for rule generation.
xvar.org.names	Original predictor names before encoding.
y	Working response: numeric response(s), recoded class labels, or external survival summary values.
y.org	Original response values for the retained observations. For survival, these are the observed times and event indicators.
yvar.names	Names of the working response columns.
family	Family of the rule-generating forest: "regr", "regr+", "class", or "surv". Multiple RMST horizons use "regr+".
max.rules.tree, max.tree	Stored rule-extraction limits used by subsequent importance calculations.
model.info	Compact description of the analysis. Its observations component records input, retained, and omitted row counts for the data supplied to this varpro call.

Note

For reproducible serial analyses, call set.seed immediately before varpro and use parallel = FALSE. The seed argument controls only the forest calls to which it is passed. Rule resampling and lasso folds also use random numbers. Calling importance again can resample rules; retain a summary for repeated inspection of the same calculation.

The ... controls are consumed by specific stages. General rfsrc arguments such as na.action, splitrule, and perf.type are not forwarded wholesale. Perform any desired imputation before calling varpro.

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:2409.09003. doi:10.48550/arXiv.2409.09003.

Friedman, J., Hastie, T. and Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1), 1–22. doi:10.18637/jss.v033.i01.

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

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

See Also

importance.varpro, cv.varpro, predict.varpro, ivarpro, uvarpro, partialpro, outpro, isopro, randomForestSRC::rfsrc, glmnet::cv.glmnet.

Examples

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


## Nonlinear regression with five signal and five noise predictors.
set.seed(137)
n <- 300
x <- matrix(runif(n * 10), nrow = n)
colnames(x) <- sprintf("x%02d", seq_len(ncol(x)))
mu <- 10 * sin(pi * x[, 1] * x[, 2]) +
      20 * (x[, 3] - 0.5)^2 + 10 * x[, 4] + 5 * x[, 5]
d <- data.frame(y = mu + rnorm(n), x)
signal <- colnames(x)[1:5]

## Shallow-tree-guided rule generation.
o <- varpro(y ~ ., d, ntree = 25, ntree.reduce = 25,
            split.weight.method = "tree", nodesize = 10,
            max.tree = 25, max.rules.tree = 20, parallel = FALSE)
imp <- importance(o)
print(data.frame(variable = rownames(imp), z = imp$z,
                 signal = rownames(imp) %in% signal))

## Automatic weighting with more trees and comparisons.
set.seed(137)
o <- varpro(y ~ ., d, ntree = 150, max.tree = 100,
            max.rules.tree = 50, nfolds = 5, parallel = FALSE)
imp <- importance(o, plot.it = TRUE)
print(imp)
print(get.vimp(o))
```

```
print(get.vimp(o, pretty = FALSE))
print(get.topvars(o))

## Choose a variable set using predictive performance.
set.seed(137)
## Add cv.folds = 5 to assess selection and obtain stability summaries.
cv <- cv.varpro(y ~ ., d, ntree = 100,
                zcut = seq(0.1, 3, length.out = 12), nblocks = 6,
                max.tree = 75, max.rules.tree = 40,
                nfolds = 5, parallel = FALSE)
print(cv$imp)
print(cv$imp.conserve)

## Hot-encoding: compare processed-column and original-variable scores.
## Several predictors are converted to factors before analysis.
data(BostonHousing, package = "mlbench")
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))
set.seed(137)
b <- varpro(medv ~ ., Boston, ntree = 100,
            max.tree = 75, max.rules.tree = 40,
            nfolds = 5, parallel = FALSE)

## Calculate the processed-column importance
set.seed(137)
b.imp <- importance(b)
print(b.imp)

## Map max scores back to original predictors.
set.seed(137)
b.org <- get.orgvimp(b)
print(b.org)
print(get.orgvimp(b, vmp = b.imp)) ## should be identical; no seed required

## Include every original predictor, filling absent scores with zero.
set.seed(137)
print(get.orgvimp(b, pretty = FALSE))


## Multiclass classification: overall and class-specific importance.
data(wine, package = "randomForestSRC")
wine$quality <- factor(wine$quality)
set.seed(137)
w <- varpro(quality ~ ., wine, ntree = 100,
            max.tree = 75, max.rules.tree = 40,
            nfolds = 5, parallel = FALSE)
w.imp <- importance(w)
print(w.imp$unconditional)
print(w.imp$conditional.z)
```

```
## Survival: canonical example
data(peakVO2, package = "randomForestSRC")
s <- varpro(Surv(ttodead, died)~., peakVO2)
print(importance(s))


## Survival: prioritize variables affecting RMST through 500 days.
data(pbc, package = "randomForestSRC")
pbc <- na.omit(pbc)
set.seed(137)
s <- varpro(Surv(days, status) ~ ., pbc, rmst = 500,
            ntree = 100, max.tree = 75, max.rules.tree = 40,
            nfolds = 5, parallel = FALSE)
print(importance(s))
print(get.orgvimp(s))
print(get.vimp(s, pretty = FALSE))

## Multiple horizons give one importance summary per horizon.
set.seed(137)
s.multi <- varpro(Surv(days, status) ~ ., pbc,
                  rmst = c(500, 1000), ntree = 100,
                  max.tree = 75, max.rules.tree = 40,
                  nfolds = 5, parallel = FALSE)
print(importance(s.multi))
```

`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 return: "importance" for release importance, "complement" for the complementary-region response summary, "oob" for the original-region OOB response summary, or "none" to omit response summaries.
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
vs <- varpro.strength(object = o, max.rules.tree = 10, max.tree = 15)

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

Index