Package ‘SuperLearner’

January 20, 2025

Type Package

Title Super Learner Prediction

Version 2.0-29

Date 2024-02-06

Maintainer Eric Polley <`epolley@uchicago.edu`>

Description Implements the super learner prediction method and contains a library of prediction algorithms to be used in the super learner.

License GPL-3

URL `https://github.com/ecpolley/SuperLearner`

Depends R (>= 2.14.0), nnls, gam (>= 1.15)

Imports cvAUC, methods

Suggests arm, bartMachine, biglasso, bigmemory, caret, class, devtools, e1071, earth, gbm, genefilter, ggplot2, glmnet, ipred, KernelKnn, kernlab, knitr, lattice, LogicReg, MASS, mlbench, nloptr, nnet, party, polspline, prettydoc, quadprog, randomForest, ranger, RhpcBLASctl, ROCR, rmarkdown, rpart, SIS, speedglm, spls, sva, testthat, xgboost (>= 0.6)

LazyLoad yes

VignetteBuilder knitr, rmarkdown

RoxygenNote 6.0.1

NeedsCompilation no

Author Eric Polley [aut, cre],
Erin LeDell [aut],
Chris Kennedy [aut],
Sam Lendle [ctb],
Mark van der Laan [aut, ths]

Repository CRAN

Date/Publication 2024-02-20 17:00:02 UTC

Contents

`create.Learner` *Factory for learner wrappers*

Description

Create custom learners and/or a sequence of learners with hyperparameter combinations defined over a grid.

Usage

```
create.Learner(base_learner, params = list(), tune = list(),
  env = parent.frame(), name_prefix = base_learner, detailed_names = F,
  verbose = F)
```

Arguments

`base_learner`	Character string of the learner function that will be customized.
`params`	List with parameters to customize.
`tune`	List of hyperparameter settings that will define custom learners.
`env`	Environment in which to create the functions. Defaults to the current environment (e.g. often the global environment).
`name_prefix`	The prefix string for the name of each function that is generated.
`detailed_names`	Set to T to have the function names include the parameter configurations.
`verbose`	Display extra details.

Value

Returns a list with expanded tuneGrid and the names of the created functions.

Examples

```
## Not run:
# Create a randomForest learner with ntree set to 1000 rather than the
# default of 500.
create_rf = create.Learner("SL.randomForest", list(ntree = 1000))
create_rf
sl = SuperLearner(Y = Y, X = X, SL.library = create_rf$names, family = binomial())
sl
# Clean up global environment.
rm(list = create_rf$names)
# Create a randomForest learner that optimizes over mtry
create_rf = create.Learner("SL.randomForest",
                     tune = list(mtry = round(c(1, sqrt(ncol(X)), ncol(X)))))
create_rf
sl = SuperLearner(Y = Y, X = X, SL.library = create_rf$names, family = binomial())
sl
```

```
# Clean up global environment.
rm(list = create_rf$names)

# Optimize elastic net over alpha, with a custom environment and detailed names.
learners = new.env()
create_enet = create.Learner("SL.glmnet", env = learners, detailed_names = T,
                            tune = list(alpha = seq(0, 1, length.out=5)))
create_enet
# List the environment to review what functions were created.
ls(learners)
# We can simply list the environment to specify the library.
sl = SuperLearner(Y = Y, X = X, SL.library = ls(learners), family = binomial(), env = learners)
sl

## End(Not run)
```

`create.SL.xgboost` *Factory for XGBoost SL wrappers*

Description

Create multiple configurations of XGBoost learners based on the desired combinations of hyperparameters.

Usage

```
create.SL.xgboost(tune = list(ntrees = c(1000), max_depth = c(4), shrinkage =
  c(0.1), minobspernode = c(10)), detailed_names = F, env = .GlobalEnv,
  name_prefix = "SL.xgb")
```

Arguments

`tune`	List of hyperparameter settings to test. If specified, each hyperparameter will need to be defined.
`detailed_names`	Set to T to have the function names include the parameter configurations.
`env`	Environment in which to create the SL.xgboost functions. Defaults to the global environment.
`name_prefix`	The prefix string for the name of each function that is generated.

Examples

```
# Create a new environment to store the learner functions.
# This keeps the global environment organized.
sl_env = new.env()
# Create 2 * 2 * 1 * 3 = 12 combinations of hyperparameters.
tune = list(ntrees = c(100, 500), max_depth = c(1, 2), minobspernode = 10,
            shrinkage = c(0.1, 0.01, 0.001))
```

```
# Generate a separate learner for each combination.
xgb_grid = create.SL.xgboost(tune = tune, env = sl_env)
# Review the function configurations.
xgb_grid
# Attach the environment so that the custom learner functions can be accessed.
attach(sl_env)
## Not run:
sl = SuperLearner(Y = Y, X = X, SL.library = xgb_grid$names)

## End(Not run)
detach(sl_env)
```

`CV.SuperLearner` *Function to get V-fold cross-validated risk estimate for super learner*

Description

Function to get V-fold cross-validated risk estimate for super learner. This function simply splits the data into V folds and then calls SuperLearner. Most of the arguments are passed directly to SuperLearner.

Usage

```
CV.SuperLearner(Y, X, V = NULL, family = gaussian(), SL.library,
  method = "method.NNLS", id = NULL, verbose = FALSE,
  control = list(saveFitLibrary = FALSE), cvControl = list(),
  innerCvControl = list(),
  obsWeights = NULL, saveAll = TRUE, parallel = "seq", env = parent.frame())
```

Arguments

`Y`	The outcome.
`X`	The covariates.
`V`	The number of folds for `CV.SuperLearner`. This argument will be depreciated and moved into the `cvControl`. If Both `V` and `cvControl` set the number of cross-validation folds, an error message will appear. The recommendation is to use `cvControl`. This is not the number of folds for `SuperLearner`. The number of folds for `SuperLearner` is controlled with `innerCvControl`.
`family`	Currently allows `gaussian` or `binomial` to describe the error distribution. Link function information will be ignored and should be contained in the method argument below.
`SL.library`	Either a character vector of prediction algorithms or a list containing character vectors. See details below for examples on the structure. A list of functions included in the SuperLearner package can be found with `listWrappers()`.

`method`	A list (or a function to create a list) containing details on estimating the coefficients for the super learner and the model to combine the individual algorithms in the library. See `?method.template` for details. Currently, the built in options are either "method.NNLS" (the default), "method.NNLS2", "method.NNloglik", "method.CC_LS", "method.CC_nloglik", or "method.AUC". NNLS and NNLS2 are non-negative least squares based on the Lawson-Hanson algorithm and the dual method of Goldfarb and Idnani, respectively. NNLS and NNLS2 will work for both gaussian and binomial outcomes. NNloglik is a non-negative binomial likelihood maximization using the BFGS quasi-Newton optimization method. NN* methods are normalized so weights sum to one. CC_LS uses Goldfarb and Idnani's quadratic programming algorithm to calculate the best convex combination of weights to minimize the squared error loss. CC_nloglik calculates the convex combination of weights that minimize the negative binomial log likelihood on the logistic scale using the sequential quadratic programming algorithm. AUC, which only works for binary outcomes, uses the Nelder-Mead method via the optim function to minimize rank loss (equivalent to maximizing AUC).
`id`	Optional cluster identification variable. For the cross-validation splits, `id` forces observations in the same cluster to be in the same validation fold. `id` is passed to the prediction and screening algorithms in SL.library, but be sure to check the individual wrappers as many of them ignore the information.
`verbose`	Logical; TRUE for printing progress during the computation (helpful for debugging).
`control`	A list of parameters to control the estimation process. Parameters include `saveFitLibrary` and `trimLogit`. See `SuperLearner.control` for details.
`cvControl`	A list of parameters to control the outer cross-validation process. The outer cross-validation is the sample spliting for evaluating the SuperLearner. Parameters include `V`, `stratifyCV`, `shuffle` and `validRows`. See `SuperLearner.CV.control` for details.
`innerCvControl`	A list of lists of parameters to control the inner cross-validation process. It should have `V` elements in the list, each a valid `cvControl` list. If only a single value, then replicated across all folds. The inner cross-validation are the values passed to each of the `V` `SuperLearner` calls. Parameters include `V`, `stratifyCV`, `shuffle` and `validRows`. See `SuperLearner.CV.control` for details.
`obsWeights`	Optional observation weights variable. As with `id` above, `obsWeights` is passed to the prediction and screening algorithms, but many of the built in wrappers ignore (or can't use) the information. If you are using observation weights, make sure the library you specify uses the information.
`saveAll`	Logical; Should the entire `SuperLearner` object be saved for each fold?
`parallel`	Options for parallel computation of the V-fold step. Use "seq" (the default) for sequential computation. `parallel = 'multicore'` to use `mclapply` for the V-fold step (but note that `SuperLearner()` will still be sequential). The default for mclapply is to check the `mc.cores` option, and if not set to default to 2 cores. Be sure to set `options()$mc.cores` to the desired number of cores if you don't want the default. Or `parallel` can be the name of a snow cluster and will use `parLapply` for the V-fold step. For both multicore and snow, the inner `SuperLearner` calls will be sequential.

`env`	Environment containing the learner functions. Defaults to the calling environment.

Details

The `SuperLearner` function builds a estimator, but does not contain an estimate on the performance of the estimator. Various methods exist for estimator performance evaluation. If you are familiar with the super learner algorithm, it should be no surprise we recommend using cross-validation to evaluate the honest performance of the super learner estimator. The function `CV.SuperLearner` computes the usual V-fold cross-validated risk estimate for the super learner (and all algorithms in `SL.library` for comparison).

