Package 'varPro'

December 11, 2025

Version 1.0.0

Date 2025-12-07

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, umap, 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 2025-12-11 19:20:02 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)
```

clusterpro *ClusterPro for Unsupervised Data Visualization*

Description

ClusterPro for unsupervised data visualization using Varpro rules.

Usage

```
clusterpro(data,
           method = c("auto", "unsupv", "rnd"),
           ntree = 100, nodesize = NULL,
           max.rules.tree = 40, max.tree = 40,
           papply = mclapply, verbose = FALSE, seed = NULL, ...)
```

Arguments

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

Details

Unsupervised data visualization tool based on the VarPro framework. For each VarPro rule and its complementary region, a two-class analysis is performed to estimate regression coefficients that quantify variable importance relative to the release variable. These coefficients are then used to scale the centroids of the two regions.

The resulting scaled centroids from all rule pairs form an enhanced learning data set for the release variable. This transformed data can be passed to standard visualization tools (e.g., UMAP or t-SNE) to explore structure and relationships in the original high-dimensional space.

Value

Object containing transformed data which can be passed to standard visualization tools.

Author(s)

Hemant Ishwaran

See Also

plot.clusterpro uvarpro

Examples

```
##------------------------------------------------------------------
##
## V-cluster simulation
##
##------------------------------------------------------------------


vcsim <- function(m=500, p=9, std=.2) {
  p <- max(p, 2)
  n <- 2 * m
  x <- runif(n, 0, 1)
  y <- rep(NA, n)
  y[1:m] <- x[1:m] + rnorm(m, sd = std)
  y[(m+1):n] <- -x[(m+1):n] + rnorm(m, sd = std)
  data.frame(x = x,
             y = y,
             z = matrix(runif(n * p, 0, 1), n))
}

dvc <- vcsim()
ovc <- clusterpro(dvc)
oldpar<-par(mfrow=c(3,3));plot(ovc,1:9);par(oldpar)
oldpar<-par(mfrow=c(3,3));plot(ovc,1:9,col.names="x");par(oldpar)

##------------------------------------------------------------------
##
## 4-cluster simulation
##
##------------------------------------------------------------------


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


fourcsim <- function(n=500, sigma=2) {

  cl1 <- mvrnorm(n,c(0,4),cbind(c(1,0),c(0,sigma)))
  cl2 <- mvrnorm(n,c(4,0),cbind(c(1,0),c(0,sigma)))
  cl3 <- mvrnorm(n,c(0,-4),cbind(c(1,0),c(0,sigma)))
  cl4 <- mvrnorm(n,c(-4,0),cbind(c(1,0),c(0,sigma)))
  dta <- data.frame(rbind(cl1,cl2,cl3,cl4))
  colnames(dta) <- c("x","y")
```

```
  data.frame(dta, noise=matrix(rnorm((n*4)*20),ncol=20))

}

d4c <- fourcsim()
o4c <- clusterpro(d4c)
oldpar<-par(mfrow=c(2,2));plot(o4c,1:4);par(oldpar)

}

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

lvsim <- function(n=1000, q=2, qnoise=15, noise=FALSE)  {
  w <- rnorm(n)
  x <- rnorm(n)
  y <- rnorm(n)
  z <- rnorm(n)
  ei <- matrix(rnorm(n * q * 4, sd = sqrt(.1)), ncol = q * 4)
  e1 <- rnorm(n, sd = sqrt(.4))
  e2 <- rnorm(n, sd = sqrt(.4))
  wi <- w + ei[, 1:q]
  xi <- x + ei[, (q+1):(2*q)]
  yi <- y + ei[, (2*q+1):(3*q)]
  zi <- z + ei[, (3*q+1):(4*q)]
  h1 <- w + x + e1
  h2 <- y + z + e2
  dta <- data.frame(w=w,wi=wi,x=x,xi=xi,y=y,yi=yi,z=z,zi=zi,h1=h1,h2=h2)
  if (noise) {
    dta <- data.frame(dta, noise = matrix(rnorm(n * qnoise), ncol = qnoise))
  }
  dta
}

dlc <- lvsim()
olc <- clusterpro(dlc)
oldpar<-par(mfrow=c(4,4));plot(olc,col.names="w");par(oldpar)


