Package: nadir (via r-universe)

October 13, 2026

Type Package

Title Super Learning with Flexible Formulas

Version 1.0.0

Description A functional programming based implementation of the super learner algorithm with an emphasis on supporting the use of formulas to specify learners. This approach offers several improvements compared to past implementations including the ability to easily use random-effects specified in formulas (like y ~ (age | strata) + ...) and construction of new learners is as simple as writing and passing a new function. The super learner algorithm was originally described in van der Laan et al. (2007) <https://biostats.bepress.com/ucbbiostat/paper222/>.

License MIT + file LICENSE

Depends R (>= 4.2.0)

Encoding UTF-8

LazyData true

Roxygen list (markdown = TRUE, roclets = c (``namespace", ``rd", ``srr::srr_stats_roclet"))

Suggests MASS, VGAM, dbarts, e1071, earth, gbm, ggplot2, glmnet, hal9001, kernlab, kknn, knitr, lightgbm, lme4, mgcv, nnet, palmerpenguins, randomForest, ranger, rmarkdown, rpart, srr, survival, testthat (>= 3.0.0), withr, xgboost

VignetteBuilder knitr

Config/testthat/edition 3

URL https://ctesta01.github.io/nadir/, https://github.com/ctesta01/nadir/

BugReports https://github.com/ctesta01/nadir/issues

Imports dplyr, future, future.apply, lifecycle, methods, nnls, origami, tibble, tidyr

Config/roxygen2/version 8.1.0

Config/pak/sysreqs libicu-dev

Repository https://ctesta01.r-universe.dev

Date/Publication 2026-10-13 21:24:49 UTC

RemoteUrl https://github.com/ctesta01/nadir

RemoteRef HEAD

RemoteSha d2eef7b6ba6095107ac848842480acbd4ff37b45

Contents

add_screener . . . 3
add_stratification . . . 5
as_multi_predictor . . . 7
binary_learners . . . 8
compare_learners . . . 9
crossfit_sl_methods . . . 10
crossfit_super_learner . . . 12
cv_character_and_factors_schema . . . 16
cv_origami_schema . . . 18
cv_random_schema . . . 19
cv_super_learner . . . 20
density_learners . . . 23
determine_super_learner_weights_nnls . . . 24
determine_weights_for_binary_outcomes . . . 25
determine_weights_using_neg_log_loss . . . 26
df_to_survival_stacked . . . 27
learners . . . 28
list_known_learners . . . 30
lnr_bart . . . 30
lnr_cvglmnet . . . 31
lnr_earth . . . 32
lnr_gam . . . 33
lnr_gausspr . . . 34
lnr_gbm . . . 35
lnr_glm . . . 36
lnr_glm_density . . . 36
lnr_glmer . . . 37
lnr_glmnet . . . 38
lnr_glmnet_grid . . . 39
lnr_hal . . . 40
lnr_hal_grid . . . 41
lnr_heteroskedastic_density . . . 42
lnr_homoskedastic_density . . . 43
lnr_knn . . . 45
lnr_knn_binary . . . 46
lnr_lightgbm . . . 47

lnr_lm 48
lnr_lm_density 49
lnr_lmer 50
lnr_logistic 50
lnr_mean 51
lnr_multinomial_nnet 52
lnr_multinomial_ranger 52
lnr_multinomial_vglm 53
lnr_nnet 54
lnr_ppr 55
lnr_ranger 56
lnr_ranger_binary 56
lnr_rf 57
lnr_rf_binary 58
lnr_rpart 59
lnr_rpart_binary 60
lnr_svm 61
lnr_svm_binary 62
lnr_xgboost 63
make_learner_names_unique 64
multiclass_learners 65
nadir_sl_model_methods 66
nadir_supported_types 68
negative_log_loss 69
negative_log_loss_for_binary 69
plot.nadir_cv_sl 70
plot.nadir_sl_model 71
screener_cor 73
screener_cor_top_n 74
screener_t_test 75
screeners 76
super_learner 77
truncate_lnr 82

Index **83**

`add_screener` *Add a Screener to a Learner*

Description

Add a Screener to a Learner

Usage

```
add_screener(learner, screener, screener_extra_args = NULL)
```

Arguments

learner	A learner to be modified by wrapping a screening stage on top of it.
screener	A screener to be added on top of the learner
screener_extra_args	Extra arguments to be passed to the screener

Value

A modified learner that when called on data and a formula now runs a screening stage before fitting the learner and returning a prediction function.

Examples

```
# construct a learner where variables with less than .6 correlation are screened out
lnr_glm_with_cor_60_thresholding <-
  add_screener(
    learner = lnr_glm,
    screener = screener_cor,
    screener_extra_args = list(threshold = 0.6)
  )

# train that on the mtcars dataset - also checking that extra arguments are
# properly passed to glm
lnr_glm_with_cor_60_thresholding(mtcars, formula = mpg ~ ., family = "gaussian")(mtcars)

# if we've screened out variables with low correlation to mpg, one such variable is qsec,
# so changing qsec shouldn't modify the predictions from our learned algorithm
mtcars_but_qsec_is_changed <- mtcars
mtcars_but_qsec_is_changed$qsec <- rnorm(n = nrow(mtcars))

identical(
  lnr_glm_with_cor_60_thresholding(mtcars, formula = mpg ~ .)(mtcars),
  lnr_glm_with_cor_60_thresholding(mtcars, formula = mpg ~ .)(mtcars_but_qsec_is_changed)
)

# earth version
lnr_earth_with_cor_60_thresholding <-
  add_screener(
    learner = lnr_earth,
    screener = screener_cor,
    screener_extra_args = list(threshold = 0.6)
  )
lnr_earth_with_cor_60_thresholding(mtcars, formula = mpg ~ .)(mtcars)

identical(
  lnr_earth_with_cor_60_thresholding(mtcars, formula = mpg ~ .)(mtcars),
  lnr_earth_with_cor_60_thresholding(mtcars, formula = mpg ~ .)(mtcars)
)

# note that this 'test' does not pass for a learner like randomForest that has
```

```
# some randomness in its predictions.
```

`add_stratification` *Stratify a Learner by One or More Categorical Variables*

Description

`add_stratification()` takes a learner and returns a new learner that, when trained, fits the original learner separately within each stratum of `stratify_by` and uses the stratum-specific fits for prediction. This is useful for enriching a `super_learner()` library with learners that do not smooth over pre-specified subgroups, e.g. when estimating nuisance regressions for subgroup-specific (heterogeneous treatment effect) parameters.

Usage

```
add_stratification(
  learner,
  stratify_by,
  min_stratum_size = 10,
  pooled_fallback = TRUE
)
```

Arguments

`learner` A learner (see `?learners`) to be fit separately within each stratum.

`stratify_by` A character vector of one or more column names to stratify by. With more than one name, strata are the observed combinations of the variables (their interaction). Columns must be factor, character, logical, or integer-valued numeric; non-integer numeric columns error with a suggestion to discretize first.

`min_stratum_size` Strata with fewer than this many observations in the training data are not given their own fit; they use the pooled fallback (or error if `pooled_fallback = FALSE`). Defaults to 10.

`pooled_fallback` If `TRUE` (default), additionally fit the learner on the full training data and use that pooled fit for too-small strata, strata whose fits error, and strata unseen at training time. If `FALSE`, each of those situations errors.

Details

Formulas must not mention the stratifying variable(s). Within a stratum, the stratifying variable is constant, so including it as a predictor is at best degenerate and at worst an error for many learners. Rather than silently rewriting the user's formula, the stratified learner errors at training time if the formula mentions any of `stratify_by` anywhere in its right-hand side (including inside transformations like `poly()`, interactions, and `(1 | x)` terms). Note that `y ~ .` therefore errors,

because its expansion includes the stratifying variable, which must be a column of the data; write `y ~ . - stratifying_var` or enumerate the predictors instead.

Pooled fallback. By default (`pooled_fallback = TRUE`), the original learner is also fit once on the full (unstratified) training data. This pooled fit is used:

- at training time, for strata with fewer than `min_stratum_size` observations, and for strata in which the stratum-specific fit errors; and
- at prediction time, for rows of `newdata` whose stratum was not present in the training data (which can happen for rare strata under cross-validation).

Each fallback signals a warning naming the affected strata; inside `super_learner()` these warnings are captured (not printed) and reported in the `$warnings_from_*` fields of the output. With `pooled_fallback = FALSE`, all of the above situations are errors instead, and inside `super_learner()` an erring learner is dropped from the ensemble by the usual machinery.

Cross-validation. When using stratified learners inside `super_learner()`, we strongly recommend passing the stratifying variable as `strata_ids` so that every cross-validation training fold contains observations from every stratum; e.g. `super_learner(..., strata_ids = data$categorical_var)`. Note that `strata_ids` (which balances CV folds) and `stratify_by` (which stratifies a learner's fits) are related but distinct.

Supported outcome types. Currently only `'continuous'` and `'binary'` learners are supported (prediction functions returning one numeric value per row of `newdata`). Density and multiclass learners are not yet supported by `add_stratification()`.

Value

A new learner: a function taking `(data, formula, weights = NULL, ...)` and returning a prediction function of `newdata`. Predictions are returned in the row order of `newdata`. The new learner carries `sl_lnr_name` (e.g. `"lm_stratified_by_cyl"`) and `sl_lnr_type` attributes, and inherits the wrapped learner's `outcome_type_dependent_args`.

See Also

learners add_screener super_learner

Examples

```
# a glm fit separately within each stratum of cyl:
lnr_lm_by_cyl <- add_stratification(lnr_lm,
  stratify_by = "cyl",
  min_stratum_size = 5
)
trained <- lnr_lm_by_cyl(mtcars, mpg ~ hp + wt)
trained(mtcars)

# note that formulas mentioning the stratifying variable error, including
# mpg ~ . (because its expansion includes cyl); instead write:
trained2 <- lnr_lm_by_cyl(mtcars, mpg ~ . - cyl)

# inside super_learner(), pass the stratifying variable as strata_ids so
# every training fold contains every stratum:
```

```
sl <- super_learner(
  data = mtcars,
  formulas = c(.default = mpg ~ hp + wt, lm_stratified_by_cyl = mpg ~ hp + wt),
  learners = list(lnr_lm, lnr_lm_by_cyl),
  strata_ids = mtcars$cyl,
  n_folds = 3
)
```

`as_multi_predictor` *Construct a Multi-Predictor*

Description

A multi-predictor is a named list of prediction functions produced by a single call to a learner, e.g. one prediction function per lambda value along a jointly-fit regularization path. `super_learner()` expands multi-predictors into distinct pseudo-learners (one per element) whose names are formed as `paste(learner_name, names(predictors), sep = '_')`.