Value

An object of class `CV.SuperLearner` (a list) with components:

`call`	The matched call.
`AllSL`	If `saveAll = TRUE`, a list with output from each call to `SuperLearner`, otherwise NULL.
`SL.predict`	The predicted values from the super learner when each particular row was part of the validation fold.
`discreteSL.predict`	The traditional cross-validated selector. Picks the algorithm with the smallest cross-validated risk (in super learner terms, gives that algorithm coefficient 1 and all others 0).
`whichDiscreteSL`	A list of length `V`. The elements in the list are the algorithm that had the smallest cross-validated risk estimate for that fold.
`library.predict`	A matrix with the predicted values from each algorithm in `SL.library`. The columns are the algorithms in `SL.library` and the rows represent the predicted values when that particular row was in the validation fold (i.e. not used to fit that estimator).
`coef`	A matrix with the coefficients for the super learner on each fold. The columns are the algorithms in `SL.library` the rows are the folds.
`folds`	A list containing the row numbers for each validation fold.
`V`	Number of folds for `CV.SuperLearner`.
`libraryNames`	A character vector with the names of the algorithms in the library. The format is 'predictionAlgorithm_screeningAlgorithm' with '_All' used to denote the prediction algorithm run on all variables in X.
`SL.library`	Returns `SL.library` in the same format as the argument with the same name above.
`method`	A list with the method functions.
`Y`	The outcome

Author(s)

Eric C Polley `<polley.eric@mayo.edu>`

See Also

SuperLearner

Examples

```
## Not run:
set.seed(23432)
## training set
n <- 500
p <- 50
X <- matrix(rnorm(n*p), nrow = n, ncol = p)
colnames(X) <- paste("X", 1:p, sep="")
X <- data.frame(X)
Y <- X[, 1] + sqrt(abs(X[, 2] * X[, 3])) + X[, 2] - X[, 3] + rnorm(n)

## build Library and run Super Learner
SL.library <- c("SL.glm", "SL.randomForest", "SL.gam", "SL.polymars", "SL.mean")

test <- CV.SuperLearner(Y = Y, X = X, V = 10, SL.library = SL.library,
  verbose = TRUE, method = "method.NNLS")
test
summary(test)
## Look at the coefficients across folds
coef(test)

# Example with specifying cross-validation options for both
# CV.SuperLearner (cvControl) and the internal SuperLearners (innerCvControl)
test <- CV.SuperLearner(Y = Y, X = X, SL.library = SL.library,
  cvControl = list(V = 10, shuffle = FALSE),
  innerCvControl = list(list(V = 5)),
  verbose = TRUE, method = "method.NNLS")

## examples with snow
library(parallel)
cl <- makeCluster(2, type = "PSOCK") # can use different types here
clusterSetRNGStream(cl, iseed = 2343)
testSNOW <- CV.SuperLearner(Y = Y, X = X, SL.library = SL.library, method = "method.NNLS",
  parallel = cl)
summary(testSNOW)
stopCluster(cl)

## End(Not run)
```

`CVFolds` *Generate list of row numbers for each fold in the cross-validation*

Description

Generate list of row numbers for each fold in the cross-validation. `CVFolds` is used in the `SuperLearner` to create the cross-validation splits.

Usage

```
CVFolds(N, id, Y, cvControl)
```

Arguments

`N`	Sample size
`id`	Optional cluster id variable. If present, all observations in the same cluster will always be in the same split.
`Y`	outcome
`cvControl`	Control parameters for the cross-validation step. See `SuperLearner.CV.control` for details.

Value

`validRows`	A list of length V where each element in the list is a vector with the row numbers of the corresponding validation sample.

Author(s)

Eric C Polley <`polley.eric@mayo.edu`>

`listWrappers` *list all wrapper functions in SuperLearner*

Description

List all wrapper functions in `SuperLearner` package

Usage

```
listWrappers(what = "both")
```

Arguments

`what`	What list to return. Can be `both` for both prediction algorithms and screening algorithms, `SL` for the prediction algorithms, `screen` for the screening algorithms, `method` for the estimation method details, or anything else will return a list of all (exported) functions in the `SuperLearner` package. Additional wrapper functions are available at `https://github.com/ecpolley/SuperLearnerExtra`.

Value

Invisible character vector with all exported functions in the SuperLearner package

Author(s)

Eric C Polley `<epolley@uchicago.edu>`

See Also

`SuperLearner`

Examples

```
listWrappers(what = "SL")
listWrappers(what = "screen")
```

`plot.CV.SuperLearner` *Graphical display of the V-fold CV risk estimates*

Description

The function plots the V-fold cross-validated risk estimates for the super learner, the discrete super learner and each algorithm in the library. By default the estimates will be sorted and include an asymptotic 95% confidence interval.

Usage

```
## S3 method for class 'CV.SuperLearner'
plot(x, package = "ggplot2", constant = qnorm(0.975), sort = TRUE, ...)
```

Arguments

`x`	The output from `CV.SuperLearner`.
`package`	Either "ggplot2" or "lattice". The package selected must be available.
`constant`	A numeric value. The confidence interval is defined as p +/- constant * se, where p is the point estimate and se is the standard error. The default is the quantile of the standard normal corresponding to a 95% CI.
`sort`	Logical. Should the rows in the plot be sorted from the smallest to the largest point estimate. If FALSE, then the order is super learner, discrete super learner, then the estimators in `SL.library`.
`...`	Additional arguments for `summary.CV.SuperLearner`

Details

see summary.CV.SuperLearner for details on how the estimates are computed

Value

Returns the plot (either a ggplot2 object (class `ggplot`) or a lattice object (class `trellis`))

Author(s)

Eric C Polley `<epolley@uchicago.edu>`

See Also

`summary.CV.SuperLearner` and `CV.SuperLearner`

`predict.SL.bartMachine` *bartMachine prediction*

Description

bartMachine prediction

Usage

```
## S3 method for class 'SL.bartMachine'
predict(object, newdata, family, X = NULL,
  Y = NULL, ...)
```

Arguments

`object`	SuperLearner object
`newdata`	Dataframe to predict the outcome
`family`	"gaussian" for regression, "binomial" for binary classification. (Not used)
`X`	Covariate dataframe (not used)
`Y`	Outcome variable (not used)
`...`	Additional arguments (not used)

`predict.SL.biglasso` *Prediction wrapper for SL.biglasso*

Description

Prediction wrapper for SL.biglasso objects.

Usage

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

Arguments

`object`	SL.kernlab object
`newdata`	Dataframe to generate predictions
`...`	Unused additional arguments

See Also

`SL.biglasso` `biglasso` `predict.biglasso`

`predict.SL.glm` *Prediction for SL.glm*

Description

Prediction for SL.glm

Usage

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

Arguments

`object`	SL.glm object
`newdata`	Dataframe to generate predictions
`...`	Unused additional arguments

See Also

`SL.glm` `glm` `predict.glm` `SL.speedglm`

`predict.SL.glmnet` *Prediction for an SL.glmnet object*

Description

Prediction for the glmnet wrapper.

Usage

```
## S3 method for class 'SL.glmnet'
predict(object, newdata, remove_extra_cols = T,
  add_missing_cols = T, ...)
```

Arguments

`object`	Result object from SL.glmnet
`newdata`	Dataframe or matrix that will generate predictions.
`remove_extra_cols`	Remove any extra columns in the new data that were not part of the original model.
`add_missing_cols`	Add any columns from original data that do not exist in the new data, and set values to 0.
`...`	Any additional arguments (not used).

See Also

`SL.glmnet`

`predict.SL.kernelKnn` *Prediction for SL.kernelKnn*

Description

Prediction for SL.kernelKnn

Usage

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

Arguments

`object`	SL.kernelKnn object
`newdata`	Dataframe to generate predictions
`...`	Unused additional arguments

`predict.SL.ksvm` *Prediction for SL.ksvm*

Description

Prediction for SL.ksvm

Usage

```
## S3 method for class 'SL.ksvm'
predict(object, newdata, family, coupler = "minpair", ...)
```

Arguments

`object`	SL.kernlab object
`newdata`	Dataframe to generate predictions
`family`	Gaussian or binomial
`coupler`	Coupling method used in the multiclass case, can be one of minpair or pkpd (see kernlab package for details). For future usage.
`...`	Unused additional arguments

See Also

`SL.ksvm` `ksvm` `predict.ksvm`

`predict.SL.lda` *Prediction wrapper for SL.lda*

Description

Prediction wrapper for SL.lda

Usage

```
## S3 method for class 'SL.lda'
predict(object, newdata, prior = object$object$prior,
  dimen = NULL, method = "plug-in", ...)
```

Arguments

`object`	SL.lda object
`newdata`	Dataframe to generate predictions
`prior`	The prior probabilities of the classes, by default the proportions in the training set or what was set in the call to lda.
`dimen`	the dimension of the space to be used. If this is less than min(p, ng-1), only the first dimen discriminant components are used (except for method="predictive"), and only those dimensions are returned in x.
`method`	This determines how the parameter estimation is handled. With "plug-in" (the default) the usual unbiased parameter estimates are used and assumed to be correct. With "debiased" an unbiased estimator of the log posterior probabilities is used, and with "predictive" the parameter estimates are integrated out using a vague prior.
`...`	Unused additional arguments

See Also

`SL.lda` `lda` `predict.lda`

`predict.SL.lm` *Prediction for SL.lm*

Description

Prediction for SL.lm

Usage

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

Arguments

`object`	SL.lm object
`newdata`	Dataframe to generate predictions
`...`	Unused additional arguments

See Also

`SL.lm` `lm` `predict.lm` `SL.speedlm`

`predict.SL.qda` *Prediction wrapper for SL.qda*

Description

Prediction wrapper for SL.qda

Usage

```
## S3 method for class 'SL.qda'
predict(object, newdata, prior = object$object$prior,
  dimen = NULL, method = "plug-in", ...)
```

Arguments

`object`	SL.lda object
`newdata`	Dataframe to generate predictions
`prior`	The prior probabilities of the classes, by default the proportions in the training set or what was set in the call to lda.
`dimen`	the dimension of the space to be used. If this is less than min(p, ng-1), only the first dimen discriminant components are used (except for method="predictive"), and only those dimensions are returned in x.
`method`	This determines how the parameter estimation is handled. With "plug-in" (the default) the usual unbiased parameter estimates are used and assumed to be correct. With "debiased" an unbiased estimator of the log posterior probabilities is used, and with "predictive" the parameter estimates are integrated out using a vague prior.
`...`	Unused additional arguments

See Also

`SL.qda` `qda` `predict.qda`

`predict.SL.ranger` *Prediction wrapper for ranger random forests*

Description

Prediction wrapper for SL.ranger objects.

Usage

```
## S3 method for class 'SL.ranger'
predict(object, newdata, family, num.threads = 1,
  verbose = object$verbose, ...)
```

Arguments

`object`	SL.kernlab object
`newdata`	Dataframe to generate predictions
`family`	Gaussian or binomial
`num.threads`	Number of threads used for parallelization
`verbose`	If TRUE output additional information during execution.
`...`	Unused additional arguments

See Also

`SL.ranger` `ranger` `predict.ranger`

`predict.SL.speedglm` *Prediction for SL.speedglm*

Description

Prediction for SL.speedglm

Usage

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

Arguments

`object`	SL.speedglm object
`newdata`	Dataframe to generate predictions
`...`	Unused additional arguments

See Also

`SL.speedglm` `speedglm` `predict.speedglm`

`predict.SL.speedlm` *Prediction for SL.speedlm*

Description

Prediction for SL.speedlm, a fast lm()

Usage

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

Arguments

`object`	SL.speedlm object
`newdata`	Dataframe to generate predictions
`...`	Unused additional arguments

See Also

`SL.speedlm` `speedlm` `predict.speedlm` `SL.speedglm`

`predict.SL.xgboost` *XGBoost prediction on new data*

Description

XGBoost prediction on new data

Usage

```
## S3 method for class 'SL.xgboost'
predict(object, newdata, family, ...)
```

Arguments

`object`	Model fit object from SuperLearner
`newdata`	Dataframe that will be converted to an xgb.DMatrix
`family`	Binomial or gaussian
`...`	Any remaining arguments (not supported though).

`predict.SuperLearner` *Predict method for SuperLearner object*

Description

Obtains predictions on a new data set from a SuperLearner fit. May require the original data if one of the library algorithms uses the original data in its predict method.