##---------------------------------------------------------------------
##
## Glass mlbench data
##
##---------------------------------------------------------------------

data(Glass, package = "mlbench")
dg <- Glass

## with class label
og <- clusterpro(dg)
```

```
oldpar<-par(mfrow=c(4,4));plot(og,1:16);par(oldpar)

## without class label
dgU <- Glass; dgU$Type <- NULL
ogU <- clusterpro(dgU)
oldpar<-par(mfrow=c(3,3));plot(ogU,1:9);par(oldpar)
```

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

Description

Selects Cutoff Value for Variable Priority (VarPro).

Usage

```
cv.varpro(f, 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),
       papply = mclapply, verbose = FALSE, seed = NULL,
       fast = FALSE, crps = FALSE, ...)
```

Arguments

`f`	Model formula specifying the outcome and predictors.
`data`	Training data set (data frame).
`nvar`	Maximum number of variables to return.
`ntree`	Number of trees to grow.
`local.std`	Use locally standardized importance values?
`zcut`	Grid of positive cutoff values used for selecting top variables.
`nblocks`	Number of blocks (folds) for cross-validation.
`split.weight`	Use guided tree-splitting? Variables are selected for splitting with probability proportional to split-weights, obtained by default from a preliminary lasso+tree step.
`split.weight.method`	Character string or vector specifying how split-weights are generated. Defaults to lasso+tree.
`sparse`	Use sparse split-weights?
`nodesize`	Minimum terminal node size. If not specified, an internal function sets the value based on sample size and data dimension.
`max.rules.tree`	Maximum number of rules per tree.

max.tree	Maximum number of trees used for rule extraction.
papply	Apply function used for R parallelization.
verbose	Print verbose output?
seed	Seed for reproducibility.
fast	Use rfsrc.fast in place of rfsrc? May improve speed at the cost of accuracy.
crps	Use CRPS (continuous ranked probability score) instead of Harrell's C-index for evaluating survival performance? Applies only to survival families.
...	Additional arguments passed to varpro.

Details

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

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

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

Value

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

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

Author(s)

Min Lu and Hemant Ishwaran

References

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

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

See Also

importance.varpro uvarpro varpro

Examples

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

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

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

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

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

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

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

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

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

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

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

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

## ------------------------------------------------------------
## iris
```

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

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

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

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

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

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

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

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

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

}


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

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


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

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

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

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

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

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

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

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

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

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

`gliomas` *Diffuse Adult Glioma*

Description

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

References

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

Examples

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

`importance.varpro` *Calculate VarPro Importance*

Description

Calculates variable importance using results from previous varpro call.

Usage

```
## S3 method for class 'varpro'
importance(o, 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,
  papply = mclapply,
  ...)
```

Arguments

`o`	`varpro` object returned from a previous call to `varpro`.
`local.std`	Logical. If `TRUE`, uses locally standardized importance values.
`y.external`	Optional user-supplied response vector. Must match the expected dimension and outcome family.
`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 `varpro` object if unspecified.
`max.tree`	Optional. Maximum number of trees used for rule extraction. Defaults to the value from the `varpro` object if unspecified.
`papply`	Apply method for parallelization; typically `mclapply` or `lapply`.
`...`	Additional arguments passed to internal methods.

Details

Calculates standardized importance values for identifying and ranking variables. Optionally, graphical output is provided, including confidence-style boxplots.

Value

Invisibly, 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`.