Usage

```
as_multi_predictor(predictors)
```

Arguments

`predictors`	A uniquely-named list of functions, each of which accepts `newdata` and returns a numeric vector of predictions with one entry per row of `newdata`.

Details

Learner authors may wrap any algorithm that fits many models in one pass (e.g., a lasso path, a boosting run saved at several iteration counts, etc.) with this to expose each sub-model along the path to the meta-learning stage of `super_learner()` at the cost of a single fit.

The sub-model names must be identical across calls on different training folds (i.e., deterministic given the learner arguments, not data-dependent), since `super_learner()` aligns pseudo-learners across folds by name.

Value

The same list, classed as `nadir_multi_predictor`.

See Also

lnr_glmnet_grid lnr_hal_grid

Examples

```
# a custom grid learner returns one fitted predictor per tuning value;
# as_multi_predictor() marks the named collection so super_learner()
# expands it into one candidate learner per sub-model
mp <- as_multi_predictor(list(
  lambda_0.1 = function(newdata) rep(1, nrow(newdata)),
  lambda_1   = function(newdata) rep(2, nrow(newdata))
))
inherits(mp, "nadir_multi_predictor")
names(mp)
mp$lambda_1(mtcars[1:3, ])
```

---

binary_learners *Binary Learners in* {nadir}

Description

- `lnr_nnet`
- `lnr_rf_binary`
- `lnr_logistic`

Details

The important thing to know about binary learners is that they need to produce predictions that the outcome is `== 1` or `TRUE`.

Also, for binary outcomes, we should make sure to use the `determine_weights_for_binary_outcomes` in our calls to `super_learner()` which calculates the estimated probability of the observed outcome (either 0 or 1) and then applies the negative log loss function afterwards. This can be done automatically by declaring `outcome_type = 'binary'` in calling `super_learner()`

Value

Every learner function shares the same contract: called with `(data, formula, ...)` it fits the underlying model and returns a *prediction closure* which is a function of `newdata` returning a numeric vector of predictions (predicted probabilities of the second factor level for binary learners; predicted densities for density learners; a matrix of class probabilities for multiclass learners).

See Also

density_learners learners

Examples

```
super_learner(
  data = mtcars,
  learners = list(logistic1 = lnr_logistic, logistic2 = lnr_logistic, lnr_rf_binary),
  formulas = list(
    .default = am ~ .,
    logistic2 = am ~ mpg * hp + .
  ),
  outcome_type = "binary"
)
```

---

compare_learners *Compare Learners*

Description

Compare learners using the specified `loss_metric`

Usage

```
compare_learners(sl_output, y_variable = NULL, loss_metric)
```

Arguments

`sl_output`	Output from running `super_learner()` with `verbose_output = TRUE`.
`y_variable`	A character vector indicating the outcome variable. `y_variable` will be automatically inferred if it is missing and can be inferred from the `sl_output`.
`loss_metric`	A loss metric, like the mean-squared-error or negative-log-loss to be used in comparing the learners. A loss metric should take two (vector) arguments: predictions, and true outcomes, and produce a single statistic summarizing the performance of each learner.

Value

A data.frame with the loss-metric on the held-out data for each learner.

Examples

```
sl_model <- super_learner(
  data = mtcars,
  learners = list(lm = lnr_lm, rf = lnr_rf, mean = lnr_mean),
  formula = mpg ~ .
)

compare_learners(sl_model)

sl_model <- super_learner(
```

```
  data = mtcars,
  learners = list(lnr_logistic, lnr_rf_binary, mean = lnr_mean),
  formula = am ~ mpg,
  outcome_type = "binary"
)
compare_learners(sl_model)
```

`crossfit_sl_methods` *Predicting from a Cross-Fitted Super Learner*

Description

A cross-fitted super learner deliberately has no single prediction function: it is one fitted super learner *per outer fold*, retained so that each observation can be predicted by an ensemble trained without it. This method therefore errors with directions to the fold-aware interfaces: `$oof_predictions` for out-of-fold predictions, `$predict_modified(modify)` for interventional/modified-data predictions, and `$predict_fold(newdata_list)` for arbitrary per-fold newdata. To fit one predictor on all the data, use `super_learner()`.

Following the convention that a super learner's "coefficients" are its meta-learned ensemble weights (see `coef.nadir_sl_model`), the coefficients of a *cross-fitted* super learner are the ensemble weights of each outer fold's fit, returned as a folds-by-learners matrix. Entries are `NA` for learners dropped (due to fitting errors) in a given fold. Column-wise variability across rows is a useful diagnostic of how stable the ensemble is across folds; see `summary.nadir_crossfit_sl` and `plot(x, type = "weights")`.

Equivalent to `object$oof_predictions`: each value is the prediction for that observation from the outer fold whose training data excluded it. Values are in the *original row order* of the (complete-case-filtered) data; `object$complete_rows` maps positions back to the data as supplied. Rows never held out by the `cv_schema` are `NA`.

Observed outcomes minus out-of-fold predictions, in the original row order of the (complete-case-filtered) data. `NA` for rows never held out by the `cv_schema`. Errors for density and multiclass outcomes, where out-of-fold predictions are densities/probabilities of the observed outcome rather than point predictions.

All outer folds share one specification by construction, so this delegates to the first fold's fit: a single `formula` when all learners share one, otherwise a named list of per-learner formulas.

The number of rows of the (complete-case-filtered) data over which cross-fitting was performed — i.e., the length of `$oof_predictions` and `$fold_assignments`.

Reports (i) the held-out loss of each outer fold alongside the overall cross-fitted loss, and (ii) a weight-stability table: the mean, standard deviation, and range of each learner's ensemble weight across the outer folds. Large across-fold variability in a learner's weight — or `n_folds_present` below `n_folds`, indicating the learner errored in some folds — is a signal that the ensemble is not stable under resampling.

Two plot types are provided:

`type = "weights" (default)` Ensemble-weight stability: each learner's weight in each outer fold (open circles) with the across-fold mean (filled points). Tight clusters indicate an ensemble that is stable under resampling; missing circles indicate folds where a learner errored and was dropped.

`type = "fitted"` Out-of-fold predictions against observed outcomes, colored by outer fold, with the identity line for reference. Every prediction comes from an ensemble trained entirely without that observation. Rows never held out by the `cv_schema` are omitted. Not defined for `outcome_type = "density"` or `"multiclass"`.

Requires the **ggplot2** package (listed in `Suggests`).

Usage

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

## S3 method for class 'nadir_crossfit_sl'
coef(object, ...)

## S3 method for class 'nadir_crossfit_sl'
fitted(object, ...)

## S3 method for class 'nadir_crossfit_sl'
residuals(object, ...)

## S3 method for class 'nadir_crossfit_sl'
formula(x, ...)

## S3 method for class 'nadir_crossfit_sl'
nobs(object, ...)

## S3 method for class 'nadir_crossfit_sl'
summary(object, ...)

## S3 method for class 'summary.nadir_crossfit_sl'
print(x, digits = 4, ...)

## S3 method for class 'nadir_crossfit_sl'
plot(x, type = c("weights", "fitted"), ...)
```

Arguments

`object`	A `nadir_crossfit_sl`.
`...`	Ignored; included for compatibility with the generic.
`x`	A `nadir_crossfit_sl` as returned by `crossfit_super_learner()`.
`digits`	the number of digits to print in the summary
`type`	One of `"fitted"` or `"weights"`.

Value

Does not return; always signals an informative error.

A numeric matrix (rows: outer folds; columns: learners); each row sums to 1 over its non-NA entries.

A numeric vector of length `nobs(object)`.

A numeric vector of length `nobs(object)`.

A `formula` or a named list of formulas.

An integer.

An object of class `summary.nadir_crossfit_sl`: a list with `$fold_losses` (data.frame: fold, n_validation, loss), `$weight_stability` (data.frame: learner, mean_weight, sd_weight, min_weight, max_weight, n_folds_present, ordered by mean weight), and the scalars `$cv_loss`, `$y_variable`, `$outcome_type`, `$n_folds`, `$inner_n_folds`, `$n_obs`, `$n_never_held_out`.

A `ggplot` object.

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  cf <- crossfit_super_learner(
    data = mtcars,
    formula = mpg ~ cyl + hp,
    n_folds = 2, inner_n_folds = 2,
    learners = list(mean = lnr_mean, lm = lnr_lm)
  )
  plot(cf) # out-of-fold predictions vs. observed
  plot(cf, type = "weights") # weight stability across folds
}
```

---

crossfit_super_learner

Cross-Fit a Super Learner and Retain Fold-Specific Predictors

---

Description

Cross-fitting proceeds by:

1. splitting the data into `n_folds` outer folds,
2. fitting a full `super_learner()` (with its own internal `inner_n_folds`-fold cross-validation to determine ensemble weights) on each outer training split,
3. predicting on the corresponding outer held-out fold,
4. retaining each fold-specific prediction function for later modified-data (e.g. intervention) prediction.

Usage

```
crossfit_super_learner(
  data,
  learners,
  formulas,
  y_variable = NULL,
  n_folds = 5,
  inner_n_folds = 5,
  determine_super_learner_weights = NULL,
  ensemble_or_discrete = c("ensemble", "discrete"),
  cv_schema = NULL,
  inner_cv_schema = NULL,
  outcome_type = c("continuous", "binary", "density", "multiclass"),
  extra_learner_args = NULL,
  cluster_ids = NULL,
  strata_ids = NULL,
  weights = NULL,
  rowids = NULL,
  loss_metric = NULL,
  use_complete_cases = FALSE
)
```

Arguments

`data` Data to use in training a `super_learner`.

`learners` A list of predictor/closure-returning-functions. See Details.

`formulas` Either a single regression formula or a vector of regression formulas.

`y_variable` Optional character name of the outcome variable; inferred from `formulas` when omitted.

`n_folds` Number of *outer* cross-fitting folds.

`inner_n_folds` Number of folds used by the inner `super_learner()` on each outer training split to estimate ensemble weights. Deliberately decoupled from `n_folds`.

`determine_super_learner_weights` Function used by the inner `super_learner()` calls to determine ensemble weights. If `NULL` (default), inferred from `outcome_type` via `default_determine_weights()`, matching `super_learner()`'s own inference.

`ensemble_or_discrete` Defaults to `'ensemble'`, but can be set to `'discrete'`. Discrete `super_learner()` chooses only one of the candidate learners to have weight 1 in the resulting prediction algorithm, while `ensemble` `super_learner()` combines predictions from 1 or more candidate learners, with respective weights adding up to 1.