Usage

```
## S3 method for class 'SuperLearner'
predict(object, newdata, X = NULL, Y = NULL,
  onlySL = FALSE, ...)
```

Arguments

`object`	Fitted object from `SuperLearner`
`newdata`	New X values for prediction
`X`	Original data set used to fit `object`, if needed by fit object.
`Y`	Original outcome used to fit `object`, if needed by fit object.
`onlySL`	Logical. If TRUE, only compute predictions for algorithms with non-zero coefficients in the super learner object. Default is FALSE (computes predictions for all algorithms in library).
`...`	Additional arguments passed to the `predict.SL.*` functions

Details

If `newdata` is omitted the predicted values from `object` are returned. Each algorithm in the Super Learner library needs to have a corresponding prediction function with the “predict.” prefixed onto the algorithm name (e.g. `predict.SL.glm` for `SL.glm`).

Value

`pred`	Predicted values from Super Learner fit
`library.predict`	Predicted values for each algorithm in library

Author(s)

Eric C Polley <`epolley@uchicago.edu`>

See Also

`SuperLearner`

`recombineCVSL` *Recombine a CV.SuperLearner fit using a new metalearning method*

Description

Function to re-compute the V-fold cross-validated risk estimate for super learner using a new metalearning method. This function takes as input an existing CV.SuperLearner fit and applies the `recombineSL` fit to each of the V Super Learner fits.

Usage

```
recombineCVSL(object, method = "method.NNloglik", verbose = FALSE,
  saveAll = TRUE, parallel = "seq")
```

Arguments

`object`	Fitted object from `CV.SuperLearner`.
`method`	A list (or a function to create a list) containing details on estimating the coefficients for the super learner and the model to combine the individual algorithms in the library. See `?method.template` for details. Currently, the built in options are either "method.NNLS" (the default), "method.NNLS2", "method.NNloglik", "method.CC_LS", "method.CC_nloglik", or "method.AUC". NNLS and NNLS2 are non-negative least squares based on the Lawson-Hanson algorithm and the dual method of Goldfarb and Idnani, respectively. NNLS and NNLS2 will work for both gaussian and binomial outcomes. NNloglik is a non-negative binomial likelihood maximization using the BFGS quasi-Newton optimization method. NN* methods are normalized so weights sum to one. CC_LS uses Goldfarb and Idnani's quadratic programming algorithm to calculate the best convex combination of weights to minimize the squared error loss. CC_nloglik calculates the convex combination of weights that minimize the negative binomial log likelihood on the logistic scale using the sequential quadratic programming algorithm. AUC, which only works for binary outcomes, uses the Nelder-Mead method via the optim function to minimize rank loss (equivalent to maximizing AUC).
`verbose`	logical; TRUE for printing progress during the computation (helpful for debugging).
`saveAll`	Logical; Should the entire `SuperLearner` object be saved for each fold?
`parallel`	Options for parallel computation of the V-fold step. Use "seq" (the default) for sequential computation. `parallel = 'multicore'` to use `mclapply` for the V-fold step (but note that `SuperLearner()` will still be sequential). Or `parallel` can be the name of a snow cluster and will use `parLapply` for the V-fold step. For both multicore and snow, the inner `SuperLearner` calls will be sequential.

Details

The function `recombineCVSL` computes the usual V-fold cross-validated risk estimate for the super learner (and all algorithms in `SL.library` for comparison), using a newly specified metalearning method. The weights for each algorithm in `SL.library` are re-estimated using the new metalearner, however the base learner fits are not regenerated, so this function saves a lot of computation time as opposed to using the `CV.SuperLearner` function with a new `method` argument. The output is identical to the output from the `CV.SuperLearner` function.

Value

An object of class `CV.SuperLearner` (a list) with components:

`call`	The matched call.
`AllSL`	If `saveAll = TRUE`, a list with output from each call to `SuperLearner`, otherwise NULL.
`SL.predict`	The predicted values from the super learner when each particular row was part of the validation fold.
`discreteSL.predict`	The traditional cross-validated selector. Picks the algorithm with the smallest cross-validated risk (in super learner terms, gives that algorithm coefficient 1 and all others 0).
`whichDiscreteSL`	A list of length `V`. The elements in the list are the algorithm that had the smallest cross-validated risk estimate for that fold.
`library.predict`	A matrix with the predicted values from each algorithm in `SL.library`. The columns are the algorithms in `SL.library` and the rows represent the predicted values when that particular row was in the validation fold (i.e. not used to fit that estimator).
`coef`	A matrix with the coefficients for the super learner on each fold. The columns are the algorithms in `SL.library` the rows are the folds.
`folds`	A list containing the row numbers for each validation fold.
`V`	Number of folds for `CV.SuperLearner`.
`libraryNames`	A character vector with the names of the algorithms in the library. The format is 'predictionAlgorithm_screeningAlgorithm' with '_All' used to denote the prediction algorithm run on all variables in X.
`SL.library`	Returns `SL.library` in the same format as the argument with the same name above.
`method`	A list with the method functions.
`Y`	The outcome

Author(s)

Erin LeDell <`ledell@berkeley.edu`>

See Also

recombineSL

Examples

```
## Not run:

# Binary outcome example adapted from SuperLearner examples

set.seed(1)
N <- 200
X <- matrix(rnorm(N*10), N, 10)
X <- as.data.frame(X)
Y <- rbinom(N, 1, plogis(.2*X[, 1] + .1*X[, 2] - .2*X[, 3] +
  .1*X[, 3]*X[, 4] - .2*abs(X[, 4])))

SL.library <- c("SL.glmnet", "SL.glm", "SL.knn", "SL.gam", "SL.mean")

# least squares loss function
set.seed(1) # for reproducibility
cvfit_nnls <- CV.SuperLearner(Y = Y, X = X, V = 10, SL.library = SL.library,
  verbose = TRUE, method = "method.NNLS", family = binomial())
cvfit_nnls$coef
#    SL.glmnet_All SL.glm_All  SL.knn_All SL.gam_All SL.mean_All
# 1      0.0000000 0.00000000 0.000000000  0.4143862   0.5856138
# 2      0.0000000 0.00000000 0.304802397  0.3047478   0.3904498
# 3      0.0000000 0.00000000 0.002897533  0.5544075   0.4426950
# 4      0.0000000 0.20322642 0.000000000  0.1121891   0.6845845
# 5      0.1743973 0.00000000 0.032471026  0.3580624   0.4350693
# 6      0.0000000 0.00000000 0.099881535  0.3662309   0.5338876
# 7      0.0000000 0.00000000 0.234876082  0.2942472   0.4708767
# 8      0.0000000 0.06424676 0.113988158  0.5600208   0.2617443
# 9      0.0000000 0.00000000 0.338030342  0.2762604   0.3857093
# 10     0.3022442 0.00000000 0.294226204  0.1394534   0.2640762


# negative log binomial likelihood loss function
cvfit_nnloglik <- recombineCVSL(cvfit_nnls, method = "method.NNloglik")
cvfit_nnloglik$coef
#    SL.glmnet_All SL.glm_All SL.knn_All SL.gam_All SL.mean_All
# 1      0.0000000  0.0000000 0.00000000  0.5974799  0.40252010
# 2      0.0000000  0.0000000 0.31177345  0.6882266  0.00000000
# 3      0.0000000  0.0000000 0.01377469  0.8544238  0.13180152
# 4      0.0000000  0.1644188 0.00000000  0.2387919  0.59678930
# 5      0.2142254  0.0000000 0.00000000  0.3729426  0.41283197
# 6      0.0000000  0.0000000 0.00000000  0.5847150  0.41528502
# 7      0.0000000  0.0000000 0.47538172  0.5080311  0.01658722
# 8      0.0000000  0.0000000 0.00000000  1.0000000  0.00000000
# 9      0.0000000  0.0000000 0.45384961  0.2923480  0.25380243
# 10     0.3977816  0.0000000 0.27927906  0.1606384  0.16230097

# If we use the same seed as the original `cvfit_nnls`, then
```

```
# the recombineCVSL and CV.SuperLearner results will be identical
# however, the recombineCVSL version will be much faster since
# it doesn't have to re-fit all the base learners, V times each.
set.seed(1)
cvfit_nnloglik2 <- CV.SuperLearner(Y = Y, X = X, V = 10, SL.library = SL.library,
  verbose = TRUE, method = "method.NNloglik", family = binomial())
cvfit_nnloglik2$coef
#    SL.glmnet_All SL.glm_All SL.knn_All SL.gam_All SL.mean_All
# 1      0.0000000  0.0000000 0.00000000  0.5974799  0.40252010
# 2      0.0000000  0.0000000 0.31177345  0.6882266  0.00000000
# 3      0.0000000  0.0000000 0.01377469  0.8544238  0.13180152
# 4      0.0000000  0.1644188 0.00000000  0.2387919  0.59678930
# 5      0.2142254  0.0000000 0.00000000  0.3729426  0.41283197
# 6      0.0000000  0.0000000 0.00000000  0.5847150  0.41528502
# 7      0.0000000  0.0000000 0.47538172  0.5080311  0.01658722
# 8      0.0000000  0.0000000 0.00000000  1.0000000  0.00000000
# 9      0.0000000  0.0000000 0.45384961  0.2923480  0.25380243
# 10     0.3977816  0.0000000 0.27927906  0.1606384  0.16230097


## End(Not run)
```

`recombineSL` *Recombine a SuperLearner fit using a new metalearning method*

Description

The `recombineSL` function takes an existing SuperLearner fit and a new metalearning method and returns a new SuperLearner fit with updated base learner weights.

Usage

```
recombineSL(object, Y, method = "method.NNloglik", verbose = FALSE)
```

Arguments

`object`	Fitted object from `SuperLearner`.
`Y`	The outcome in the training data set. Must be a numeric vector.
`method`	A list (or a function to create a list) containing details on estimating the coefficients for the super learner and the model to combine the individual algorithms in the library. See `?method.template` for details. Currently, the built in options are either "method.NNLS" (the default), "method.NNLS2", "method.NNloglik", "method.CC_LS", "method.CC_nloglik", or "method.AUC". NNLS and NNLS2 are non-negative least squares based on the Lawson-Hanson algorithm and the dual method of Goldfarb and Idnani, respectively. NNLS and NNLS2 will work for both gaussian and binomial outcomes. NNloglik is a non-negative binomial likelihood maximization using the BFGS quasi-Newton optimization method. NN* methods are normalized so weights sum to one. CC_LS uses Goldfarb and

	Idnani's quadratic programming algorithm to calculate the best convex combination of weights to minimize the squared error loss. CC_nloglik calculates the convex combination of weights that minimize the negative binomial log likelihood on the logistic scale using the sequential quadratic programming algorithm. AUC, which only works for binary outcomes, uses the Nelder-Mead method via the optim function to minimize rank loss (equivalent to maximizing AUC).
`verbose`	logical; TRUE for printing progress during the computation (helpful for debugging).

Details

`recombineSL` re-fits the super learner prediction algorithm using a new metalearning method. The weights for each algorithm in `SL.library` are re-estimated using the new metalearner, however the base learner fits are not regenerated, so this function saves a lot of computation time as opposed to using the `SuperLearner` function with a new `method` argument. The output is identical to the output from the `SuperLearner` function.