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`

Examples

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

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



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

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

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

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

data(BostonHousing, package = "mlbench")

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

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

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



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

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

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

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

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

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

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

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

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

## ------------------------------------------------------------
## illustrates y-external: survival
```

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

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

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

`isopro` *Identify Anomalous Data*

Description

Use isolation forests to identify rare/anomalous data.

Usage

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

Arguments

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

Details

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

There are several ways to apply the method:

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

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

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

Value

Trained isolation forest and anomaly scores.

Author(s)

Min Lu and Hemant Ishwaran

References

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

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

See Also

`predict.isopro` `uvarpro` `varpro`

Examples

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

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

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

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

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


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

data(BostonHousing, package = "mlbench")

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

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

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

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

data(BostonHousing, package = "mlbench")

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

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


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

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

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

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

```
  c(tail(i.rnd$howbad,1),
    tail(i.uns$howbad,1),
    tail(i.aut$howbad,1))
}))


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

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

Description

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

Usage

```
ivarpro(object,
        cut = NULL,
        cut.max = 1,
        ncut = 21,
        nmin = 20, nmax = 150,
        y.external = NULL,
        noise.na = TRUE,
        papply = mclapply,
        max.rules.tree = NULL,
        max.tree = NULL,
        use.loo = TRUE,
        adaptive = FALSE)
```

Arguments

`object`	`varpro` object from a previous call to `varpro`, or a `rfsrc` object.
`cut`	Optional user-supplied sequence of λ values used to relax the constraint region in the local linear regression model. For continuous release variables, each value in `cut` is calibrated so that `cut = 1` corresponds to one standard deviation of the release coordinate. If `cut` is supplied, it is used as-is and the arguments `cut.max`, `ncut`, and `adaptive` are ignored. For binary or one-hot encoded release variables, the full released region is used and `cut` does not control neighborhood size.

cut.max	Maximum value of the λ grid used to define the local neighborhood for continuous release variables when cut is not supplied. By default, cut is constructed as seq(0, cut.max, length.out = ncut) (or up to a data-adaptive value if adaptive = TRUE). Smaller values of cut.max yield more local, sharper case-specific gradients, while larger values yield smoother, more global behavior.
ncut	Length of the cut grid when cut is not supplied. The default is 21. The grid is constructed as seq(0, cut.max, length.out = ncut) (or up to an adaptively chosen maximum if adaptive = TRUE).
nmin	Minimum number of observations required for fitting a local linear model.
nmax	Maximum number of observations allowed for fitting a local linear model. Internally, nmax is capped at 10% of the sample size.
y.external	Optional user-supplied response vector or matrix to use as the dependent variable in the local linear regression. Must have the same number of rows as the feature matrix and match the dimension and type expected for the outcome family.
noise.na	Logical. If TRUE (default), gradients for noisy or non-signal variables are set to NA; if FALSE, they are set to zero.
papply	Apply function used for R parallelization.
max.rules.tree	Maximum number of rules per tree. If unspecified, the value from the varpro object is used, while for rfsrc objects, a default value is used.
max.tree	Maximum number of trees used to extract rules. If unspecified, the value from the varpro object is used, while for rfsrc objects, a default value is used.
use.loo	Logical. If TRUE (default), leave-one-out cross-validation is used to select the best neighborhood size (i.e., the best value in cut) for each rule and release variable. If FALSE, the neighborhood is chosen to use the largest available sample that satisfies nmin and nmax.
adaptive	Logical. If FALSE (default) and cut is not supplied, the cut grid is constructed as seq(0, cut.max, length.out = ncut). If TRUE and cut is not supplied, a data-adaptive upper bound for the neighborhood scale is computed from the sample size using a simple bandwidth-style rule-of-thumb, and cut is constructed as a sequence from 0 to this data-adaptive maximum (subject to cut.max). This provides a convenient way to automatically sharpen the local neighborhood for case-specific gradients when the sample size is moderate to large.

Details

Understanding individual-level (case-specific) variable importance is critical in applications where personalized decisions are required. Traditional variable importance methods focus on average (population-level) effects and often fail to capture heterogeneity across individuals. In many real-world problems, it is not sufficient to determine whether a variable is important on average; we must also understand how it affects each individual prediction.

The VarPro framework identifies feature-space regions through rule-based splitting and computes importance using only observed data. This avoids biases introduced by permutation or synthetic data, leading to robust, population-level importance estimates. However, VarPro does not directly capture individual-level effects.