`cv_schema` Function taking `(data, n_folds)` and returning a list with `training_data` and `validation_data`, each of length `n_folds`, used for the *outer* split. If omitted and `cluster_ids` or `strata_ids` are supplied, `cv_origami_schema` is used so the outer split respects them; otherwise `cv_random_schema`.

`inner_cv_schema`
: Optional `cv_schema` for the inner `super_learner()` calls. If `NULL` (default), `super_learner()`'s own defaults apply (which respect `cluster_ids`/`strata_ids`, subset to each outer training fold).

`outcome_type`
: One of 'continuous', 'binary', 'multiclass', or 'density'. `outcome_type` is used to infer the correct `determine_super_learner_weights` function if it is not explicitly passed.

`extra_learner_args`
: A list of equal length to the `learners` with additional arguments to pass to each of the specified learners.

`cluster_ids`
: (default: null) If specified, clusters will either be entirely assigned to training or validation (not both) in each cross-validation split.

`strata_ids`
: (default: null) If specified, strata are balanced across training and validation splits so that strata appear in both the training and validation splits.

`weights`
: If specified, (per observation) weights are used to indicate that risk minimization across models (i.e., the meta-learning step) should be targeted to higher weight observations.

`rowids`
: (default: null) If specified, rowids are stored for use in out-of-fold prediction (`$oof_predict()`) and can also be used to validate/observe the behavior of the cross-validation folds construction.

`loss_metric`
: Optional loss function used only to report `cv_loss`, the empirical loss of the cross-fitted out-of-fold predictions. If `NULL` (default), inferred from `outcome_type` via `default_loss_metric()`.

`use_complete_cases`
: (default: FALSE) If the `data` passed have any NA or NaN missing data, restrict the `data` to `data[complete.cases(data),]`.

Details

Each observation's out-of-fold prediction therefore comes from an ensemble whose base learners *and* ensemble weights were estimated entirely without that observation, which is the property required of cross-fitted nuisance estimators in semiparametric workflows (AIPW, TMLE, DML). Typical use is to obtain out-of-fold fits of an outcome model m, its interventional counterparts m_1, m_0 (via `$predict_modified()`), and a propensity model g, each from one `crossfit_super_learner()` call per nuisance.

Value

An object of class `"nadir_crossfit_sl"`, a list with:

`$oof_predict(newdata, rowids = NULL)` A prediction function that takes newdata with the same columns and rows as the training dataset and optionally rowids. If rowids are specified, the rows can be out of order compared to the original training data, but if rowids are not passed, then the ordering is assumed to be the same as the training `data` passed.

`$oof_predictions` Numeric vector of out-of-fold predictions in the original row order of `data` (after any complete-case filtering; see `$complete_rows`). Rows never held out by `cv_schema` are `NA`.

`$oof_predict_modified(modify)` Full-length out-of-fold predictions after applying `modify(newdata)` to each held-out fold, e.g. `modify = function(d) { d$treatment <- 1; d }` to obtain cross-fitted m_1.

`$oof_predict_fold(newdata_list = NULL, modify = NULL)` List of per-fold prediction vectors; entry `i` is produced by the Super Learner trained without outer fold `i`.

`$fold_assignments` Integer vector giving each row's outer validation fold (`NA` if never held out).

`$fold_rows` List of row indices (into the analysis data) comprising each validation fold.

`$sl_fits` List of the `n_folds` fitted `super_learner()` objects, e.g. for inspecting per-fold ensemble weights.

`$cv_loss` Empirical loss of the cross-fitted predictions on held-out rows, or `NA` if it could not be computed.

`$complete_rows` Indices into the originally supplied `data` retained after complete-case filtering (identity when no filtering occurred).

`$warnings_from_fold_predictions` Present only if non-empty: warnings captured while predicting on the outer held-out folds.

`$warnings_from_inner_super_learners` Present only if non-empty: a per-fold list of the warning fields captured by each inner `super_learner()` (see `?super_learner`).

`$warning_learners` Present only if non-empty: names of learners that signaled warnings in any fold.

plus `y_variable`, `outcome_type`, `n_folds`, `inner_n_folds`, `training_data`, and `validation_data`.

Parallelism

Make sure to load the `future` package first using `library(future)`.

Fitting is parallelized over outer folds through `future.apply::future_lapply()`. With `plan(multisession)` only the outer folds run in parallel and each inner `super_learner()` runs sequentially inside its worker. To also parallelize within each outer fold, declare a nested topology, e.g. `plan(list(tweak(multisession, workers = 2),tweak(multisession, workers = I(4))))`; the product across levels should not exceed `future::availableCores()`.

See Also

super_learner cv_super_learner cv_origami_schema

Examples

```
cf <- crossfit_super_learner(
  data = mtcars,
  formulas = mpg ~ disp + hp + am,
  learners = list(mean = lnr_mean, lm = lnr_lm, rf = lnr_rf),
  n_folds = 5,
  rowids = seq_len(nrow(mtcars))
)

# basic usage
```

```
coef(cf)
head(fitted(cf))
head(residuals(cf))
formula(cf)
nobs(cf)
summary(cf)

# but what cf is really for is out of fold predictions:
cf$oof_predict(mtcars, rowids = seq_len(nrow(mtcars)))

cf$oof_predictions # cross-fitted \hat m(X_i) prediction models

m1 <- cf$oof_predict_modified(function(d) {
  d$am <- 1
  d
}) # \hat m(1, X_i)
m0 <- cf$oof_predict_modified(function(d) {
  d$am <- 0
  d
}) # \hat m(0, X_i)

cf$cv_loss # honest empirical loss
lapply(cf$sl_fits, `[[`, "learner_weights") # per-fold ensemble weights
```

`cv_character_and_factors_schema` *Cross Validation Training/Validation Splits with Characters/Factor Columns*

Description

Designed to handle cross-validation on models like randomForest, ranger, glmnet, etc., where the model matrix of newdata must match eactly the model matrix of the training dataset, this function intends to answer the need "The training datasets need to have every level of every discrete-type column that appears in the data."

Usage

```
cv_character_and_factors_schema(
  data,
  n_folds = 5,
  cv_sl_mode = TRUE,
  check_validation_datasets_too = TRUE
)
```

Arguments

`data` Data to use in training a `super_learner`.

`n_folds` The number of cross-validation folds to use in constructing the `super_learner`. Must be a scalar integer of value >= 2.

`cv_sl_mode` A binary (default: TRUE) indicator for if the output training/validation data lists will be used inside another `super_learner` call. If so, then the training data needs to have every level appear at least twice so that the data can be put into further training/validation splits.

`check_validation_datasets_too` Enforce that the validation datasets produced also have every level of every character / factor type column present. This is particularly useful for learners like `glmnet` which require that the `newx` have the exact same shape/structure as the training data, down to binary indicators for every level that appears.

Details

The fundamental idea is to check if the unique levels of character and/or factor columns are represented in every training dataset.

Above and beyond this, this function is designed to support cv_super_learner, which inherently involves two layers of cross-validation. As a result, more stringent conditions are specified when the `cv_sl_mode` is enabled. For convenience this mode is enabled by default

Where factor levels appear in the data, no information about if they are ordered factor columns is used in the CV splits generated.

Value

A list of two lists (`$training_data` and `$validation_data`) which are each lists of length `n_folds`. In each of those entries is a data.frame that contains the nth training or validation fold of the data.

a named list of two lists, each being a list of `n_folds` data frames.

Examples

```
if (requireNamespace("palmerpenguins", quietly = TRUE)) {
  training_validation_splits <- cv_character_and_factors_schema(
    palmerpenguins::penguins
  )

  # we can see the population breakdown across all the training
  # splits:
  sapply(training_validation_splits$training_data, function(df) {
    table(df$species)
  })
  # notably, none of them are empty! this is crucial for certain
  # types of learning algorithms that must see all levels appear in the
  # training data, like random forests.

  # certain models like glmnet require that the prediction dataset
  # newx have the _exact_ same shape as the training data, so it
```

```
  # can be important that every level appears in the validation data
  # as well.  check that by looking into these types of tables:
  sapply(training_validation_splits$validation_data, function(df) {
    table(df$species)
  })

  # if you don't need this level of stringency, but you just want
  # to make cv_splits where every level appears in the training_data,
  # you can do so using the check_validation_datasets_too = FALSE
  # argument.
  penguins_small <- palmerpenguins::penguins[c(1:3, 154:156, 277:279), ]
  penguins_small <- penguins_small[complete.cases(penguins_small), ]

  training_validation_splits <- cv_character_and_factors_schema(
    penguins_small,
    cv_sl_mode = FALSE,
    n_folds = 5,
    check_validation_datasets_too = FALSE
  )

  sapply(training_validation_splits$training_data, function(df) {
    table(df$species)
  })

  # now you can see plenty of non-appearing levels in the validation data:
  sapply(training_validation_splits$validation_data, function(df) {
    table(df$species)
  })
}
```

`cv_origami_schema` *Cross-Validation with Origami*

Description

Cross-Validation with Origami

Usage

```
cv_origami_schema(
  data = data,
  n_folds = 5,
  fold_fun = origami::folds_vfold,
  cluster_ids = NULL,
  strata_ids = NULL,
  ...
)
```

Arguments

data	a data.frame (or similar) to split into training and validation datasets.
n_folds	The number of training_data and validation_data data frames to make.
fold_fun	An origami::folds_* function
cluster_ids	A vector of cluster ids. Clusters are treated as a unit – that is, all observations within a cluster are placed in either the training or validation set. See ?origami::make_folds.
strata_ids	A vector of strata ids. Strata are balanced: insofar as possible the distribution in the sample should be the same as the distribution in the training and validation sets. See ?origami::make_folds.
...	Extra arguments to be passed to origami::make_folds()

Value

A list of n_folds training_data and validation_data data.frames

Examples

```
# to use origami::folds_vfold behind the scenes, just tell nadir::super_learner
# you want to use cv_origami_schema.

sl_model <- super_learner(
  data = mtcars,
  formula = mpg ~ cyl + hp,
  learners = list(rf = lnr_rf, lm = lnr_lm, mean = lnr_mean),
  cv_schema = cv_origami_schema
)

# if you want to use a different origami::folds_* function, pass it into
# cv_origami_schema
sl_model <- super_learner(
  data = mtcars,
  formula = mpg ~ cyl + hp,
  learners = list(rf = lnr_rf, lm = lnr_lm, mean = lnr_mean),
  cv_schema = \(data, n_folds) {
    cv_origami_schema(data, n_folds, fold_fun = origami::folds_loo)
  }
)
```

---

cv_random_schema — *Assign Data to One of n_folds Randomly and Produce Training/Validation Data Lists*

Description

Each row in the data are assigned to one of `1:n_folds` at random. Then for each of `i` in `1:n_folds`, the `training_data[[i]]` is comprised of the data with `sl_fold != i`, i.e., capturing roughly `(n-folds-1)/n_folds` proportion of the data. The validation data is a list of dataframes, each comprising of roughly `1/n_folds` proportion of the data.