Value

`call`	The matched call.
`libraryNames`	A character vector with the names of the algorithms in the library. The format is 'predictionAlgorithm_screeningAlgorithm' with '_All' used to denote the prediction algorithm run on all variables in X.
`SL.library`	Returns `SL.library` in the same format as the argument with the same name above.
`SL.predict`	The predicted values from the super learner for the rows in `newX`.
`coef`	Coefficients for the super learner.
`library.predict`	A matrix with the predicted values from each algorithm in `SL.library` for the rows in `newX`.
`Z`	The Z matrix (the cross-validated predicted values for each algorithm in `SL.library`).
`cvRisk`	A numeric vector with the V-fold cross-validated risk estimate for each algorithm in `SL.library`. Note that this does not contain the CV risk estimate for the SuperLearner, only the individual algorithms in the library.
`family`	Returns the `family` value from above
`fitLibrary`	A list with the fitted objects for each algorithm in `SL.library` on the full training data set.
`varNames`	A character vector with the names of the variables in `X`.
`validRows`	A list containing the row numbers for the V-fold cross-validation step.
`method`	A list with the method functions.
`whichScreen`	A logical matrix indicating which variables passed each screening algorithm.
`control`	The `control` list.
`cvControl`	The `cvControl` list.

`errorsInCVLibrary`	A logical vector indicating if any algorithms experienced an error within the CV step.
`errorsInLibrary`	A logical vector indicating if any algorithms experienced an error on the full data.

Author(s)

Erin LeDell <`ledell@berkeley.edu`>

References

van der Laan, M. J., Polley, E. C. and Hubbard, A. E. (2008) Super Learner, *Statistical Applications of Genetics and Molecular Biology*, **6**, article 25.

Examples

```
## Not run:

# Binary outcome example adapted from SuperLearner examples

set.seed(1)
N <- 200
X <- matrix(rnorm(N*10), N, 10)
X <- as.data.frame(X)
Y <- rbinom(N, 1, plogis(.2*X[, 1] + .1*X[, 2] - .2*X[, 3] +
  .1*X[, 3]*X[, 4] - .2*abs(X[, 4])))

SL.library <- c("SL.glmnet", "SL.glm", "SL.knn", "SL.gam", "SL.mean")

# least squares loss function
set.seed(1) # for reproducibility
fit_nnls <- SuperLearner(Y = Y, X = X, SL.library = SL.library,
  verbose = TRUE, method = "method.NNLS", family = binomial())
fit_nnls
#                    Risk       Coef
# SL.glmnet_All 0.2439433 0.01293059
# SL.glm_All    0.2461245 0.08408060
# SL.knn_All    0.2604000 0.09600353
# SL.gam_All    0.2471651 0.40761918
# SL.mean_All   0.2486049 0.39936611


# negative log binomial likelihood loss function
fit_nnloglik <- recombineSL(fit_nnls, Y = Y, method = "method.NNloglik")
fit_nnloglik
#                    Risk      Coef
# SL.glmnet_All 0.6815911 0.1577228
# SL.glm_All    0.6918926 0.0000000
# SL.knn_All          Inf 0.0000000
# SL.gam_All    0.6935383 0.6292881
```

```
# SL.mean_All   0.6904050 0.2129891

# If we use the same seed as the original `fit_nnls`, then
# the recombineSL and SuperLearner results will be identical
# however, the recombineSL version will be much faster since
# it doesn't have to re-fit all the base learners.
set.seed(1)
fit_nnloglik2 <- SuperLearner(Y = Y, X = X, SL.library = SL.library,
  verbose = TRUE, method = "method.NNloglik", family = binomial())
fit_nnloglik2
#                    Risk      Coef
# SL.glmnet_All 0.6815911 0.1577228
# SL.glm_All    0.6918926 0.0000000
# SL.knn_All          Inf 0.0000000
# SL.gam_All    0.6935383 0.6292881
# SL.mean_All   0.6904050 0.2129891


## End(Not run)
```

`SampleSplitSuperLearner`

Super Learner Prediction Function

Description

A Prediction Function for the Super Learner. The `SuperLearner` function takes a training set pair (X,Y) and returns the predicted values based on a validation set. SampleSplitSuperLearner uses sample split validation whereas SuperLearner uses V-fold cross-validation.

Usage

```
SampleSplitSuperLearner(Y, X, newX = NULL, family = gaussian(), SL.library,
  method = "method.NNLS", id = NULL, verbose = FALSE,
  control = list(), split = 0.8, obsWeights = NULL)
```

Arguments

`Y`	The outcome in the training data set. Must be a numeric vector.
`X`	The predictor variables in the training data set, usually a data.frame.
`newX`	The predictor variables in the validation data set. The structure should match X. If missing, uses X for newX.
`SL.library`	Either a character vector of prediction algorithms or a list containing character vectors. See details below for examples on the structure. A list of functions included in the SuperLearner package can be found with `listWrappers()`.
`verbose`	logical; TRUE for printing progress during the computation (helpful for debugging).

`family`	Currently allows `gaussian` or `binomial` to describe the error distribution. Link function information will be ignored and should be contained in the method argument below.
`method`	A list (or a function to create a list) containing details on estimating the coefficients for the super learner and the model to combine the individual algorithms in the library. See `?method.template` for details. Currently, the built in options are either "method.NNLS" (the default), "method.NNLS2", "method.NNloglik", "method.CC_LS", or "method.CC_nloglik". NNLS and NNLS2 are non-negative least squares based on the Lawson-Hanson algorithm and the dual method of Goldfarb and Idnani, respectively. NNLS and NNLS2 will work for both gaussian and binomial outcomes. NNloglik is a non-negative binomial likelihood maximization using the BFGS quasi-Newton optimization method. NN* methods are normalized so weights sum to one. CC_LS uses Goldfarb and Idnani's quadratic programming algorithm to calculate the best convex combination of weights to minimize the squared error loss. CC_nloglik calculates the convex combination of weights that minimize the negative binomial log likelihood on the logistic scale using the sequential quadratic programming algorithm.
`id`	Optional cluster identification variable. For the cross-validation splits, `id` forces observations in the same cluster to be in the same validation fold. `id` is passed to the prediction and screening algorithms in SL.library, but be sure to check the individual wrappers as many of them ignore the information.
`obsWeights`	Optional observation weights variable. As with `id` above, `obsWeights` is passed to the prediction and screening algorithms, but many of the built in wrappers ignore (or can't use) the information. If you are using observation weights, make sure the library you specify uses the information.
`control`	A list of parameters to control the estimation process. Parameters include `saveFitLibrary` and `trimLogit`. See `SuperLearner.control` for details.
`split`	Either a single value between 0 and 1 indicating the fraction of the samples for the training split. A value of 0.8 will randomly assign 80 percent of the samples to the training split and the other 20 percent to the validation split. Alternatively, split can be a numeric vector with the row numbers of `X` corresponding to the validation split. All other rows not in the vector will be considered in the training split.

Details

`SuperLearner` fits the super learner prediction algorithm. The weights for each algorithm in `SL.library` is estimated, along with the fit of each algorithm.

The prescreen algorithms. These algorithms first rank the variables in `X` based on either a univariate regression p-value of the `randomForest` variable importance. A subset of the variables in `X` is selected based on a pre-defined cut-off. With this subset of the X variables, the algorithms in `SL.library` are then fit.

The SuperLearner package contains a few prediction and screening algorithm wrappers. The full list of wrappers can be viewed with `listWrappers()`. The design of the SuperLearner package is such that the user can easily add their own wrappers. We also maintain a website with additional examples of wrapper functions at `https://github.com/ecpolley/SuperLearnerExtra`.

Value

`call`	The matched call.
`libraryNames`	A character vector with the names of the algorithms in the library. The format is 'predictionAlgorithm_screeningAlgorithm' with '_All' used to denote the prediction algorithm run on all variables in X.
`SL.library`	Returns `SL.library` in the same format as the argument with the same name above.
`SL.predict`	The predicted values from the super learner for the rows in `newX`.
`coef`	Coefficients for the super learner.
`library.predict`	A matrix with the predicted values from each algorithm in `SL.library` for the rows in `newX`.
`Z`	The Z matrix (the cross-validated predicted values for each algorithm in `SL.library`).
`cvRisk`	A numeric vector with the V-fold cross-validated risk estimate for each algorithm in `SL.library`. Note that this does not contain the CV risk estimate for the SuperLearner, only the individual algorithms in the library.
`family`	Returns the `family` value from above
`fitLibrary`	A list with the fitted objects for each algorithm in `SL.library` on the full training data set.
`varNames`	A character vector with the names of the variables in `X`.
`validRows`	A list containing the row numbers for the V-fold cross-validation step.
`method`	A list with the method functions.
`whichScreen`	A logical matrix indicating which variables passed each screening algorithm.
`control`	The `control` list.
`split`	The `split` value.
`errorsInCVLibrary`	A logical vector indicating if any algorithms experienced an error within the CV step.
`errorsInLibrary`	A logical vector indicating if any algorithms experienced an error on the full data.

Author(s)

Eric C Polley <`epolley@uchicago.edu`>

References

van der Laan, M. J., Polley, E. C. and Hubbard, A. E. (2008) Super Learner, *Statistical Applications of Genetics and Molecular Biology*, **6**, article 25.

Examples

```
## Not run:
## simulate data
set.seed(23432)
## training set
n <- 500
p <- 50
X <- matrix(rnorm(n*p), nrow = n, ncol = p)
colnames(X) <- paste("X", 1:p, sep="")
X <- data.frame(X)
Y <- X[, 1] + sqrt(abs(X[, 2] * X[, 3])) + X[, 2] - X[, 3] + rnorm(n)

## test set
m <- 1000
newX <- matrix(rnorm(m*p), nrow = m, ncol = p)
colnames(newX) <- paste("X", 1:p, sep="")
newX <- data.frame(newX)
newY <- newX[, 1] + sqrt(abs(newX[, 2] * newX[, 3])) + newX[, 2] -
  newX[, 3] + rnorm(m)

# generate Library and run Super Learner
SL.library <- c("SL.glm", "SL.randomForest", "SL.gam",
  "SL.polymars", "SL.mean")
test <- SampleSplitSuperLearner(Y = Y, X = X, newX = newX, SL.library = SL.library,
  verbose = TRUE, method = "method.NNLS")
test

# library with screening
SL.library <- list(c("SL.glmnet", "All"), c("SL.glm", "screen.randomForest",
  "All", "screen.SIS"), "SL.randomForest", c("SL.polymars", "All"), "SL.mean")
test <- SuperLearner(Y = Y, X = X, newX = newX, SL.library = SL.library,
  verbose = TRUE, method = "method.NNLS")
test

# binary outcome
set.seed(1)
N <- 200
X <- matrix(rnorm(N*10), N, 10)
X <- as.data.frame(X)
Y <- rbinom(N, 1, plogis(.2*X[, 1] + .1*X[, 2] - .2*X[, 3] +
  .1*X[, 3]*X[, 4] - .2*abs(X[, 4])))

SL.library <- c("SL.glmnet", "SL.glm", "SL.knn", "SL.gam", "SL.mean")

# least squares loss function
test.NNLS <- SampleSplitSuperLearner(Y = Y, X = X, SL.library = SL.library,
  verbose = TRUE, method = "method.NNLS", family = binomial())
test.NNLS

## End(Not run)
```

`SL.bartMachine` *Wrapper for bartMachine learner*

Description

Support bayesian additive regression trees via the bartMachine package.