To address this limitation, individual variable priority (iVarPro) extends VarPro by estimating, for each case and each variable identified by the VarPro analysis, a local gradient that quantifies how small changes in that variable influence that case's predicted outcome. Formally, for each observation $i = 1, \ldots, n$ and each predictor j that appears as a release variable in the VarPro rule set, iVarPro returns a case-specific importance score measuring the sensitivity of the prediction for case i to perturbations in variable j, holding the other variables locally fixed.

iVarPro leverages the release region concept from VarPro. A region R is first defined using VarPro rules. Since using only data within R often results in insufficient sample size for stable gradient estimation, iVarPro releases R along a coordinate s. This means the constraint on s is removed while all others are held fixed, yielding additional variation specifically in the s-direction, precisely what is needed to compute directional derivatives.

Local gradients are then estimated via linear regression on the expanded region. For continuous release variables, the parameter `cut` controls the amount of constraint relaxation, where `cut = 1` corresponds to one standard deviation of the release coordinate, calibrated automatically from the data. When `cut` is not supplied, it is constructed as a grid from 0 to `cut.max`. Smaller values of `cut.max` force more local neighborhoods and can sharpen case-specific gradients near discontinuities, while larger values induce smoother, more global behavior.

For binary or one-hot encoded release variables, the gradient is interpreted as a scaled finite difference in the predicted outcome between the two levels (0 and 1), conditional on the rule constraints for the other variables. In this case, the full released region is used, and `cut` is not used to define the local neighborhood along the binary coordinate.

When `use.loo = TRUE`, a leave-one-out cross-validation (LOO) score is computed for each candidate neighborhood defined by `cut` for continuous release variables. The value of `cut` that minimizes the LOO score is selected, providing an adaptive, data-driven choice of expansion region size for each rule and release coordinate. When `use.loo = FALSE`, the expansion region is chosen based only on sample size, favoring the largest neighborhood that satisfies `nmin` and `nmax`.

When `adaptive = TRUE` and `cut` is not supplied, the maximum neighborhood scale is chosen based on the sample size using a simple bandwidth-style rule-of-thumb, and the `cut` grid is constructed up to this data-adaptive maximum (subject to the upper bound `cut.max`). This can be particularly useful in settings with discontinuous or highly nonlinear responses, where a globally wide neighborhood may over-smooth important local features and obscure case-specific effects.

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

Value

For univariate outcomes (and two-class classification treated as a single score), a numeric matrix of dimension $n \times p_*$ containing case-specific (individual-level) variable priority values, where p_* is the number of predictor variables identified by the VarPro analysis (typically those used as release coordinates in the rule set).

- Each row corresponds to a case (observation) in the original data.
- Each column corresponds to a predictor variable identified by the VarPro analysis as having an associated release rule.

The entry in row i and column j is the iVarPro importance score for variable j for case i, measuring the local sensitivity of that case's prediction to changes in that variable. Predictors that are never

used as release variables in the VarPro analysis do not receive a case-specific importance score (or, depending on implementation, may appear with constant NA or zero values).

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.

See Also

varpro

Examples

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

data(peakVO2, package = "randomForestSRC")
o <- varpro(Surv(ttodead, died)~., peakVO2, ntree = 50)
imp <- ivarpro(o, adaptive = TRUE)
ivarpro.shap(imp, o$x)