Usage

```
cv_random_schema(data, n_folds = 5)
```

Arguments

`data`	a data.frame (or similar) to split into training and validation datasets.
`n_folds`	The number of `training_data` and `validation_data` data frames to make.

Details

Since the assignment to folds is random, the proportions are not exact or guaranteed and there is some variability in the size of each `training_data` data frame, and likewise for the `validation_data` data frames.

Value

A list of two lists (`$training_data` and `$validation_data`) which are each lists of length `n_folds`. In each of those entries is a data.frame that contains the nth training or validation fold of the data.

Examples

```
data(Boston, package = "MASS")
training_validation_data <- cv_random_schema(Boston, n_folds = 3)
# take a look at what's in the output:
str(training_validation_data, max.level = 2)
```

`cv_super_learner` *Cross-Validating a* `super_learner`

Description

Produce cv-rmse for a `super_learner` specified by a closure that accepts data and returns a `super_learner` prediction function.

Usage

```
cv_super_learner(
  data,
  learners,
  formulas,
  y_variable = NULL,
  n_folds = 5,
  determine_super_learner_weights = NULL,
  ensemble_or_discrete = c("ensemble", "discrete"),
  cv_schema = NULL,
  outcome_type = c("continuous", "binary", "density", "multiclass"),
  extra_learner_args = NULL,
  cluster_ids = NULL,
  strata_ids = NULL,
  weights = NULL,
  rowids = NULL,
  loss_metric = NULL,
  use_complete_cases = FALSE,
  inner_n_folds = NULL,
  inner_cv_schema = NULL
)
```

Arguments

`data` Data to use in training a `super_learner`.

`learners` A list of predictor/closure-returning-functions. See Details.

`formulas` Either a single regression formula or a vector of regression formulas.

`y_variable` Typically `y_variable` can be inferred automatically from the `formulas`, but if needed, the y_variable can be specified explicitly.

`n_folds` The number of cross-validation folds to use in constructing the `super_learner`. Must be a scalar integer of value >= 2.

`determine_super_learner_weights` A function/method to determine the weights for each of the candidate `learners`. The default is to use `determine_super_learner_weights_nnls`.

`ensemble_or_discrete` Defaults to `'ensemble'`, but can be set to `'discrete'`. Discrete `super_learner()` chooses only one of the candidate learners to have weight 1 in the resulting prediction algorithm, while `ensemble` `super_learner()` combines predictions from 1 or more candidate learners, with respective weights adding up to 1.

`cv_schema` A function that takes `data`, `n_folds` and returns a list containing `training_data` and `validation_data`, each of which are lists of `n_folds` data frames.

`outcome_type` One of 'continuous', 'binary', 'multiclass', or 'density'. `outcome_type` is used to infer the correct `determine_super_learner_weights` function if it is not explicitly passed.

`extra_learner_args` A list of equal length to the `learners` with additional arguments to pass to each of the specified learners.

`cluster_ids` (default: null) If specified, clusters will either be entirely assigned to training or validation (not both) in each cross-validation split.

`strata_ids` (default: null) If specified, strata are balanced across training and validation splits so that strata appear in both the training and validation splits.

`weights` If specified, (per observation) weights are used to indicate that risk minimization across models (i.e., the meta-learning step) should be targeted to higher weight observations.

`rowids` (default: null) If specified, rowids are stored for use in out-of-fold prediction (`$oof_predict()`) and can also be used to validate/observe the behavior of the cross-validation folds construction.

`loss_metric` A loss metric function, like the mean-squared-error or negative-log-loss to be used in evaluating the learners on held-out data and minimized through convex optimization. A loss metric should take two (vector) arguments: predictions, and true outcomes, and produce a single statistic summarizing the performance of each learner. Defaults to nadir's internal mean-squared-error function.

`use_complete_cases` (default: FALSE) If the `data` passed have any NA or NaN missing data, restrict the `data` to `data[complete.cases(data),]`.

`inner_n_folds` Number of folds used by the inner `super_learner()` on each outer training split to estimate ensemble weights. Defaults to `n_folds`, matching the historical behavior in which one fold count governed both.

`inner_cv_schema` Optional `cv_schema` for the inner `super_learner()` calls; defaults to `cv_schema` when one is supplied, and otherwise to `super_learner()`'s own defaults.

Details

The idea is that `cv_super_learner` splits the data into training/validation splits, trains `super_learner` on each training split, and then evaluates their predictions on the held-out validation data, calculating a root-mean-squared-error on those held-out data.

This function prints a message if the `loss_function` argument is not set explicitly, letting the user know that the mean-squared-error will be used by default. Pass in a loss function explicitly to `super_learner()` if you'd like to suppress this message, or use a similar approach for the appropriate loss function depending on context.

Value

A list containing `$trained_learners` and `$cv_loss` which respectively include 1) the trained super learner models on each fold of the data, their holdout predictions and, 2) the cross-validated estimate of the risk (expected loss) on held-out data.

Examples

```
cv_sl <- cv_super_learner(
  data = mtcars,
  formula = mpg ~ cyl + hp,
  learners = list(lnr_mean, lnr_lm)
```

```
)
print(cv_sl)
```

---

density_learners *Conditional Density Estimation in the* {nadir} *Package*

Description

The following learners are available for conditional density estimation:

- `lnr_lm_density`
- `lnr_glm_density`
- `lnr_homoscedastic_density`

Details

There are a few important things to know about conditional density estimation in the `nadir` package.

Firstly, conditional density learners must produce prediction functions that predict *densities* at the new outcome values given the new covariates.

Secondly, the implemented density estimators come in two flavors: those with a strong assumption (that of conditional normality), and those with much weaker assumptions. The strong assumption is encoded into learners like `lnr_lm_density` and `lnr_glm_density` and says "after we model the predicted mean given covariates, we expect the remaining errors to be normally distributed." The more flexible learners produced by `lnr_homoskedastic_density` are similar in spirit, except they fit a `stats::density` kernel bandwidth smoother to the error distribution (after predicting the conditional expected mean).

A subpoint to the above point that's worth calling attention to is that `lnr_homoskedastic_density` is a learner factory. That is to say, given a `mean_lnr`, `lnr_homoskedastic_density` produces a conditional density learner that uses that `mean_lnr`.

Work is ongoing on implementing a `lnr_heteroskedastic_density` learner that allows for predicting higher or lower variance in the conditional density given covariates.

Conditional density learners should be combined with the negative log loss function when using `super_learner()` or using `compare_learners`. Refer to the 2003 Dudoit and van der Laan paper for a starting place on the appropriate loss functions to use for different types of outcomes. https://biostats.bepress.com/ucbbiostat/paper130/

Value

Every learner function shares the same structure: when called with `(data, formula, ...)` it fits the underlying model and returns a *prediction closure* which is a function of `newdata` returning a numeric vector of predictions (predicted probabilities of the second factor level for binary learners; predicted densities for density learners; a matrix of class probabilities for multiclass learners).

See Also

learners binary_learners multiclass_learners

Examples

```
# density learners return a closure mapping newdata to the predicted
# conditional density of the *observed* outcome in newdata:
density_predictor <- lnr_lm_density(mtcars, mpg ~ hp + wt)
density_predictor(mtcars[1:5, ])


# inside super_learner(), with outcome_type = "density", candidate
# density learners are weighted by held-out negative log loss:
sl_density <- super_learner(
  data = mtcars,
  formulas = mpg ~ hp + wt,
  learners = list(lm_density = lnr_lm_density),
  outcome_type = "density",
  n_folds = 2
)
sl_density$learner_weights
```

---

determine_super_learner_weights_nnls

Determine SuperLearner Weights with Nonnegative Least Squares

---

Description

This function accepts a dataframe that is structured to have one column `Y` and other columns with unique names corresponding to different model predictions for `Y`, and it will use nonnegative least squares to determine the weights to use for a SuperLearner.

Usage

```
determine_super_learner_weights_nnls(data, y_variable, obs_weights = NULL)
```

Arguments

`data`	A data frame consisting of an outcome (y_variable) and other columns corresponding to predictions from candidate learners.
`y_variable`	The string name of the outcome column in `data`.
`obs_weights`	A vector of weights for each observation that dictate how prediction should be more targeted to higher weighted observations.

Value

A vector of weights to be used for each of the learners.

Examples

```
# suppose that we have a data.frame of predictions from different candidate
# learners:
prediction_data <- data.frame(
  lm = lnr_lm(mtcars, mpg ~ hp)(mtcars),
  rf = lnr_rf(mtcars, mpg ~ hp)(mtcars),
  rf2 = lnr_rf(mtcars, mpg ~ hp, ntree = 20)(mtcars),
  earth = lnr_earth(mtcars, mpg ~ hp)(mtcars)
)
# make sure it includes the outcome y_variable
prediction_data$mpg <- mtcars$mpg

# we can use determine_super_learner_weights() fn to apply the non-negative least
# squares algorithm to produce weights for averaging the learners
determine_super_learner_weights_nnls(
  data = prediction_data,
  y_variable = "mpg"
)
```

---

determine_weights_for_binary_outcomes

Determine Weights Appropriately for Super Learner given Binary Outcomes

---

Description

Determine Weights Appropriately for Super Learner given Binary Outcomes

Usage

```
determine_weights_for_binary_outcomes(
  data,
  y_variable,
  obs_weights = NULL,
  bound_eps = 0.001
)
```

Arguments

data	A data.frame with columns corresponding to predicted probabilities of 1 from each learner and the true y_variable from held-out data
y_variable	A character indicating the outcome variable in the data.frame.
obs_weights	A vector of weights for each observation that dictate how prediction should be more targeted to higher weighted observations.
bound_eps	A numeric value used to truncate probabilities away from 0 and 1.

Value

A vector of weights to be used for each of the learners.