Usage

```
SL.bartMachine(Y, X, newX, family, obsWeights, id, num_trees = 50,
  num_burn_in = 250, verbose = F, alpha = 0.95, beta = 2, k = 2,
  q = 0.9, nu = 3, num_iterations_after_burn_in = 1000, ...)
```

Arguments

`Y`	Outcome variable
`X`	Covariate dataframe
`newX`	Optional dataframe to predict the outcome
`family`	"gaussian" for regression, "binomial" for binary classification
`obsWeights`	Optional observation-level weights (supported but not tested)
`id`	Optional id to group observations from the same unit (not used currently).
`num_trees`	The number of trees to be grown in the sum-of-trees model.
`num_burn_in`	Number of MCMC samples to be discarded as "burn-in".
`verbose`	Prints information about progress of the algorithm to the screen.
`alpha`	Base hyperparameter in tree prior for whether a node is nonterminal or not.
`beta`	Power hyperparameter in tree prior for whether a node is nonterminal or not.
`k`	For regression, k determines the prior probability that E(Y|X) is contained in the interval (y_min, y_max), based on a normal distribution. For example, when k=2, the prior probability is 95%. For classification, k determines the prior probability that E(Y|X) is between (-3,3). Note that a larger value of k results in more shrinkage and a more conservative fit.
`q`	Quantile of the prior on the error variance at which the data-based estimate is placed. Note that the larger the value of q, the more aggressive the fit as you are placing more prior weight on values lower than the data-based estimate. Not used for classification.
`nu`	Degrees of freedom for the inverse chi^2 prior. Not used for classification.
`num_iterations_after_burn_in`	Number of MCMC samples to draw from the posterior distribution of f(x).
`...`	Additional arguments (not used)

`SL.biglasso` *SL wrapper for biglasso*

Description

SL wrapper for biglasso

Usage

```
SL.biglasso(Y, X, newX, family, obsWeights, penalty = "lasso",
  alg.logistic = "Newton", screen = "SSR", alpha = 1, nlambda = 100,
  eval.metric = "default", ncores = 1, nfolds = 5, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Gaussian or binomial
`obsWeights`	Observation-level weights
`penalty`	The penalty to be applied to the model. Either "lasso" (default), "ridge", or "enet" (elastic net).
`alg.logistic`	The algorithm used in logistic regression. If "Newton" then the exact hessian is used (default); if "MM" then a majorization-minimization algorithm is used to set an upper-bound on the hessian matrix. This can be faster, particularly in data-larger-than-RAM case.
`screen`	"SSR" (default) is the sequential strong rule; "SEDPP" is the (sequential) EDPP rule. "SSR-BEDPP", "SSR-Dome", and "SSR-Slores" are our newly proposed screening rules which combine the strong rule with a safe rule (BEDPP, Dome test, or Slores rule). Among the three, the first two are for lasso-penalized linear regression, and the last one is for lasso-penalized logistic regression. "None" is to not apply a screening rule.
`alpha`	The elastic-net mixing parameter that controls the relative contribution from the lasso (l1) and the ridge (l2) penalty.
`nlambda`	The number of lambda values to check. Default is 100.
`eval.metric`	The evaluation metric for the cross-validated error and for choosing optimal `lambda`. "default" for linear regression is MSE (mean squared error), for logistic regression is misclassification error. "MAPE", for linear regression only, is the Mean Absolute Percentage Error.
`ncores`	The number of cores to use for parallel execution across a cluster created by the `parallel` package.
`nfolds`	The number of cross-validation folds. Default is 5.
`...`	Any additional arguments, not currently used.

References

Zeng Y, Breheny P (2017). biglasso: Extending Lasso Model Fitting to Big Data. https://CRAN.R-project.org/package=biglasso.

See Also

`predict.SL.biglasso` `biglasso` `cv.biglasso` `predict.biglasso` `SL.glmnet`

Examples

```
data(Boston, package = "MASS")
Y = Boston$medv
# Remove outcome from covariate dataframe.
X = Boston[, -14]

set.seed(1)

# Sample rows to speed up example.
row_subset = sample(nrow(X), 30)

# Subset rows and columns & use only 2 folds to speed up example.
sl = SuperLearner(Y[row_subset], X[row_subset, 1:2, drop = FALSE],
                  family = gaussian(), cvControl = list(V = 2),
                  SL.library = "SL.biglasso")
sl

pred = predict(sl, X)
summary(pred$pred)
```

`SL.cforest` *cforest (party)*

Description

These defaults emulate cforest_unbiased() but allow customization.

Usage

```
SL.cforest(Y, X, newX, family, obsWeights, id, ntree = 1000,
  mtry = max(floor(ncol(X)/3), 1), mincriterion = 0, teststat = "quad",
  testtype = "Univ", replace = F, fraction = 0.632, ...)
```

Arguments

`Y`	Outcome variable
`X`	Covariate dataframe
`newX`	Optional dataframe to predict the outcome

`family`	"gaussian" for regression, "binomial" for binary classification
`obsWeights`	Optional observation-level weights (supported but not tested)
`id`	Optional id to group observations from the same unit (not used currently).
`ntree`	Number of trees
`mtry`	Number of randomly selected features per node
`mincriterion`	See ?cforest_control
`teststat`	See ?cforest_control
`testtype`	See ?cforest_control
`replace`	See ?cforest_control
`fraction`	See ?cforest_control
`...`	Remaining arguments (unused)

`SL.glm` *Wrapper for glm*

Description

Wrapper for generalized linear models via glm().

Note that for outcomes bounded by [0, 1] the binomial family can be used in addition to gaussian.

Usage

```
SL.glm(Y, X, newX, family, obsWeights, model = TRUE, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Gaussian or binomial
`obsWeights`	Observation-level weights
`model`	Whether to save model.matrix of data in fit object. Set to FALSE to save memory.
`...`	Any remaining arguments, not used.

References

Fox, J. (2015). Applied regression analysis and generalized linear models. Sage Publications.

See Also

`predict.SL.glm` `glm` `predict.glm` `SL.speedglm`

Examples

```
data(Boston, package = "MASS")
Y = Boston$medv
# Remove outcome from covariate dataframe.
X = Boston[, -14]

set.seed(1)

sl = SuperLearner(Y, X, family = gaussian(),
                  SL.library = c("SL.mean", "SL.glm"))

print(sl)
```

`SL.glmnet` *Elastic net regression, including lasso and ridge*

Description

Penalized regression using elastic net. Alpha = 0 corresponds to ridge regression and alpha = 1 corresponds to Lasso.

See `vignette("glmnet_beta", package = "glmnet")` for a nice tutorial on glmnet.

Usage

```
SL.glmnet(Y, X, newX, family, obsWeights, id, alpha = 1, nfolds = 10,
  nlambda = 100, useMin = TRUE, loss = "deviance", ...)
```

Arguments

`Y`	Outcome variable
`X`	Covariate dataframe
`newX`	Dataframe to predict the outcome
`family`	"gaussian" for regression, "binomial" for binary classification. Untested options: "multinomial" for multiple classification or "mgaussian" for multiple response, "poisson" for non-negative outcome with proportional mean and variance, "cox".
`obsWeights`	Optional observation-level weights
`id`	Optional id to group observations from the same unit (not used currently).
`alpha`	Elastic net mixing parameter, range [0, 1]. 0 = ridge regression and 1 = lasso.
`nfolds`	Number of folds for internal cross-validation to optimize lambda.
`nlambda`	Number of lambda values to check, recommended to be 100 or more.
`useMin`	If TRUE use lambda that minimizes risk, otherwise use 1 standard-error rule which chooses a higher penalty with performance within one standard error of the minimum (see Breiman et al. 1984 on CART for background).

`loss`	Loss function, can be "deviance", "mse", or "mae". If family = binomial can also be "auc" or "class" (misclassification error).
`...`	Any additional arguments are passed through to cv.glmnet.

References

Friedman, J., Hastie, T., & Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. Journal of statistical software, 33(1), 1.

Hoerl, A. E., & Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. Technometrics, 12(1), 55-67.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society. Series B (Methodological), 267-288.

Zou, H., & Hastie, T. (2005). Regularization and variable selection via the elastic net. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 67(2), 301-320.

See Also

`predict.SL.glmnet` `cv.glmnet` `glmnet`

Examples

```
# Load a test dataset.
data(PimaIndiansDiabetes2, package = "mlbench")
data = PimaIndiansDiabetes2

# Omit observations with missing data.
data = na.omit(data)

Y = as.numeric(data$diabetes == "pos")
X = subset(data, select = -diabetes)

set.seed(1, "L'Ecuyer-CMRG")

sl = SuperLearner(Y, X, family = binomial(),
                  SL.library = c("SL.mean", "SL.glm", "SL.glmnet"))
sl
```

`SL.kernelKnn` *SL wrapper for KernelKNN*

Description

Wrapper for a configurable implementation of k-nearest neighbors. Supports both binomial and gaussian outcome distributions.

Usage

```
SL.kernelKnn(Y, X, newX, family, k = 10, method = "euclidean",
  weights_function = NULL, extrema = F, h = 1, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Gaussian or binomial
`k`	Number of nearest neighbors to use
`method`	Distance method, can be 'euclidean' (default), 'manhattan', 'chebyshev', 'canberra', 'braycurtis', 'pearson_correlation', 'simple_matching_coefficient', 'minkowski' (by default the order 'p' of the minkowski parameter equals k), 'hamming', 'mahalanobis', 'jaccard_coefficient', 'Rao_coefficient'
`weights_function`	Weighting method for combining the nearest neighbors. Can be 'uniform' (default), 'triangular', 'epanechnikov', 'biweight', 'triweight', 'tricube', 'gaussian', 'cosine', 'logistic', 'gaussianSimple', 'silverman', 'inverse', 'exponential'.
`extrema`	if TRUE then the minimum and maximum values from the k-nearest-neighbors will be removed (can be thought as outlier removal).
`h`	the bandwidth, applicable if the weights_function is not NULL. Defaults to 1.0.
`...`	Any additional parameters, not currently passed through.

Value

List with predictions and the original training data & hyperparameters.

Examples

```
# Load a test dataset.
data(PimaIndiansDiabetes2, package = "mlbench")

data = PimaIndiansDiabetes2

# Omit observations with missing data.
data = na.omit(data)

Y_bin = as.numeric(data$diabetes)
X = subset(data, select = -diabetes)

set.seed(1)

sl = SuperLearner(Y_bin, X, family = binomial(),
                  SL.library = c("SL.mean", "SL.kernelKnn"))
sl
```

`SL.ksvm` *Wrapper for Kernlab's SVM algorithm*

Description

Wrapper for Kernlab's support vector machine algorithm.

Usage