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

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

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

## iVarPro importance plot
ivarpro.plot <- function(dta, release = 1, combined.range = TRUE,
                         cex = 1.0, cex.title = 1.0, sc = 5.0,
                         gscale = 30, title = NULL) {
  x1 <- dta[, "x1"]
  x2 <- dta[, "x2"]
  x1n <- expression(x^{(1)})
  x2n <- expression(x^{(2)})
  if (release == 1) {
    if (is.null(title))
      title <- bquote("iVarPro Case-Specific Gradient " ~ x^{(1)})
    cex.pt <- dta[, "Importance.x1"]
  }
  else {
    if (is.null(title))
      title <- bquote("iVarPro Case-Specific Gradient " ~ x^{(2)})
    cex.pt <- dta[, "Importance.x2"]
  }
  if (combined.range) {
    cex.pt <- cex.pt / max(dta[, c("Importance.x1", "Importance.x2")],
                           na.rm = TRUE)
  }
  rng <- range(c(x1, x2))
  oldpar <- par(mar = c(4, 5, 5, 1),
                mgp = c(2.25, 1.0, 0),
                bg = "white")
  gscalev <- gscale
  gscale <- paste0("gray", gscale)
  plot(x1, x2, xlab = x1n, ylab = x2n,
       ylim = rng, xlim = rng,
       col = "#FFA500", pch = 19,
       cex = (sc * cex.pt), cex.axis = cex, cex.lab = cex,
       panel.first = rect(par("usr")[1], par("usr")[3],
                          par("usr")[2], par("usr")[4],
                          col = gscale, border = NA))
  abline(a = 0, b = 1,
         lty = 2, col = if (gscalev < 50) "white" else "black")
  mtext(title, cex = cex.title, line = .5)
  par(oldpar)
}
```

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

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

## canonical ivarpro analysis (default cut.max = 1)
imp1 <- ivarpro(o)

## sharper local analysis using a smaller cut.max
imp2 <- ivarpro(o, cut.max = 0.5)

## data-adaptive analysis
imp3 <- ivarpro(o, adaptive = TRUE)

## build data for plotting the results (using x1 and x2)
df.imp1 <- data.frame(Importance = imp1, df[, c("x1", "x2")])
df.imp2 <- data.frame(Importance = imp2, df[, c("x1", "x2")])
df.imp3 <- data.frame(Importance = imp3, df[, c("x1", "x2")])

## plot the case-specific importance surfaces
oldpar <- par(mfrow = c(3, 2))
ivarpro.plot(df.imp1, 1); ivarpro.plot(df.imp1, 2)
ivarpro.plot(df.imp2, 1); ivarpro.plot(df.imp2, 2)
ivarpro.plot(df.imp3, 1); ivarpro.plot(df.imp3, 2)
par(oldpar)
```

`outpro` *Model and subsapce aware out-of-distribution (OOD) scoring with outPro*

Description

`outpro` computes an out-of-distribution (OOD) score for new inputs using a fitted model, integrating variable prioritization and local neighborhoods derived from the model. The procedure is model aware and subspace aware: it scores departures in the coordinates that the model has learned to rely on, rather than relying on a global distance in the full feature space. Applicable across all outcome types.

Usage

```
outpro(object,
       newdata,
       neighbor = NULL,
       distancef = "prod",
       reduce = TRUE,
       cutoff = NULL,
       max.rules.tree = 150,
```

```
        max.tree = 150)

outpro.null(object,
            nulldata = NULL,
            neighbor = NULL,
            distancef = "prod",
            reduce = TRUE,
            cutoff = .79,
            max.rules.tree = 150,
            max.tree = 150)
```

Arguments

object	A fitted varpro object or an rfsrc object with classes c("rfsrc","grow").
newdata	New data to score. If omitted, the training design matrix is used. For varpro objects, encodings are aligned to training with get.hotencode.test.
neighbor	Number of training neighbors per case, as determined by the model structure. If NULL, a default of min(n/10, 5000) is used where n is the number of training rows.
distancef	Distance function for aggregation. One of "prod", "euclidean", "mahalanobis", "manhattan", "minkowski", "kernel". The default is "prod".
reduce	Controls variable selection. If TRUE with a varpro object, uses model based prioritization with threshold cutoff. A character vector selects variables by name. A named numeric vector supplies variable weights. Otherwise all predictors are used with unit weights.
cutoff	Threshold used with varpro variable importance z. If NULL, a default based on the number of predictors is used: .79 when the number of predictors is not large, else 0.
max.rules.tree	Maximum number of rules per tree for neighbor extraction.
max.tree	Maximum number of trees to use for neighbor extraction.
nulldata	For outpro.null, optional data representing an in distribution reference. If omitted, the training design matrix is used.