Examples

```
predicted_probabilities <- data.frame(
  logistic = lnr_logistic(mtcars, am ~ hp)(mtcars),
  nnet = lnr_nnet(mtcars, am ~ hp)(mtcars),
  am = mtcars$am
)
determine_weights_for_binary_outcomes(predicted_probabilities, y_variable = "am")
```

---

determine_weights_using_neg_log_loss *Determine Weights for Density Estimators for SuperLearner*

Description

Determine Weights for Density Estimators for SuperLearner

Usage

```
determine_weights_using_neg_log_loss(
  data,
  y_variable,
  obs_weights = NULL,
  bound_eps = 0.001
)
```

Arguments

data	A data.frame with columns corresponding to predicted densities from each learner and the true y_variable from held-out data
y_variable	A character indicating the outcome variable in the data.frame.
obs_weights	A vector of weights for each observation that dictate how prediction should be more targeted to higher weighted observations.
bound_eps	A numeric value used for truncating probability (densities) away from 0.

Value

A vector of weights to be used for each of the learners.

Examples

```
predicted_densities <- data.frame(
  lm = lnr_lm_density(mtcars, mpg ~ hp)(mtcars),
  earth = lnr_homoskedastic_density(mtcars, mpg ~ hp, mean_lnr = lnr_earth)(mtcars),
  rf = lnr_homoskedastic_density(mtcars, mpg ~ hp, mean_lnr = lnr_rf)(mtcars),
  rf2 = lnr_homoskedastic_density(mtcars, mpg ~ hp,
    mean_lnr = lnr_rf,
    mean_lnr_args = list(ntree = 20)
  )(mtcars),
  mpg = mtcars$mpg
)
determine_weights_using_neg_log_loss(predicted_densities, y_variable = "mpg")
```

---

df_to_survival_stacked

Repeat Observations for Survival Stacking

Description

Per the approach in *A review of survival stacking: a method to cast survival regression analysis as a classification problem* doi:10.1515/ijb20220055 https://arxiv.org/abs/2107.13480, we provide `df_to_survival_stacked` as a helper function for converting traditional survival data (one observation = one row) into the survival stacked data structure, a repeated observations data structure where multiple rows exist for each individual for each timepoint at which they were still in the risk set up to and including their event time.

Usage

```
df_to_survival_stacked(
  data,
  id_col = NULL,
  time_col,
  status_col,
  covariate_cols,
  period_duration = 1,
  custom_times = NULL
)
```

Arguments

`data`	A data frame with survival -type outcomes including an event indicator and a time-to-event-or-censoring column
`id_col`	(string) name of the id column that is unique to each observation in `data`. If one is not specified, one will be created (called `.id`) assuming that each row is a unique observation.
`time_col`	(string) name of the time-to-event column
`status_col`	(string) name of the 0/1 event indicator column

`covariate_cols` (string vector) names of your predictors

`period_duration` (numeric) length of each time-period (e.g. 1)

`custom_times` (numeric vector) (optional) A vector of the time-period breakpoints. If events could have occurred at any time after zero, this should begin with 0.

Value

a data.frame of repeated observations, one row for each time-step, that indicates which observations remain in the risk set and whether or not an event occurs.

Examples

```
if (requireNamespace("survival", quietly = TRUE)) {
  df_to_survival_stacked(
    data = survival::kidney,
    id_col = "id",
    time_col = "time",
    status_col = "status",
    covariate_cols = c("age", "sex", "disease", "frail")
  )
}
```

---

learners *Learners in the* {nadir} *Package*

Description

The following learners are available for continuous outcomes:

Details

- `lnr_bart`
- `lnr_cvglmnet`
- `lnr_earth`
- `lnr_gam`
- `lnr_gausspr`
- `lnr_gbm`
- `lnr_glm`
- `lnr_glmer`
- `lnr_glmnet`
- `lnr_glmnet_grid`
- `lnr_hal`

- lnr_hal_grid
- lnr_knn
- lnr_lightgbm
- lnr_lm
- lnr_lmer
- lnr_mean
- lnr_ppr
- lnr_ranger
- lnr_rpart
- lnr_rf
- lnr_svm
- lnr_xgboost

See ?density_learners to learn more about using conditional density estimation in nadir.

lnr_mean is generally provided only for benchmarking purposes to compare other learners against to ensure correct specification of learners, since any prediction algorithm should (in theory) outperform just using the mean of the outcome for all predictions.

If you'd like to build a new learner, we recommend reading the source code of several of the learners provided with {nadir} to get a sense of how they should be specified.

A learner, as {nadir} understands them, is a function which takes in data, a formula, possibly ..., and returns a function that predicts on its input newdata.

A simple example is reproduced here for ease of reference:

Value

Every learner function shares the same structure: when called with (data, formula, ...) it fits the underlying model and returns a *prediction closure* which is a function of newdata returning a numeric vector of predictions (predicted probabilities of the second factor level for binary learners; predicted densities for density learners; a matrix of class probabilities for multiclass learners).

Examples

```
lnr_glm <- function(data, formula, weights = NULL, ...) {
  model <- stats::glm(formula = formula, data = data, weights = weights, ...)

  return(function(newdata) {
    predict(model, newdata = newdata, type = "response")
  })
}
```

list_known_learners *List Known Learners*

Description

List Known Learners

Usage

```
list_known_learners(
  type = c("any", "continuous", "binary", "density", "multiclass")
)
```

Arguments

type	One of 'any' or a supported outcome type in nadir including at least 'continuous', 'binary', 'multiclass', 'density'. See ?super_learner().

Value

A character vector of functions that were automatically recognized as nadir learners with the prediction/outcome type given.

Examples

```
list_known_learners()
list_known_learners("continuous")
list_known_learners("binary")
list_known_learners("density")
list_known_learners("multiclass")
```

lnr_bart *Bayesian Additive Regression Trees (BART) Learner*

Description

A wrapper for dbarts::bart2() for use in nadir::super_learner().

Usage

```
lnr_bart(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

BART is a Bayesian nonparametric sum-of-trees model with strong empirical performance in the causal inference and prediction literature; predictions returned are posterior mean predictions. Hyperparameters such as `n.trees`, `n.samples`, and `n.burn` may be passed through `....`

Categorical predictor levels are retained from the training data so that prediction data are encoded using the same design matrix as the training data, even when some factor levels are absent from `newdata`.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_bart(mtcars, mpg ~ hp + disp + wt)(mtcars)
```

---

lnr_cvglmnet *cv.glmnet Learner*

Description

A wrapper for `glmnet::cv.glmnet()` for use in `nadir::super_learner()`.

Usage

```
lnr_cvglmnet(data, formula, weights = NULL, lambda = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`lambda`	The multiplier parameter grid for the penalty; see `?glmnet::cv.glmnet`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

The returning prediction function defaults to passing `s = "lambda.min"` to the `predict.cv.glmnet` method built into `glmnet` defaults to predicting which says to use the minimum cross-validated loss lambda value from the CV grid `cv.glmnet` sets up. The other option is to pass `s = "lambda.1se"` to the returned prediction function which returns the largest lambda estimated to be within one standard deviation of the CV-optimal lambda according to the stored `cv.glmnet` object.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_cvglmnet(mtcars, mpg ~ hp + disp + am + wt)(mtcars)
```

`lnr_earth` *Earth Learner*

Description

A wrapper for `earth::earth()` for use in `nadir::super_learner()`.

Usage

```
lnr_earth(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_earth(mtcars, mpg ~ hp + disp + am + wt)(mtcars)
```

---

lnr_gam *Generalized Additive Model Learner*

Description

A wrapper for `mgcv::gam()` for use in `nadir::super_learner()`.

Usage

```
lnr_gam(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_gam(mtcars, mpg ~ s(hp) + disp + am + wt)(mtcars)
lnr_gam(mtcars, mpg ~ s(hp) + disp + am + wt, family = Gamma)(mtcars)
```

`lnr_gausspr` *Gaussian Process Regression Learner*

Description

A wrapper for `kernlab::gausspr()` for use in `nadir::super_learner()`.

Usage

```
lnr_gausspr(data, formula, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

Gaussian process regression is a Bayesian kernel method that places a prior over functions and returns the posterior mean prediction. It is a smooth, nonparametric paradigm distinct from both the tree ensembles and the penalized regressions already in the library. The kernel and its hyperparameters can be specified through `...` (e.g., `kernel = 'rbfdot'`, `kpar = list(sigma = 0.1)`).

Note that `kernlab::gausspr()` does not support observation weights, so no `weights` argument is accepted here.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_gausspr(mtcars, mpg ~ hp + disp + wt)(mtcars)
```

lnr_gbm *Gradient Boosting Machines Learner*

Description

A wrapper for gbm::gbm() for use in nadir::super_learner().

Usage

```
lnr_gbm(
  data,
  formula,
  weights = NULL,
  verbose = FALSE,
  n.minobsinnode = 0,
  ...
)
```

Arguments

data	A dataframe to train a learner / learners on.
formula	A regression formula to use inside this learner.
weights	Observation weights passed to gbm::gbm(). Defaults to NULL, in which case equal weights (rep(1, nrow(data))) are used.
verbose	(default: FALSE) if set to TRUE, information about the automatic outcome type inferred by gbm will be messaged to the console, as well as the number of trees used.
n.minobsinnode	(default: 0) An integer specifying the minimum number of observations in the terminal nodes of the trees. See the gbm documentation for more. Set here to 0 to account for the potential of very small splits in cross-fitting.
...	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts newdata, which returns predictions (a numeric vector of values, one for each row of newdata).

See Also

learners

Examples

```
lnr_gbm(mtcars, mpg ~ hp)(mtcars)
```

lnr_glm — *GLM Learner*

Description

A wrapper for `stats::glm()` for use in `nadir::super_learner()`.

Usage

```
lnr_glm(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_glm(mtcars, mpg ~ hp + disp + am + wt)(mtcars)
lnr_glm(mtcars, mpg ~ hp + disp + am + wt, family = Gamma)(mtcars)
```

lnr_glm_density — *Conditional Normal Density Estimation Given Mean Predictors — with GLMs*

Description

This is a step up from the `lnr_lm_density` in that it uses a `glm` for the conditional mean model. Note that this allows for specification of `glm` features like `family = ...` in the `, ..` arguments, and that's the main advantage over the `lnr_lm_density`. Also note that this still differs from using `lnr_homoskedastic_density` with `mean_lnr = lnr_glm` because `lnr_homoscedastic_density` uses `stats::density` to do kernel bandwidth smoothing on the error distribution of the mean predictions..

Usage

```
lnr_glm_density(data, formula, weights = NULL, ...)
```

Arguments

data	A dataframe to train a learner / learners on.
formula	A regression formula to use inside this learner.
weights	Observation weights; see ?lm
...	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

a closure (function) that produces density estimates at the newdata given according to the fit model.