```
SL.ksvm(Y, X, newX, family, type = NULL, kernel = "rbfdot",
  kpar = "automatic", scaled = T, C = 1, nu = 0.2, epsilon = 0.1,
  cross = 0, prob.model = family$family == "binomial",
  class.weights = NULL, cache = 40, tol = 0.001, shrinking = T, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Gaussian or binomial
`type`	ksvm can be used for classification , for regression, or for novelty detection. Depending on whether y is a factor or not, the default setting for type is C-svc or eps-svr, respectively, but can be overwritten by setting an explicit value. See ?ksvm for more details.
`kernel`	the kernel function used in training and predicting. This parameter can be set to any function, of class kernel, which computes the inner product in feature space between two vector arguments. See ?ksvm for more details.
`kpar`	the list of hyper-parameters (kernel parameters). This is a list which contains the parameters to be used with the kernel function. See ?ksvm for more details.
`scaled`	A logical vector indicating the variables to be scaled. If scaled is of length 1, the value is recycled as many times as needed and all non-binary variables are scaled. Per default, data are scaled internally (both x and y variables) to zero mean and unit variance. The center and scale values are returned and used for later predictions.
`C`	cost of constraints violation (default: 1) this is the 'C'-constant of the regularization term in the Lagrange formulation.
`nu`	parameter needed for nu-svc, one-svc, and nu-svr. The nu parameter sets the upper bound on the training error and the lower bound on the fraction of data points to become Support Vectors (default: 0.2).
`epsilon`	epsilon in the insensitive-loss function used for eps-svr, nu-svr and eps-bsvm (default: 0.1)
`cross`	if a integer value k>0 is specified, a k-fold cross validation on the training data is performed to assess the quality of the model: the accuracy rate for classification and the Mean Squared Error for regression

`prob.model`	if set to TRUE builds a model for calculating class probabilities or in case of regression, calculates the scaling parameter of the Laplacian distribution fitted on the residuals. Fitting is done on output data created by performing a 3-fold cross-validation on the training data. (default: FALSE)
`class.weights`	a named vector of weights for the different classes, used for asymmetric class sizes. Not all factor levels have to be supplied (default weight: 1). All components have to be named.
`cache`	cache memory in MB (default 40)
`tol`	tolerance of termination criterion (default: 0.001)
`shrinking`	option whether to use the shrinking-heuristics (default: TRUE)
`...`	Any additional parameters, not currently passed through.

Value

List with predictions and the original training data & hyperparameters.

References

Hsu, C. W., Chang, C. C., & Lin, C. J. (2016). A practical guide to support vector classification. `https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`

Scholkopf, B., & Smola, A. J. (2001). Learning with kernels: support vector machines, regularization, optimization, and beyond. MIT press.

Vapnik, V. N. (1998). Statistical learning theory (Vol. 1). New York: Wiley.

Zeileis, A., Hornik, K., Smola, A., & Karatzoglou, A. (2004). kernlab-an S4 package for kernel methods in R. Journal of statistical software, 11(9), 1-20.

See Also

`predict.SL.ksvm` `ksvm` `predict.ksvm`

Examples

```
data(Boston, package = "MASS")
Y = Boston$medv
# Remove outcome from covariate dataframe.
X = Boston[, -14]

set.seed(1)

sl = SuperLearner(Y, X, family = gaussian(),
                  SL.library = c("SL.mean", "SL.ksvm"))
sl

pred = predict(sl, X)
summary(pred$pred)
```

`SL.lda` *SL wrapper for MASS:lda*

Description

Linear discriminant analysis, used for classification.

Usage

```
SL.lda(Y, X, newX, family, obsWeights = rep(1, nrow(X)), id = NULL,
  verbose = F, prior = as.vector(prop.table(table(Y))), method = "mle",
  tol = 1e-04, CV = F, nu = 5, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Binomial only, cannot be used for regression.
`obsWeights`	Observation-level weights
`id`	Not supported.
`verbose`	If TRUE, display additional output during execution.
`prior`	the prior probabilities of class membership. If unspecified, the class proportions for the training set are used. If present, the probabilities should be specified in the order of the factor levels.
`method`	"moment" for standard estimators of the mean and variance, "mle" for MLEs, "mve" to use cov.mve, or "t" for robust estimates based on a t distribution.
`tol`	tolerance
`CV`	If true, returns results (classes and posterior probabilities) for leave-one-out cross-validation. Note that if the prior is estimated, the proportions in the whole dataset are used.
`nu`	degrees of freedom for method = "t".
`...`	Any additional arguments, not currently used.

References

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An Introduction to Statistical Learning (Vol. 6). New York: Springer. Section 4.4.

See Also

`predict.SL.lda` `lda` `predict.lda` `SL.qda`

Examples

```
data(Boston, package = "MASS")
Y = as.numeric(Boston$medv > 23)
# Remove outcome from covariate dataframe.
X = Boston[, -14]

set.seed(1)

# Use only 2 CV folds to speed up example.
sl = SuperLearner(Y, X, family = binomial(), cvControl = list(V = 2),
                  SL.library = c("SL.mean", "SL.lda"))
sl

pred = predict(sl, X)
summary(pred$pred)
```

`SL.lm` *Wrapper for lm*

Description

Wrapper for OLS via lm(), which may be faster than glm().

Usage

```
SL.lm(Y, X, newX, family, obsWeights, model = TRUE, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Gaussian or binomial
`obsWeights`	Observation-level weights
`model`	Whether to save model.matrix of data in fit object. Set to FALSE to save memory.
`...`	Any remaining arguments, not used.

References

Fox, J. (2015). Applied regression analysis and generalized linear models. Sage Publications.

See Also

`predict.SL.lm` `lm` `predict.lm` `SL.speedlm`

Examples

```
data(Boston, package = "MASS")
Y = Boston$medv
# Remove outcome from covariate dataframe.
X = Boston[, -14]

set.seed(1)

sl = SuperLearner(Y, X, family = gaussian(),
                  SL.library = c("SL.mean", "SL.lm"))

print(sl)
```

`SL.qda` *SL wrapper for MASS:qda*

Description

Quadratic discriminant analysis, used for classification.

Usage

```
SL.qda(Y, X, newX, family, obsWeights = rep(1, nrow(X)), verbose = F,
  id = NULL, prior = as.vector(prop.table(table(Y))), method = "mle",
  tol = 1e-04, CV = F, nu = 5, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Binomial only, cannot be used for regression.
`obsWeights`	Observation-level weights
`verbose`	If TRUE, display additional output during execution.
`id`	Not supported.
`prior`	the prior probabilities of class membership. If unspecified, the class proportions for the training set are used. If present, the probabilities should be specified in the order of the factor levels.
`method`	"moment" for standard estimators of the mean and variance, "mle" for MLEs, "mve" to use cov.mve, or "t" for robust estimates based on a t distribution.
`tol`	tolerance
`CV`	If true, returns results (classes and posterior probabilities) for leave-one-out cross-validation. Note that if the prior is estimated, the proportions in the whole dataset are used.
`nu`	degrees of freedom for method = "t".
`...`	Any additional arguments, not currently used.

References

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An Introduction to Statistical Learning (Vol. 6). New York: Springer. Section 4.4.

See Also

`predict.SL.qda` `qda` `predict.qda` `SL.lda`

Examples

```
data(Boston, package = "MASS")
Y = as.numeric(Boston$medv > 23)
# Remove outcome from covariate dataframe.
X = Boston[, -14]

set.seed(1)

# Use only 2 CV folds to speed up example.
sl = SuperLearner(Y, X, family = binomial(), cvControl = list(V = 2),
                  SL.library = c("SL.mean", "SL.qda"))
sl

pred = predict(sl, X)
summary(pred$pred)
```

`SL.ranger` *SL wrapper for ranger*

Description

Ranger is a fast implementation of Random Forest (Breiman 2001) or recursive partitioning, particularly suited for high dimensional data.

Extending code by Eric Polley from the SuperLearnerExtra package.

Usage

```
SL.ranger(Y, X, newX, family, obsWeights, num.trees = 500,
  mtry = floor(sqrt(ncol(X))), write.forest = TRUE,
  probability = family$family == "binomial",
  min.node.size = ifelse(family$family == "gaussian", 5, 1), replace = TRUE,
  sample.fraction = ifelse(replace, 1, 0.632), num.threads = 1,
  verbose = T, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Gaussian or binomial
`obsWeights`	Observation-level weights
`num.trees`	Number of trees.
`mtry`	Number of variables to possibly split at in each node. Default is the (rounded down) square root of the number variables.
`write.forest`	Save ranger.forest object, required for prediction. Set to FALSE to reduce memory usage if no prediction intended.
`probability`	Grow a probability forest as in Malley et al. (2012).
`min.node.size`	Minimal node size. Default 1 for classification, 5 for regression, 3 for survival, and 10 for probability.
`replace`	Sample with replacement.
`sample.fraction`	Fraction of observations to sample. Default is 1 for sampling with replacement and 0.632 for sampling without replacement.
`num.threads`	Number of threads to use.
`verbose`	If TRUE, display additional output during execution.
`...`	Any additional arguments, not currently used.

References

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

Wright, M. N. & Ziegler, A. (2016). ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R. Journal of Statistical Software, in press. http://arxiv.org/abs/1508.04409.

See Also

`SL.ranger` `ranger` `predict.ranger`

Examples

```
data(Boston, package = "MASS")
Y = Boston$medv
# Remove outcome from covariate dataframe.
X = Boston[, -14]

set.seed(1)

# Use only 2 CV folds to speed up example.
sl = SuperLearner(Y, X, family = gaussian(), cvControl = list(V = 2),
                  SL.library = c("SL.mean", "SL.ranger"))
sl
```

```
pred = predict(sl, X)
summary(pred$pred)
```

`SL.speedglm` *Wrapper for speedglm*

Description

Speedglm is a fast version of glm()

Usage

```
SL.speedglm(Y, X, newX, family, obsWeights, maxit = 25, k = 2, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Gaussian or binomial
`obsWeights`	Observation-level weights
`maxit`	Maximum number of iterations before stopping.
`k`	numeric, the penalty per parameter to be used; the default k = 2 is the classical AIC.
`...`	Any remaining arguments, not used.

References

Enea, M. A. R. C. O. (2013). Fitting linear models and generalized linear models with large data sets in R. Statistical Methods for the Analysis of Large Datasets: book of short papers, 411-414.

See Also

`predict.SL.speedglm` `speedglm` `predict.speedglm`

`SL.speedlm` *Wrapper for speedlm*

Description

Speedlm is a fast version of lm()

Usage

```
SL.speedlm(Y, X, newX, family, obsWeights, ...)
```

Arguments

`Y`	Outcome variable
`X`	Training dataframe
`newX`	Test dataframe
`family`	Gaussian or binomial
`obsWeights`	Observation-level weights
`...`	Any remaining arguments, not used.

References

Enea, M. A. R. C. O. (2013). Fitting linear models and generalized linear models with large data sets in R. Statistical Methods for the Analysis of Large Datasets: book of short papers, 411-414.

See Also

`predict.SL.speedlm` `speedlm` `predict.speedlm` `SL.speedglm`

`SL.xgboost` *XGBoost SuperLearner wrapper*

Description

Supports the Extreme Gradient Boosting package for SuperLearnering, which is a variant of gradient boosted machines (GBM).

Usage