Details

Out-of-distribution (OOD) detection is essential for determining when a supervised model encounters inputs that differ in ways that matter for prediction. The approach here embeds variable prioritization directly in the detection step, constructing localized, task relevant neighborhoods from the fitted model and aggregating coordinate wise deviations within the selected subspace to obtain a distance value for an input.

For a varpro object, variable prioritization is obtained from the model and controlled by cutoff. For an rfsrc object, all predictors are used unless a reduction is supplied. Distances are computed after standardizing the selected variables with training means and scales. Variables with zero standard deviation in the training data are removed automatically before scoring.

The multiplicative "prod" metric uses a small ϵ to avoid zero multiplicands. Since differences are measured on a standardized scale, ϵ is set automatically by default as a small fraction of the median

absolute coordinate difference across variables and neighbors; users can keep the default or pass a custom value via `out.distance` if calling it directly.

The Mahalanobis option uses absolute differences by design and the covariance of standardized training features. A small ridge is added to the covariance for numerical stability.

Value

`outpro` returns a list with components:

- `distance`: numeric vector of length `nrow(newdata)` with one score per case.
- `distance.object`: ingredients used for distance computation, including
  - `score`: neighbor frames returned by `varpro.strength`.
  - `neighbor`: neighbor count per case.
  - `xvar.names`: selected variable names after zero sd removal.
  - `xvar.wt`: variable weights used after normalization.
  - `dist.xvar`: list of absolute coordinate difference matrices (neighbors by cases) in standardized units.
  - `xorg.scale`, `xnew.scale`: standardized training and test matrices for the selected variables.
  - `means`, `sds`: training means and scales for the selected variables.
  - `dropped.zero.sd.variables`: variables removed due to zero standard deviation in training.
- `distance.args`: list of metric arguments actually used, including `distancef`, `weights.used`, `normalize.weights`, `p`, and `epsilon.used`.
- `score`: the neighbor information returned by `varpro.strength`.
- `neighbor`: neighbor setting used.
- `cutoff`: cutoff used for variable prioritization.
- `oob.bits`: indicator of whether scoring was done on training rows or new data.
- `selected.variables`: the variables used in scoring after all filters.
- `selected.weights`: the normalized squared weights for the selected variables.
- `means`, `sds`: duplicates for convenience.
- `call`: the matched call.

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

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

Background

The method follows a model centered view of out-of-distribution (OOD) detection that is both model aware and subspace aware. Variable prioritization is embedded directly in the detection process to focus on coordinates that matter for prediction and to discount nuisance directions. Scoring does not rely on global feature density estimation. The implementation uses a random forest engine whose rule based structure provides localized neighborhoods reflecting the learned predictive mapping.

See Also

varpro, rfsrc.

Examples

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

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

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

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

---

partialpro *Partial Effects for Variable(s)*

Description

Obtain the partial effect of x-variables from a VarPro analysis.

Usage

```
partialpro(object, xvar.names, nvar,
           target, learner, newdata, method = c("unsupv", "rnd", "auto"),
           verbose = FALSE, papply = mclapply, ...)
```

Arguments

object	varpro object returned from a previous call to varpro.
xvar.names	Names of the x-variables to use.
nvar	Number of variables to include. Defaults to all.
target	For classification, specifies the class for which the partial effect is computed. Can be an integer or character label. Defaults to the last class.
learner	Optional function specifying a user-defined prediction model. See Details.

newdata	Optional data frame containing test features. If not provided, the training data is used.
method	Isolation forest method used for Unlimited Virtual Twins (UVT). Options are "unsupv" (default), "rnd" (pure random splitting), and "auto" (autoencoder). See isopro for details.
verbose	Print verbose output?
papply	Parallel apply method; typically mclapply or lapply.
...	Additional hidden options: "cut", "nsmp", "nvirtual", "nmin", "alpha", "df", "sampsize", "ntree", "nodesize", "mse.tolerance".