Examples

```
# for example, we could use a Poisson assumption with identity link:

lnr_glm_density(mtcars, hp ~ mpg, family = poisson(link = "identity"))(mtcars)
hp_seq <- seq(min(mtcars$hp), max(mtcars$hp), length.out = 1000)
plot(
  x = hp_seq,
  y = lnr_glm_density(mtcars, hp ~ mpg, family = poisson(link = "identity"))(
    data.frame(hp = hp_seq, mpg = rep(mean(mtcars$mpg), 1000))),
  xlab = "hp",
  ylab = "density",
  main = "normal density model of horsepower given mean(mpg)"
)
```

---

lnr_glmer — *Generalized Linear Mixed-Effects (*lme4::glmer*) Learner*

Description

A wrapper for lme4::glmer() for use in nadir::super_learner().

Usage

```
lnr_glmer(data, formula, weights = NULL, ...)
```

Arguments

data	A dataframe to train a learner / learners on.
formula	A regression formula to use inside this learner.
weights	Observation weights; see ?lm
...	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
# random intercepts for each level of cyl column:
suppressMessages({
  # singular fit, but that's ok if all you need is prediction:
  lnr_glmer(mtcars, mpg ~ (1 | cyl) + disp + wt, family = Gamma)(mtcars)
})
```

`lnr_glmnet` *glmnet Learner*

Description

A wrapper for `glmnet::glmnet()` for use in `nadir::super_learner()`.

Usage

```
lnr_glmnet(data, formula, weights = NULL, lambda = 0.2, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`lambda`	The multiplier parameter for the penalty; see `?glmnet::glmnet`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

glmnet predictions will by default, if lambda is unspecified, return a matrix of predictions for varied lambda values, hence the need to explicitly handle the lambda argument in building glmnet learners.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_glmnet(mtcars, mpg ~ hp + disp + am + wt, lambda = .5)(mtcars)
```

---

lnr_glmnet_grid *glmnet Learner over a Grid of Lambda Values*

Description

A wrapper for `glmnet::glmnet()` fit jointly over a decreasing grid of lambda values, for use in `nadir::super_learner()`.

Usage

```
lnr_glmnet_grid(data, formula, weights = NULL, lambda, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`lambda`	A numeric vector of penalty values (a grid). Must be explicitly specified; see `validate_lambda_grid` for why auto-generated grids are not supported. Values are deduplicated and sorted in decreasing order.
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

`glmnet` exploits warm starts along the regularization path, so fitting an entire grid of lambda values costs little more than fitting one – and is often *faster* than fitting a single small lambda from a cold start. `lnr_glmnet_grid` fits the whole path once and returns a multi-predictor: one prediction function per lambda value. `super_learner()` expands these into distinct pseudo-learners (named e.g. `glmnet_grid_lambda_0.1`) which each receive their own weight in the meta-learning stage.

Compare with `lnr_glmnet`, which fits a single lambda value, and `lnr_cvglmnet`, which internally selects one lambda by cross-validation and exposes only that model to the meta-learner.

Value

A `nadir_multi_predictor`: a named list of prediction functions, one per lambda value, each of which accepts `newdata` and returns a numeric vector of predictions.

See Also

learners lnr_glmnet lnr_hal_grid as_multi_predictor

Examples

```
multi_predictor <- lnr_glmnet_grid(
  mtcars, mpg ~ hp + disp + am + wt,
  lambda = c(0.01, 0.1, 0.5, 1)
)
names(multi_predictor)
multi_predictor[["lambda_0.5"]](mtcars)
```

---

lnr_hal *Highly Adaptive Lasso*

Description

A wrapper for `hal9001::fit_hal()` for use in `nadir::super_learner()`.

Usage

```
lnr_hal(data, formula, weights = NULL, lambda = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`lambda`	The multiplier parameter for the penalty; see `?glmnet::glmnet`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

If a single `lambda` value is specified, `fit_control$cv_select` is set to `FALSE`, since `fit_hal`'s default (`cv_select = TRUE`) hands `lambda` to `cv.glmnet`, which errors with "Need more than one value of lambda for cv.glmnet" — and inside `super_learner()` that error would cause `lnr_hal` to be silently dropped from the ensemble.

If a vector of `lambda` values is specified (with the default `fit_control`), the grid is treated as a search space: `cv.glmnet` internally selects a single lambda, and only that model's predictions are returned. To instead expose *each* lambda value in a grid to the meta-learning stage of `super_learner()` as its own candidate learner, use `lnr_hal_grid`.

Value

A prediction function that accepts newdata, which returns predictions (a numeric vector of values, one for each row of newdata).

See Also

learners lnr_hal_grid

Examples

```
suppressWarnings({
  # hal prints a lot of messages about some threads not reaching convergence
  lnr_hal(mtcars, mpg ~ hp + am + cyl + disp)(mtcars)
})
```

---

lnr_hal_grid *Highly Adaptive Lasso over a Grid of Lambda Values*

Description

A wrapper for hal9001::fit_hal() fit jointly over a decreasing grid of lambda values, for use in nadir::super_learner().

Usage

```
lnr_hal_grid(data, formula, weights = NULL, lambda, ...)
```

Arguments

data	A dataframe to train a learner / learners on.
formula	A regression formula to use inside this learner.
weights	Observation weights; see ?lm
lambda	A numeric vector of penalty values (a grid). Must be explicitly specified; see validate_lambda_grid for why auto-generated grids are not supported. Values are deduplicated and sorted in decreasing order.
...	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

Like lnr_glmnet_grid, this fits the entire lasso path (over the HAL basis expansion) in a single call and returns a multi-predictor with one prediction function per lambda value, which super_learner() expands into distinct pseudo-learners. Internally fit_hal is called with fit_control = list(cv_select = FALSE) so that the fits at every lambda in the grid are retained rather than one being selected by cv.glmnet.

Since the HAL basis expansion (typically the expensive step at prediction time) is shared across the whole grid, predictions for all lambda values on the same newdata are computed once and cached.

Value

A `nadir_multi_predictor`: a named list of prediction functions, one per lambda value, each of which accepts `newdata` and returns a numeric vector of predictions.

See Also

learners lnr_hal lnr_glmnet_grid as_multi_predictor

Examples

```
suppressWarnings({
  multi_predictor <- lnr_hal_grid(
    mtcars, mpg ~ hp + wt,
    lambda = c(0.01, 0.1, 1),
    max_degree = 1, num_knots = 3
  )
})
names(multi_predictor)
```

---

lnr_heteroskedastic_density

Conditional Density Estimation with Heteroskedasticity

Description

Conditional Density Estimation with Heteroskedasticity

Usage

```
lnr_heteroskedastic_density(
  data,
  formula,
  mean_lnr,
  var_lnr,
  mean_lnr_args = NULL,
  var_lnr_args = NULL,
  density_args = NULL
)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`mean_lnr`	A learner (function) passed in to be trained on the data with the given formula and then used to predict conditional means for provided `newdata`.

var_lnr	A learner (function) passed in to be trained on the squared error from the mean_lnr on the given data and then used to predict the expected variance for the density distribution of the outcome centered around the predicted conditional mean in the output.
mean_lnr_args	Extra arguments to be passed to the mean_lnr
var_lnr_args	Extra arguments to be passed to the var_lnr
density_args	Extra arguments to be passed to the kernel density smoother stats::density, especially things like bw for specifying the smoothing bandwidth. See ?stats::density.

Value

a closure (function) that produces density estimates at the newdata given according to the fit model.

Examples

```
# fit a conditional density model with mean model as a randomForest
fit_density_hetero <- lnr_heteroskedastic_density(
  data = mtcars,
  formula = mpg ~ hp,
  mean_lnr = lnr_rf,
  var_lnr = lnr_lm
)

# and what we should get back should be predicted densities at the
# observed mpg given the covariates hp
fit_density_hetero(mtcars)

if (requireNamespace("ggplot2", quietly = TRUE)) {
  hp_grid <- with(mtcars, seq(min(hp), max(hp), length.out = 100))
  mpg_grid <- with(mtcars, seq(min(mpg), max(mpg), length.out = 100))
  mt_grid <- expand.grid(mpg = mpg_grid, hp = hp_grid)
  plt_df <- cbind(mt_grid, pred_dens = fit_density_hetero(mt_grid))
  require(ggplot2)
  ggplot(plt_df, aes(x = hp, y = mpg, fill = pred_dens)) +
    geom_tile() +
    scale_fill_viridis_c() +
    ggtitle("Density Model of MPG given HP")
}
```

---

lnr_homoskedastic_density

Conditional Density Estimation with Homoskedasticity Assumption

---

Description

This function accepting an mean_lnr, which it then trains on the data and formula given. Then stats::density is fit to the error (difference between observed outcome and the mean_lnr predictions).

Usage

```
lnr_homoskedastic_density(
  data,
  formula,
  mean_lnr,
  mean_lnr_args = NULL,
  density_args = NULL,
  weights = NULL
)
```

Arguments

data	A dataframe to train a learner / learners on.
formula	A regression formula to use inside this learner.
mean_lnr	A learner (function) passed in to be trained on the data with the given formula and then used to predict conditional means for provided newdata.
mean_lnr_args	Extra arguments to be passed to the mean_lnr
density_args	Extra arguments to be passed to the kernel density smoother stats::density, especially things like bw for specifying the smoothing bandwidth. See ?stats::density.
weights	Observation weights; see ?lm

Details

This returns a function that takes in newdata and produces density estimates according to the estimated stats::density fit the error from the newdata observed outcome and the prediction from the mean_lnr.

That is to say, this follows the following procedure (assuming Y as the outcome and X as a matrix of predictors):

$$\texttt{obtain } \hat{\mathbb{E}}(Y|X) \texttt{ using mean_learner}$$

$$\texttt{fit } \hat{f} \leftarrow \texttt{density}(Y - \hat{\mathbb{E}}(Y|X))$$

$$\texttt{return function(newdata)}\{\hat{f}(\texttt{newdata\$Y} - \hat{\mathbb{E}}[Y|\texttt{newdata\$X}])\}$$

Value

A predictor function that takes in newdata and produces density estimates

Examples

```
# fit a conditional density model with mean model as a randomForest
fit_density_lnr <- lnr_homoskedastic_density(
  data = mtcars,
  formula = mpg ~ hp,
  mean_lnr = lnr_rf
)
```

```
# and what we should get back should be predicted densities at the
# observed mpg given the covariates hp
fit_density_lnr(mtcars)
```

`lnr_knn` *k-Nearest Neighbors Learner*

Description

A wrapper for `kknn::kknn()` for use in `nadir::super_learner()`.