```
SL.xgboost(Y, X, newX, family, obsWeights, id, ntrees = 1000, max_depth = 4,
  shrinkage = 0.1, minobspernode = 10, params = list(), nthread = 1,
  verbose = 0, save_period = NULL, ...)
```

Arguments

`Y`	Outcome variable
`X`	Covariate dataframe
`newX`	Optional dataframe to predict the outcome
`family`	"gaussian" for regression, "binomial" for binary classification, "multinomial" for multiple classification (not yet supported).
`obsWeights`	Optional observation-level weights (supported but not tested)
`id`	Optional id to group observations from the same unit (not used currently).
`ntrees`	How many trees to fit. Low numbers may underfit but high numbers may overfit, depending also on the shrinkage.
`max_depth`	How deep each tree can be. 1 means no interactions, aka tree stubs.
`shrinkage`	How much to shrink the predictions, in order to reduce overfitting.
`minobspernode`	Minimum observations allowed per tree node, after which no more splitting will occur.
`params`	Many other parameters can be customized. See `https://xgboost.readthedocs.io/en/latest/parameter.html`
`nthread`	How many threads (cores) should xgboost use. Generally we want to keep this to 1 so that XGBoost does not compete with SuperLearner parallelization.
`verbose`	Verbosity of XGB fitting.
`save_period`	How often (in tree iterations) to save current model to disk during processing. If NULL does not save model, and if 0 saves model at the end.
`...`	Any remaining arguments (not supported though).

Details

The performance of XGBoost, like GBM, is sensitive to the configuration settings. Therefore it is best to create multiple configurations using create.SL.xgboost and allow the SuperLearner to choose the best weights based on cross-validated performance.

If you run into errors please first try installing the latest version of XGBoost from drat as described here: `https://xgboost.readthedocs.io/en/latest/build.html`

`summary.CV.SuperLearner`

Summary Function for Cross-Validated Super Learner

Description

summary method for the `CV.SuperLearner` function

Usage

```
## S3 method for class 'CV.SuperLearner'
summary(object, obsWeights = NULL, ...)

## S3 method for class 'summary.CV.SuperLearner'
print(x, digits, ...)
```

Arguments

`object`	An object of class "CV.SuperLearner", the result of a call to `CV.SuperLearner`.
`x`	An object of class "summary.CV.SuperLearner", the result of a call to `summary.CV.SuperLearner`.
`obsWeights`	Optional vector for observation weights.
`digits`	The number of significant digits to use when printing.
`...`	additional arguments ...

Details

Summary method for `CV.SuperLearner`. Calculates the V-fold cross-validated estimate of either the mean squared error or the -2*log(L) depending on the loss function used.

Value

`summary.CV.SuperLearner` returns a list with components

`call`	The function call from `CV.SuperLearner`
`method`	Describes the loss function used. Currently either least squares of negative log Likelihood.
`V`	Number of folds
`Risk.SL`	Risk estimate for the super learner
`Risk.dSL`	Risk estimate for the discrete super learner (the cross-validation selector)
`Risk.library`	A matrix with the risk estimates for each algorithm in the library
`Table`	A table with the mean risk estimate and standard deviation across the folds for the super learner and all algorithms in the library

Author(s)

Eric C Polley <`eric.polley@nih.gov`>

See Also

`CV.SuperLearner`

`SuperLearner` *Super Learner Prediction Function*

Description

A Prediction Function for the Super Learner. The `SuperLearner` function takes a training set pair (X,Y) and returns the predicted values based on a validation set.

Usage

```
SuperLearner(Y, X, newX = NULL, family = gaussian(), SL.library,
  method = "method.NNLS", id = NULL, verbose = FALSE,
  control = list(), cvControl = list(), obsWeights = NULL, env = parent.frame())
```

Arguments

`Y`	The outcome in the training data set. Must be a numeric vector.
`X`	The predictor variables in the training data set, usually a data.frame.
`newX`	The predictor variables in the validation data set. The structure should match X. If missing, uses X for newX.
`SL.library`	Either a character vector of prediction algorithms or a list containing character vectors. See details below for examples on the structure. A list of functions included in the SuperLearner package can be found with `listWrappers()`.
`verbose`	logical; TRUE for printing progress during the computation (helpful for debugging).
`family`	Currently allows `gaussian` or `binomial` to describe the error distribution. Link function information will be ignored and should be contained in the method argument below.
`method`	A list (or a function to create a list) containing details on estimating the coefficients for the super learner and the model to combine the individual algorithms in the library. See `?method.template` for details. Currently, the built in options are either "method.NNLS" (the default), "method.NNLS2", "method.NNloglik", "method.CC_LS", "method.CC_nloglik", or "method.AUC". NNLS and NNLS2 are non-negative least squares based on the Lawson-Hanson algorithm and the dual method of Goldfarb and Idnani, respectively. NNLS and NNLS2 will work for both gaussian and binomial outcomes. NNloglik is a non-negative binomial likelihood maximization using the BFGS quasi-Newton optimization method. NN* methods are normalized so weights sum to one. CC_LS uses Goldfarb and Idnani's quadratic programming algorithm to calculate the best convex combination of weights to minimize the squared error loss. CC_nloglik calculates the convex combination of weights that minimize the negative binomial log likelihood on the logistic scale using the sequential quadratic programming algorithm. AUC, which only works for binary outcomes, uses the Nelder-Mead method via the optim function to minimize rank loss (equivalent to maximizing AUC).

`id`	Optional cluster identification variable. For the cross-validation splits, `id` forces observations in the same cluster to be in the same validation fold. `id` is passed to the prediction and screening algorithms in SL.library, but be sure to check the individual wrappers as many of them ignore the information.
`obsWeights`	Optional observation weights variable. As with `id` above, `obsWeights` is passed to the prediction and screening algorithms, but many of the built in wrappers ignore (or can't use) the information. If you are using observation weights, make sure the library you specify uses the information.
`control`	A list of parameters to control the estimation process. Parameters include `saveFitLibrary` and `trimLogit`. See `SuperLearner.control` for details.
`cvControl`	A list of parameters to control the cross-validation process. Parameters include `V`, `stratifyCV`, `shuffle` and `validRows`. See `SuperLearner.CV.control` for details.
`env`	Environment containing the learner functions. Defaults to the calling environment.

Details

`SuperLearner` fits the super learner prediction algorithm. The weights for each algorithm in `SL.library` is estimated, along with the fit of each algorithm.

The prescreen algorithms. These algorithms first rank the variables in `X` based on either a univariate regression p-value of the `randomForest` variable importance. A subset of the variables in `X` is selected based on a pre-defined cut-off. With this subset of the X variables, the algorithms in `SL.library` are then fit.

The SuperLearner package contains a few prediction and screening algorithm wrappers. The full list of wrappers can be viewed with `listWrappers()`. The design of the SuperLearner package is such that the user can easily add their own wrappers. We also maintain a website with additional examples of wrapper functions at `https://github.com/ecpolley/SuperLearnerExtra`.

Value

`call`	The matched call.
`libraryNames`	A character vector with the names of the algorithms in the library. The format is 'predictionAlgorithm_screeningAlgorithm' with '_All' used to denote the prediction algorithm run on all variables in X.
`SL.library`	Returns `SL.library` in the same format as the argument with the same name above.
`SL.predict`	The predicted values from the super learner for the rows in `newX`.
`coef`	Coefficients for the super learner.
`library.predict`	A matrix with the predicted values from each algorithm in `SL.library` for the rows in `newX`.
`Z`	The Z matrix (the cross-validated predicted values for each algorithm in `SL.library`).
`cvRisk`	A numeric vector with the V-fold cross-validated risk estimate for each algorithm in `SL.library`. Note that this does not contain the CV risk estimate for the SuperLearner, only the individual algorithms in the library.

`family`	Returns the `family` value from above
`fitLibrary`	A list with the fitted objects for each algorithm in `SL.library` on the full training data set.
`cvFitLibrary`	A list with fitted objects for each algorithm in `SL.library` on each of `V` different training data sets.
`varNames`	A character vector with the names of the variables in `X`.
`validRows`	A list containing the row numbers for the V-fold cross-validation step.
`method`	A list with the method functions.
`whichScreen`	A logical matrix indicating which variables passed each screening algorithm.
`control`	The `control` list.
`cvControl`	The `cvControl` list.
`errorsInCVLibrary`	A logical vector indicating if any algorithms experienced an error within the CV step.
`errorsInLibrary`	A logical vector indicating if any algorithms experienced an error on the full data.
`env`	Environment passed into function which will be searched to find the learner functions. Defaults to the calling environment.
`times`	A list that contains the execution time of the SuperLearner, plus separate times for model fitting and prediction.

Author(s)

Eric C Polley `<epolley@uchicago.edu>`

References

van der Laan, M. J., Polley, E. C. and Hubbard, A. E. (2008) Super Learner, *Statistical Applications of Genetics and Molecular Biology*, **6**, article 25.

Examples

```
## Not run:
## simulate data
set.seed(23432)
## training set
n <- 500
p <- 50
X <- matrix(rnorm(n*p), nrow = n, ncol = p)
colnames(X) <- paste("X", 1:p, sep="")
X <- data.frame(X)
Y <- X[, 1] + sqrt(abs(X[, 2] * X[, 3])) + X[, 2] - X[, 3] + rnorm(n)

## test set
m <- 1000
newX <- matrix(rnorm(m*p), nrow = m, ncol = p)
```

```
colnames(newX) <- paste("X", 1:p, sep="")
newX <- data.frame(newX)
newY <- newX[, 1] + sqrt(abs(newX[, 2] * newX[, 3])) + newX[, 2] -
  newX[, 3] + rnorm(m)

# generate Library and run Super Learner
SL.library <- c("SL.glm", "SL.randomForest", "SL.gam",
  "SL.polymars", "SL.mean")
test <- SuperLearner(Y = Y, X = X, newX = newX, SL.library = SL.library,
  verbose = TRUE, method = "method.NNLS")
test

# library with screening
SL.library <- list(c("SL.glmnet", "All"), c("SL.glm", "screen.randomForest",
  "All", "screen.SIS"), "SL.randomForest", c("SL.polymars", "All"), "SL.mean")
test <- SuperLearner(Y = Y, X = X, newX = newX, SL.library = SL.library,
  verbose = TRUE, method = "method.NNLS")
test

# binary outcome
set.seed(1)
N <- 200
X <- matrix(rnorm(N*10), N, 10)
X <- as.data.frame(X)
Y <- rbinom(N, 1, plogis(.2*X[, 1] + .1*X[, 2] - .2*X[, 3] +
  .1*X[, 3]*X[, 4] - .2*abs(X[, 4])))

SL.library <- c("SL.glmnet", "SL.glm", "SL.knn", "SL.gam", "SL.mean")

# least squares loss function
test.NNLS <- SuperLearner(Y = Y, X = X, SL.library = SL.library,
  verbose = TRUE, method = "method.NNLS", family = binomial())
test.NNLS

# negative log binomial likelihood loss function
test.NNloglik <- SuperLearner(Y = Y, X = X, SL.library = SL.library,
  verbose = TRUE, method = "method.NNloglik", family = binomial())
test.NNloglik

# 1 - AUC loss function
test.AUC <- SuperLearner(Y = Y, X = X, SL.library = SL.library,
  verbose = TRUE, method = "method.AUC", family = binomial())
test.AUC