Details

Computes partial effects for selected variables based on a VarPro analysis. If a variable was filtered out during VarPro (e.g., due to noise), its partial effect cannot be computed.

Partial effects are derived using predictions from the forest built during VarPro. These predictions are restricted using Unlimited Virtual Twins (UVT), which apply an isolation forest criterion to filter unlikely combinations of partial data. The filtering threshold is governed by the internal cut parameter. Isolation forests are constructed via isopro.

Interpretation of partial effects depends on the outcome type:

- For regression: effects are on the response scale.
- For survival: effects are either on mortality (default) or RMST (if specified in the original varpro call).
- For classification: effects are log-odds for the specified target class.

Partial effects are estimated locally using polynomial linear models fit to the predicted values. The degrees of freedom for the local model are controlled by the df option (default = 2, i.e., quadratic).

By default, predictions use the forest from the VarPro object. Alternatively, users may supply a custom prediction function via learner. This function should accept a data frame of features and return:

- A numeric vector for regression or survival outcomes.
- A matrix of class probabilities (one column per class, in original class order) for classification.
- If newdata is missing, the function should return predictions on the original training data.

See the examples for use cases with external learners, including:

1. Random forest (external to VarPro),
2. Gradient tree boosting,
3. Bayesian Additive Regression Trees (BART).

Value

Named list, with entries containing the partial plot information for a variable.

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

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 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.clusterpro` *Plots for Unsupervised Data Visualization*

Description

Plots for unsupervised data visualization

Usage

```
## S3 method for class 'clusterpro'
plot(x, xvar.names, shrink=TRUE,
   col=TRUE, col.names=NULL, sort=TRUE, cex=FALSE, breaks=10, ... )
```

Arguments

x	clusterpro object returned from a previous call to clusterpro.
xvar.names	Names (or integer indices) of the x-variables to plot. Defaults to all variables.
shrink	Logical. If TRUE, shrinks the release variable to zero.
col	Logical. If TRUE, colors the points in the plot.
col.names	Variable used to color the plots. Defaults to the release variable. Can also be an integer index.
sort	Logical. If TRUE, sorts plots by variable importance.
cex	Numeric value to scale point size.
breaks	Number of breaks used when mapping colors to points.
...	Additional arguments passed to plot.

Details

Generates a two-dimensional visualization using UMAP applied to the enhanced data corresponding to a release variable. This provides a low-dimensional representation of the clustered structure derived from the rule-based transformation of the original data.

Value

No return value, called for the purpose of generating a plot.

Author(s)

Hemant Ishwaran

References

McInnes L., Healy J. and Melville J. (2018). UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. ArXiv e-prints.

See Also

clusterpro

plot.partialpro *Partial Plots for VarPro*

Description

Plot partial effects of x-variable(s) from a VarPro 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	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.
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, partial effect values 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 parametric 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

No return value, called for the purpose of generating a plot.

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 (default)
plot(oo.boston, ylab="parametric est.")

## non-parametric local estimation
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)
```

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

Description

Obtain predicted values on test data for unsupervised forests.

Usage

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

Arguments

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

Details

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

Value

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

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

Author(s)

Min Lu and Hemant Ishwaran

See Also

`uvarpro`

Examples

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

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

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

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

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

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

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

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

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

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

Description

Obtain predicted values on test data for VarPro object.

Usage

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

Arguments

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

Details

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

Value

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

Author(s)

Min Lu and Hemant Ishwaran

References

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

See Also

`varpro`

Examples

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

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

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



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

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

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

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

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

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

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

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

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

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

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

---

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

Description

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

Usage

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

Arguments

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

Details

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

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

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

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

A more sophisticated procedure, described in Zhou et al. (2025), reframes each region-release contrast as a supervised classification task, with membership in the region serving as the class label. Variable effects are estimated using lasso-based logistic regression, and coefficients are aggregated over the collection of region-release tasks. This produces a sparser and often more interpretable assessment of importance compared to the entropy method. Although more computationally intensive, the lasso-driven approach can provide sharper separation of relevant and irrelevant features. See examples below for details.

Value

A uvarpro object.

Author(s)

Min Lu and Hemant Ishwaran

References

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

Zhou L., Lu M. and Ishwaran H. (2025). Variable priority for unsupervised variable selection. PhD dissertation, University of Miami.

See Also

varpro

Examples

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

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



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

data(BostonHousing, package = "mlbench")

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

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

data(BostonHousing, package = "mlbench")

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

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

## load the data
```