Usage

```
lnr_knn(data, formula, k = 7, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`k`	The number of nearest neighbors to use; see `?kknn::kknn`.
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

k-nearest neighbors is a "lazy" learner: no model is fit at training time. Instead, the training data are stored and predictions for `newdata` are formed as a (kernel-weighted) average of the outcomes of the `k` nearest training observations in covariate space. As a purely local, instance-based method, kNN occupies a very different corner of the bias-variance landscape than the global regression methods (`lnr_lm`, `lnr_glmnet`, etc.) and the tree ensembles (`lnr_rf`, `lnr_ranger`, `lnr_xgboost`), making it a useful addition to a super learner library.

Note that `kknn::kknn()` does not support observation weights, so no `weights` argument is accepted here.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_knn(mtcars, mpg ~ hp + disp + wt)(mtcars)
lnr_knn(mtcars, mpg ~ hp + disp + wt, k = 5)(mtcars)
```

`lnr_knn_binary` *k-Nearest Neighbors Learner for Binary Classification*

Description

A wrapper for `kknn::kknn()` for use in `nadir::super_learner()` with binary outcomes. Predicted probabilities for the outcome being 1/TRUE are the (kernel-weighted) proportion of the `k` nearest neighbors with outcome 1/TRUE.

Usage

```
lnr_knn_binary(data, formula, k = 7, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`k`	The number of nearest neighbors to use; see `?kknn::kknn`.
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

Note that `kknn::kknn()` does not support observation weights, so no `weights` argument is accepted here.

Value

A prediction function that accepts `newdata`, which returns predictions for the probability of the outcome being 1/TRUE (a numeric vector of values, one for each row of `newdata`).

See Also

binary_learners

Examples

```
lnr_knn_binary(mtcars, am ~ hp + mpg)(mtcars)
```

lnr_lightgbm *LightGBM Learner*

Description

A wrapper for `lightgbm::lgb.train()` for use in `nadir::super_learner()`.

Usage

```
lnr_lightgbm(
  data,
  formula,
  weights = NULL,
  nrounds = 100,
  objective = NULL,
  verbose = -1,
  ...
)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Optional observation weights.
`nrounds`	Number of boosting iterations.
`objective`	Optional objective. For binary Super Learner fits, `nadir` supplies `objective = "binary"` through `outcome_type_dependent_args`. If unspecified, defaults to `"regression"`.
`verbose`	(default: -1) Verbosity level passed to `lightgbm`; -1 suppresses lightgbm's messages, larger values (0, 1, 2) show progressively more output.
`...`	Additional named parameters passed into the `params` list of `lightgbm::lgb.train()`, e.g. `learning_rate`, `num_leaves`, `min_data_in_leaf`.

Details

`{lightgbm}` does not have a formula interface, so internally this learner constructs a numeric design matrix via `stats::model.matrix.lm()` (as in `lnr_xgboost`) and trains on that.

Note that `min_data_in_leaf` defaults to 1 here (rather than the `{lightgbm}` default of 20) to account for the potential of very small splits in cross-fitting; users can override this by passing `min_data_in_leaf` explicitly.

Value

A prediction function that accepts `newdata` and returns a numeric vector of predictions (probabilities of the outcome being 1 when `objective = "binary"`), one for each row of `newdata`.

See Also

learners

Examples

```
lnr_lightgbm(mtcars, mpg ~ hp + wt, nrounds = 10)(mtcars)

lnr_lightgbm(mtcars, am ~ cyl + disp + hp,
  objective = "binary",
  nrounds = 10
)(mtcars)
```

---

lnr_lm *Linear Model Learner*

Description

A wrapper for `lm()` for use in `nadir::super_learner()`.

Usage

```
lnr_lm(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_lm(mtcars, mpg ~ hp + disp + am + wt)(mtcars)
```

`lnr_lm_density` *Conditional Normal Density Estimation Given Mean Predictors*

Description

This is the simplest possible density estimator that is entertainable. It fits a `lm` model to the data, and uses the variance of the residuals to parameterize a model of the data as $\mathcal{N}(y|\beta x, \sigma^2)$.

Usage

```
lnr_lm_density(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

a closure (function) that produces density estimates at the `newdata` given according to the fit model.

Examples

```
lnr_lm_density(mtcars, hp ~ mpg)(mtcars)
hp_seq <- seq(min(mtcars$hp), max(mtcars$hp), length.out = 1000)
plot(
  x = hp_seq,
  y = lnr_lm_density(mtcars, hp ~ mpg)(
    data.frame(hp = hp_seq, mpg = rep(mean(mtcars$mpg), 1000))),
  xlab = "hp",
  ylab = "density",
  main = "normal density model of horsepower given mean(mpg)"
)
```

lnr_lmer — *Random/Mixed-Effects (lme4::lmer) Learner*

Description

A wrapper for `lme4::lmer` for use in `nadir::super_learner()`.

Usage

```
lnr_lmer(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
# random intercepts for each level of cyl column:
lnr_lmer(mtcars, mpg ~ (1 | cyl) + disp + am + wt)(mtcars)
```

lnr_logistic — *Standard Logistic Regression for Binary Classification*

Description

A wrapper provided for convenience around `lnr_glm` that sets `family = binomial(link = 'logit')`.

Usage

```
lnr_logistic(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions for the probability of the outcome being 1/TRUE (a numeric vector of values, one for each row of `newdata`).

Examples

```
lnr_logistic(mtcars, am ~ hp)(mtcars)
```

`lnr_mean` *Mean Learner*

Description

This is a very naive/simple learner that simply predicts the mean of the outcome for every row of input `newdata`. This is primarily useful for benchmarking and confirming that other learners are performing better than `lnr_mean`. Additionally, it may be the case that some learners are over-fitting the data, and giving some weight to `lnr_mean` helps to reduce over-fitting in `super_learner()`.

Usage

```
lnr_mean(data, formula, weights = NULL)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_mean(mtcars, mpg ~ hp)(mtcars)
```

lnr_multinomial_nnet nnet::multinom *Multinomial Learner*

Description

nnet::multinom Multinomial Learner

Usage

```
lnr_multinomial_nnet(data, formula, weights = NULL, ...)
```

Arguments

data	A dataframe to train a learner / learners on.
formula	A regression formula to use inside this learner.
weights	Observation weights; see ?lm
...	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts newdata, which returns predictions (a numeric vector of density prediction values at the outcome value observed in the newdata conditioning on the predictor variables in newdata).

Examples

```
df <- mtcars
df$cyl <- as.factor(df$cyl)
lnr_multinomial_nnet(df, cyl ~ hp + mpg)(df)
lnr_multinomial_nnet(iris, Species ~ .)(iris)
```

lnr_multinomial_ranger *ranger Multinomial Learner*

Description

A wrapper for ranger::ranger() with probability = TRUE for use in nadir::super_learner() with multiclass outcomes. This adds a flexible, nonparametric tree-ensemble counterpart to the parametric multinomial learners lnr_multinomial_vglm and lnr_multinomial_nnet.

Usage

```
lnr_multinomial_ranger(data, formula, weights = NULL, ...)
```

Arguments

data	A dataframe to train a learner / learners on.
formula	A regression formula to use inside this learner.
weights	Observation weights; see ?lm
...	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts newdata, which returns predictions (a numeric vector of density prediction values at the outcome value observed in the newdata conditioning on the predictor variables in newdata).

Examples

```
df <- mtcars
df$cyl <- as.factor(df$cyl)
lnr_multinomial_ranger(df, cyl ~ hp + mpg)(df)
lnr_multinomial_ranger(iris, Species ~ .)(iris)
```

---

lnr_multinomial_vglm VGAM::vglm *Multinomial Learner*

Description

VGAM::vglm Multinomial Learner

Usage

```
lnr_multinomial_vglm(data, formula, ...)
```

Arguments

data	A dataframe to train a learner / learners on.
formula	A regression formula to use inside this learner.
...	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts newdata, which returns predictions (a numeric vector of density prediction values at the outcome value observed in the newdata conditioning on the predictor variables in newdata).

Examples

```
df <- mtcars
df$cyl <- as.factor(df$cyl)
lnr_multinomial_vglm(df, cyl ~ hp + mpg)(df)
lnr_multinomial_vglm(iris, Species ~ .)(iris)
```

`lnr_nnet` *Use nnet for Binary Classification*

Description

Use nnet for Binary Classification

Usage

```
lnr_nnet(data, formula, trace = FALSE, size, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`trace`	Whether nnet should print out its optimization success
`size`	Size for neural network hidden layer
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

Examples

```
lnr_nnet(mtcars, am ~ ., size = 50)(mtcars)
lnr_nnet(iris, I(Species == "setosa") ~ ., size = 50)(iris)
```

`lnr_ppr` *Projection Pursuit Regression Learner*

Description

A wrapper for `stats::ppr()` for use in `nadir::super_learner()`.

Usage

```
lnr_ppr(data, formula, weights = NULL, nterms = 3, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`nterms`	Number of ridge terms to include in the final model; see `?stats::ppr`.
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

Projection pursuit regression models the outcome as a sum of smooth ridge functions of linear combinations of the covariates, $\hat{y} = \sum_m g_m(\alpha_m^\top x)$. It captures interactions and nonlinearities along learned directions in covariate space, and being part of base R's `stats` package, adds no new dependencies.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_ppr(mtcars, mpg ~ hp + disp + wt)(mtcars)
lnr_ppr(mtcars, mpg ~ ., nterms = 4)(mtcars)
```

lnr_ranger *ranger Learner*

Description

A wrapper for `ranger::ranger()` for use in `nadir::super_learner()`.

Usage

```
lnr_ranger(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_ranger(mtcars, mpg ~ hp)(mtcars)
```

lnr_ranger_binary *ranger Learner for Binary Outcomes*

Description

A wrapper for `ranger::ranger()` for use in `nadir::super_learner()`.

Usage

```
lnr_ranger_binary(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_ranger_binary(mtcars, am ~ hp)(mtcars)
```

---

lnr_rf *randomForest Learner*

Description

A wrapper for `randomForest::randomForest()` for use in `nadir::super_learner()`.

Usage

```
lnr_rf(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_rf(mtcars, mpg ~ hp + disp + am + wt, ntree = 20)(mtcars)
```

`lnr_rf_binary` *Use Random Forest for Binary Classification*

Description

Use Random Forest for Binary Classification

Usage

```
lnr_rf_binary(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Value

A prediction function that accepts `newdata`, which returns predictions for the probability of the outcome being 1/TRUE (a numeric vector of values, one for each row of `newdata`).

Examples

```
lnr_rf_binary(mtcars, am ~ hp)(mtcars)
```

`lnr_rpart` *Recursive Partitioning (CART) Learner*

Description

A wrapper for `rpart::rpart()` for use in `nadir::super_learner()`.