# 2
# adapted from library(SIS)
set.seed(1)
# training
b <- c(2, 2, 2, -3*sqrt(2))
n <- 150
p <- 200
truerho <- 0.5
corrmat <- diag(rep(1-truerho, p)) + matrix(truerho, p, p)
```

```
corrmat[, 4] = sqrt(truerho)
corrmat[4, ] = sqrt(truerho)
corrmat[4, 4] = 1
cholmat <- chol(corrmat)
x <- matrix(rnorm(n*p, mean=0, sd=1), n, p)
x <- x
feta <- x[, 1:4]
fprob <- exp(feta) / (1 + exp(feta))
y <- rbinom(n, 1, fprob)

# test
m <- 10000
newx <- matrix(rnorm(m*p, mean=0, sd=1), m, p)
newx <- newx
newfeta <- newx[, 1:4]
newfprob <- exp(newfeta) / (1 + exp(newfeta))
newy <- rbinom(m, 1, newfprob)

DATA2 <- data.frame(Y = y, X = x)
newDATA2 <- data.frame(Y = newy, X=newx)

create.SL.knn <- function(k = c(20, 30)) {
  for(mm in seq(length(k))){
    eval(parse(text = paste('SL.knn.', k[mm], '<- function(..., k = ', k[mm],
      ') SL.knn(..., k = k)', sep = '')), envir = .GlobalEnv)
  }
  invisible(TRUE)
}
create.SL.knn(c(20, 30, 40, 50, 60, 70))

# library with screening
SL.library <- list(c("SL.glmnet", "All"), c("SL.glm", "screen.randomForest"),
  "SL.randomForest", "SL.knn", "SL.knn.20", "SL.knn.30", "SL.knn.40",
  "SL.knn.50", "SL.knn.60", "SL.knn.70",
  c("SL.polymars", "screen.randomForest"))
test <- SuperLearner(Y = DATA2$Y, X = DATA2[, -1], newX = newDATA2[, -1],
  SL.library = SL.library, verbose = TRUE, family = binomial())
test

## examples with multicore
set.seed(23432, "L'Ecuyer-CMRG")  # use L'Ecuyer for multicore seeds. see ?set.seed for details
## training set
n <- 500
p <- 50
X <- matrix(rnorm(n*p), nrow = n, ncol = p)
colnames(X) <- paste("X", 1:p, sep="")
X <- data.frame(X)
Y <- X[, 1] + sqrt(abs(X[, 2] * X[, 3])) + X[, 2] - X[, 3] + rnorm(n)

## test set
m <- 1000
newX <- matrix(rnorm(m*p), nrow = m, ncol = p)
colnames(newX) <- paste("X", 1:p, sep="")
```

```
newX <- data.frame(newX)
newY <- newX[, 1] + sqrt(abs(newX[, 2] * newX[, 3])) + newX[, 2] - newX[, 3] + rnorm(m)

# generate Library and run Super Learner
SL.library <- c("SL.glm", "SL.randomForest", "SL.gam",
  "SL.polymars", "SL.mean")

testMC <- mcSuperLearner(Y = Y, X = X, newX = newX, SL.library = SL.library,
  method = "method.NNLS")
testMC

## examples with snow
library(parallel)
cl <- makeCluster(2, type = "PSOCK") # can use different types here
clusterSetRNGStream(cl, iseed = 2343)
# make SL functions available on the clusters, use assignment to avoid printing
foo <- clusterEvalQ(cl, library(SuperLearner))
testSNOW <- snowSuperLearner(cluster = cl, Y = Y, X = X, newX = newX,
  SL.library = SL.library, method = "method.NNLS")
testSNOW
stopCluster(cl)

## snow example with user-generated wrappers
# If you write your own wrappers and are using snowSuperLearner()
# These new wrappers need to be added to the SuperLearner namespace and exported to the clusters
# Using a simple example here, but can define any new SuperLearner wrapper
my.SL.wrapper <- function(...) SL.glm(...)
# assign function into SuperLearner namespace
environment(my.SL.wrapper) <-asNamespace("SuperLearner")

cl <- makeCluster(2, type = "PSOCK") # can use different types here
clusterSetRNGStream(cl, iseed = 2343)
# make SL functions available on the clusters, use assignment to avoid printing
foo <- clusterEvalQ(cl, library(SuperLearner))
clusterExport(cl, c("my.SL.wrapper"))  # copy the function to all clusters
testSNOW <- snowSuperLearner(cluster = cl, Y = Y, X = X, newX = newX,
  SL.library = c("SL.glm", "SL.mean", "my.SL.wrapper"), method = "method.NNLS")
testSNOW
stopCluster(cl)

## timing
replicate(5, system.time(SuperLearner(Y = Y, X = X, newX = newX,
  SL.library = SL.library, method = "method.NNLS")))

replicate(5, system.time(mcSuperLearner(Y = Y, X = X, newX = newX,
  SL.library = SL.library, method = "method.NNLS")))

cl <- makeCluster(2, type = 'PSOCK')
# make SL functions available on the clusters, use assignment to avoid printing
foo <- clusterEvalQ(cl, library(SuperLearner))
replicate(5, system.time(snowSuperLearner(cl, Y = Y, X = X, newX = newX,
  SL.library = SL.library, method = "method.NNLS")))
stopCluster(cl)
```

```
## End(Not run)
```

`SuperLearner.control` *Control parameters for the SuperLearner*

Description

Control parameters for the `SuperLearner`

Usage

```
SuperLearner.control(saveFitLibrary = TRUE, saveCVFitLibrary = FALSE, trimLogit = 0.001)
```

Arguments

`saveFitLibrary`	Logical. Should the fit for each algorithm be saved in the output from `SuperLearner`.
`saveCVFitLibrary`	Logical. Should cross-validated fits for each algorithm be saved in the output from `SuperLearner`.
`trimLogit`	number between 0.0 and 0.5. What level to truncate the logit transformation to maintain a bounded loss function when using the NNloglik method.

Value

A list containing the control parameters.

`SuperLearner.CV.control` *Control parameters for the cross validation steps in* `SuperLearner`

Description

Control parameters for the cross validation steps in `SuperLearner`

Usage

```
SuperLearner.CV.control(V = 10L, stratifyCV = FALSE, shuffle = TRUE,
  validRows = NULL)
```

Arguments

`V`	Integer. Number of splits for the V-fold cross-validation step. The default is 10. In most cases, between 10 and 20 splits works well.
`stratifyCV`	Logical. Should the data splits be stratified by a binary response? Attempts to maintain the same ratio in each training and validation sample.
`shuffle`	Logical. Should the rows of `X` be shuffled before creating the splits.
`validRows`	A List. Use this to pass pre-specified rows for the sample splits. The length of the list should be `V` and each entry in the list should contain a vector with the row numbers of the corresponding validation sample.

Value

A list containing the control parameters

`SuperLearnerNews` *Show the NEWS file for the SuperLearner package*

Description

Show the NEWS file of the SuperLearner package. The function is simply a wrapper for the `RShowDoc` function

Usage

```
SuperLearnerNews(...)
SuperLearnerDocs(what = 'SuperLearnerR.pdf', ...)
```

Arguments

`...`	additional arguments passed to `RShowDoc`
`what`	specify what document to open. Currently supports the NEWS file and the PDF files 'SuperLearner.pdf' and 'SuperLearnerR.pdf'.

Value

A invisible character string given the path to the SuperLearner NEWS file

---

`trimLogit` *truncated-probabilities logit transformation*

Description

computes the logit transformation on the truncated probabilities

Usage

```
trimLogit(x, trim = 1e-05)
```

Arguments

`x`	vector of probabilities.
`trim`	value to truncate probabilities at. Currently symmetric truncation (trim and 1-trim).

Value

logit transformed values

Examples

```
x <- c(0.00000001, 0.0001, 0.001, 0.01, 0.1, 0.3, 0.7, 0.9, 0.99,
  0.999, 0.9999, 0.99999999)
trimLogit(x, trim = 0.001)
data.frame(Prob = x, Logit = qlogis(x), trimLogit = trimLogit(x, 0.001))
```

`write.method.template` *Method to estimate the coefficients for the super learner*

Description

These functions contain the information on the loss function and the model to combine algorithms

Usage

```
write.method.template(file = "", ...)

## a few built in options:
method.NNLS()
method.NNLS2()
method.NNloglik()
method.CC_LS()
method.CC_nloglik()
method.AUC(nlopt_method=NULL, optim_method="L-BFGS-B", bounds=c(0, Inf), normalize=TRUE)
```

Arguments

`file`	A connection, or a character string naming a file to print to. Passed to `cat`.
`optim_method`	Passed to the `optim` call method. See `optim` for details.
`nlopt_method`	Either `optim_method` or `nlopt_method` must be provided, the other must be `NULL`
`bounds`	Bounds for parameter estimates
`normalize`	Logical. Should the parameters be normalized to sum up to 1
`...`	Additional arguments passed to `cat`.

Details

A `SuperLearner` method must be a list (or a function to create a list) with exactly 3 elements. The 3 elements must be named `require`, `computeCoef` and `computePred`.

Value

A list containing 3 elements:

`require`	A character vector listing any required packages. Use `NULL` if no additional packages are required
`computeCoef`	A function. The arguments are: `Z`, `Y`, `libraryNames`, `obsWeights`, `control`, `verbose`. The value is a list with two items: `cvRisk` and `coef`. This function computes the coefficients of the super learner. As the super learner minimizes the cross-validated risk, the loss function information is contained in this function as well as the model to combine the algorithms in `SL.library`.
`computePred`	A function. The arguments are: `predY`, `coef`, `control`. The value is a numeric vector with the super learner predicted values.

Author(s)

Eric C Polley `<Polley.Eric@mayo.edu>`

See Also

`SuperLearner`

Examples

```
write.method.template(file = '')
```

`write.screen.template` *screening algorithms for SuperLearner*

Description

Screening algorithms for `SuperLearner` to be used with `SL.library`.

Usage

```
write.screen.template(file = "", ...)
```

Arguments

`file`	A connection, or a character string naming a file to print to. Passed to `cat`.
`...`	Additional arguments passed to `cat`

Details

Explain structure of a screening algorithm here:

Value

`whichVariable`	A logical vector with the length equal to the number of columns in `X`. TRUE indicates the variable (column of X) should be included.

Author(s)

Eric C Polley `<polley.eric@mayo.edu>`

See Also

`SuperLearner`

Examples

```
write.screen.template(file = '')
```

`write.SL.template` *Wrapper functions for prediction algorithms in SuperLearner*

Description

Template function for SuperLearner prediction wrappers and built in options.

Usage

```
write.SL.template(file = "", ...)
```

Arguments

`file`	A connection, or a character string naming a file to print to. Passed to `cat`.
`...`	Additional arguments passed to `cat`

Details

Describe SL.* structure here

Value

A list with two elements:

`pred`	The predicted values for the rows in `newX`.
`fit`	A list. Contains all objects necessary to get predictions for new observations from specific algorithm.

Author(s)

Eric C Polley <`epolley@uchicago.edu`>

See Also

`SuperLearner`

Examples

```
write.SL.template(file = '')
```

Index