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

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

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

## get top variables
get.topvars(o)

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

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


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

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

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


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

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

```
## remove the outcome
Glass$Type <- NULL

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

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

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

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

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

## canonical call
o <- uvarpro(iowa)

## standard importance
print(importance(o))

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

## s-dependent graph
sdependent(beta)

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

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


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

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

  ## xC     x feature data from complementary region
```

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

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

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

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

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


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

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

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

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

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

```
  }))
  if (!is.null(rO)) {
    val <- colMeans(rO, na.rm = TRUE)
    names(val) <- colnames(rO)
    return(val)
  } else {
    return(NULL)
  }
}) |> setNames(names(o$entropy)))

print(imp)


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

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

}
```

---

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

Description

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

Usage

```
varpro(f, 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(),
       papply = mclapply, verbose = FALSE, seed = NULL, ...)
```

Arguments

f	Formula specifying the model to be fit.
data	Data frame containing the training data.
nvar	Maximum number of variables to return.
ntree	Number of trees to grow.

split.weight	Use guided tree-splitting? Candidate variables for splitting are selected with probability proportional to a split-weight, obtained by default from a preliminary lasso+tree step.
split.weight.method	Character string (or vector) specifying method used to generate split-weights. Defaults to lasso+tree. See Details.
sparse	Use sparse split-weights?
nodesize	Minimum terminal node size. If not specified, value is set internally based on sample size and dimension.
max.rules.tree	Maximum number of rules per tree.
max.tree	Maximum number of trees used to extract rules.
parallel	Use parallel execution for lasso folds using **doMC**.
cores	Number of cores for parallel processing. Defaults to parallel::detectCores().
papply	Apply function used for R parallelization (e.g., mclapply or lapply). The default is mclapply, which uses multicore parallelism on Unix-like systems (including macOS) and falls back to serial evaluation on Windows. Users can supply lapply or other apply-like functions if desired.
verbose	Print verbose output?
seed	Seed for reproducibility.
...	Additional arguments for advanced use.

Details

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

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

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

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

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

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

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

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

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

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

Value

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

Author(s)

Min Lu and Hemant Ishwaran

References

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

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

See Also

`alzheimers` `cv.varpro` `glioma` `importance.varpro` `predict.varpro` `isopro` `uvarpro`

Examples

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

## get top variables
get.topvars(o)

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

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

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

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

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

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

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

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

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

data(BostonHousing, package = "mlbench")

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

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

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

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

  ## experimental settings
  n <- 5000
  q <- 20
```

```
  ir <- 6
  f <- as.formula(Class ~ .)

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

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

}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

swt <- c(swt, strange=99999)

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

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

`varpro.news` *Show the NEWS file*

Description

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

Usage

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

Arguments

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

Value

None.

Author(s)

Min Lu and Hemant Ishwaran

---

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

Description

Used to parse values from a VarPro object.

Usage

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

Arguments

object	rfsrc object.
newdata	Optional test data. If provided, returns branch and complementary branch membership of the training data corresponding to the test cases.
m.target	Character string specifying the target outcome for multivariate families. If unspecified, a default is selected automatically.
max.rules.tree	Maximum number of rules extracted per tree.
max.tree	Maximum number of trees used for rule extraction.
stat	Statistic to output. Options include "importance", "complement mean", and "oob mean".

membership	Return out-of-bag and complementary membership indices for each rule?
neighbor	Nearest neighbor parameter, used only when newdata is specified.
seed	Seed for reproducibility.
do.trace	Enable detailed trace output.
...	Additional arguments.

Details

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

Value

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

Examples

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

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

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

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

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

Index