Usage

```
lnr_rpart(data, formula, weights = NULL, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

A single regression tree is a classic member of super learner libraries: it is fast, handles interactions and nonlinearities automatically, and its predictions are piecewise-constant, complementing the smooth learners in the library. Complexity may be controlled through `rpart.control` arguments passed via `...` (e.g., `cp`, `minsplit`, `maxdepth`).

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_rpart(mtcars, mpg ~ hp + disp + wt)(mtcars)
lnr_rpart(mtcars, mpg ~ ., cp = 0.05)(mtcars)
```

`lnr_rpart_binary` *Recursive Partitioning (CART) Learner for Binary Classification*

Description

A wrapper for `rpart::rpart()` with `method = 'class'` for use in `nadir::super_learner()` with binary outcomes.

Usage

```
lnr_rpart_binary(data, formula, weights = NULL, bound = 0.0025, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Observation weights; see `?lm`
`bound`	Predicted probabilities are truncated into `[bound, 1 - bound]` to avoid infinite negative log loss from pure terminal nodes.
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

Because classification trees can produce pure terminal nodes, raw predicted probabilities of exactly 0 or 1 are possible, which yield infinite negative log loss on held-out data where such predictions are wrong. To keep `lnr_rpart_binary` compatible with `determine_weights_for_binary_outcomes`, predicted probabilities are bounded into `[bound, 1 - bound]`. Set `bound = 0` to disable this behavior.

Value

A prediction function that accepts `newdata`, which returns predictions for the probability of the outcome being 1/TRUE (a numeric vector of values, one for each row of `newdata`).

See Also

binary_learners

Examples

```
lnr_rpart_binary(mtcars, am ~ hp + mpg)(mtcars)
```

`lnr_svm` *Support Vector Machine Learner*

Description

A wrapper for `e1071::svm()` for use in `nadir::super_learner()`, performing support vector (eps-)regression.

Usage

```
lnr_svm(data, formula, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

Support vector regression fits a function in a kernel-induced feature space (radial basis by default) that is at most ε away from the observed outcomes wherever possible, yielding a flexible, margin-based regression paradigm not otherwise represented in the built-in learners. Kernel choice and hyperparameters (e.g., `kernel`, `cost`, `gamma`, `epsilon`) may be passed through `....`

Note that `e1071::svm()` does not support observation weights, so no `weights` argument is accepted here.

Value

A prediction function that accepts `newdata`, which returns predictions (a numeric vector of values, one for each row of `newdata`).

See Also

learners

Examples

```
lnr_svm(mtcars, mpg ~ hp + disp + wt)(mtcars)
lnr_svm(mtcars, mpg ~ ., kernel = "polynomial", cost = 2)(mtcars)
```

`lnr_svm_binary` *Support Vector Machine Learner for Binary Classification*

Description

A wrapper for `e1071::svm()` with `probability = TRUE` for use in `nadir::super_learner()` with binary outcomes.

Usage

```
lnr_svm_binary(data, formula, ...)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`...`	Any extra arguments that should be passed to the internal model for model fitting purposes.

Details

Predicted probabilities are obtained via Platt scaling (see `?e1071::svm`) and returned for the outcome being 1/TRUE.

Note that `e1071::svm()` does not support observation weights, so no `weights` argument is accepted here.

Value

A prediction function that accepts `newdata`, which returns predictions for the probability of the outcome being 1/TRUE (a numeric vector of values, one for each row of `newdata`).

See Also

binary_learners

Examples

```
lnr_svm_binary(mtcars, am ~ hp + mpg)(mtcars)
```

lnr_xgboost *XGBoost Learner*

Description

A wrapper for `xgboost::xgb.train()` for use in `nadir::super_learner()`.

Usage

```
lnr_xgboost(
  data,
  formula,
  weights = NULL,
  nrounds = 1000,
  xgb.params = xgboost::xgb.params(),
  objective = NULL,
  ...
)
```

Arguments

`data`	A dataframe to train a learner / learners on.
`formula`	A regression formula to use inside this learner.
`weights`	Optional observation weights.
`nrounds`	Number of boosting iterations.
`xgb.params`	An `xgboost::xgb.params()` object, or a named list of parameters that can be passed to `xgboost::xgb.params()`.
`objective`	Optional objective. If supplied, this is inserted into `xgb.params`. For binary Super Learner fits, `nadir` supplies `objective = "binary:logistic"` through `outcome_type_dependent_args`.
`...`	Additional arguments passed to `xgboost::xgb.params()`.

Details

This version avoids the high-level `xgboost::xgboost()` wrapper and instead uses the lower-level `xgb.DMatrix()` + `xgb.train()` interface. This is more stable for programmatic Super Learner use because `xgb.train()` supports the full range of XGBoost objectives.

Value

A prediction function that accepts `newdata` and returns a numeric vector of predictions.

See Also

learners

Examples

```
lnr_xgboost(mtcars, mpg ~ hp, nrounds = 5)(mtcars)

lnr_xgboost(
  mtcars,
  am ~ cyl + disp + hp,
  objective = "binary:logistic",
  nrounds = 5
)(mtcars)
```

---

make_learner_names_unique

Make Unique Learner Names

Description

Make Unique Learner Names

Usage

```
make_learner_names_unique(learners)
```

Arguments

learners	A list of learners. See ?learners

Value

A list of learners with (possibly) improved names.

Examples

```
learners <-
  list(
    mean = lnr_mean,
    rf = lnr_rf,
    rf = lnr_rf,
    lnr_glm,
    lnr_xgboost,
    function(data, formula) {},
    function(data, formula) {}
  )
learners <- nadir::make_learner_names_unique(learners)
names(learners)

learners <-
  list(
    lnr_mean,
```

```
    lnr_rf,
    lnr_rf,
    lnr_glm,
    lnr_xgboost,
    function(data, formula) {},
    function(data, formula) {}
  )
learners <- nadir::make_learner_names_unique(learners)
names(learners)
```

`multiclass_learners` *Multiclass Learners in* `{nadir}`

Description

- `lnr_multinomial_nnet`
- `lnr_multinomial_vglm`

Details

Suppose one of these is trained on some data and the fit learner is stored. Suppose we are going to call it on `newdata` and `newdata$class` is the outcome variable being predicting.

The important thing to know about multiclass learners is that they produce predictions that the outcome class is equal to `newdata$class` given the covariates specified in `newdata`.

This means that `newdata` passed to the returned prediction closure must contain the outcome column, or else an error is produced.

Similar to density estimation, we want to use `determine_weights_using_neg_log_loss` in our calls to `super_learner()`. This can be done automatically by declaring `outcome_type = 'multiclass'` in calling `super_learner()`

Value

Every learner function shares the same structure: when called with `(data, formula, ...)` it fits the underlying model and returns a *prediction closure* which is a function of `newdata` returning a numeric vector of predictions (predicted probabilities of the second factor level for binary learners; predicted densities for density learners; and a vector of the predicted class probabilities for the observed classes for multiclass learners).

See Also

density_learners binary_learners learners

Examples

```
super_learner(
  data = iris,
  learners = list(lnr_multinomial_vglm, lnr_multinomial_vglm, lnr_multinomial_nnet),
  formulas = list(
    .default = Species ~ .,
    multinomial_vglm_2 = Species ~ Petal.Length * Petal.Width + .
  ),
  outcome_type = "multiclass"
)
```

---

nadir_sl_model_methods

Print a `nadir_sl_model`

Description

Summarises the model inputs (outcome, outcome type, number of observations and cross-validation folds) and outputs (ensemble weights), plus any errors captured during training.

Combines the ensemble weights with each candidate learner's cross-validated held-out loss, using the loss appropriate to the model's `outcome_type` (mean squared error for continuous outcomes; negative log loss for binary, multiclass, and density outcomes).

Following the convention of `coef.SuperLearner` in the **SuperLearner** package, the "coefficients" of a super learner are the meta-learned ensemble weights on the candidate learners.

Returns the ensemble's *cross-validated* predictions on the held-out folds: each value is the learner-weighted prediction for an observation, produced by learners trained without that observation. These are honest (out-of-fold) rather than in-sample fitted values.

Observed outcomes minus the cross-validated ensemble predictions of `fitted.nadir_sl_model`. I.e., honest out-of-fold residuals. See that function's documentation for the row-ordering convention.

Returns the regression formula shared by all learners when a single formula was used, and otherwise the named list of per-learner formulas.

If the model was fit with `train_on_whole_dataset = FALSE`, no whole-dataset learner fits exist and this method errors with a message directing users to the out-of-fold prediction interfaces (`$oof_predictions`, `$oof_predict()`, `$oof_predict_modified()`, `$oof_predict_fold()`).

Usage

```
## S3 method for class 'nadir_sl_model'
print(x, digits = 3, ...)

## S3 method for class 'nadir_sl_model'
summary(object, ...)
```

```
## S3 method for class 'nadir_sl_model'
coef(object, ...)

## S3 method for class 'nadir_sl_model'
fitted(object, ...)

## S3 method for class 'nadir_sl_model'
residuals(object, ...)

## S3 method for class 'nadir_sl_model'
formula(x, ...)

## S3 method for class 'nadir_sl_model'
nobs(object, ...)

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

Arguments

`x`	An object of class `nadir_sl_model`.
`digits`	Number of digits for the printed weights.
`...`	Ellipses, solely provided so that the `predict.nadir_sl_model` method is compatible with the generic `predict`, which takes ellipses as an argument.
`object`	An object of class inheriting from `nadir_sl_model`.
`newdata`	A tabular data structure (data.frame or matrix) of predictor variables.

Details

Values are returned in cross-validation fold order, matching the rows of `$holdout_predictions` (which contains the corresponding observed outcomes), not in the row order of the originally supplied data. For out-of-fold predictions in original row order, use `crossfit_super_learner()$oof_predictions`.

Value

`x`, invisibly.

An object of class `summary.nadir_sl_model`: a list with a `$comparison` data.frame (one row per learner: weight and held-out loss, best first) plus `$y_variable`, `$outcome_type`, `$n_obs`, and `$n_folds`.

A named numeric vector of ensemble weights summing to 1.

A numeric vector of cross-validated ensemble predictions (or, for density/multiclass outcomes, ensemble mixture densities or probabilities of the observed outcomes).

A numeric vector of out-of-fold residuals. Errors for `outcome_type = 'density'` or `'multiclass'`, where held-out predictions are not point predictions.

A `formula` if all learners share one; otherwise a named list of formulas, one per learner.

Integer number of rows of the (complete-case-filtered) training data.

a numeric vector of predicted values

Examples

```
sl_fit <- super_learner(mtcars, mpg ~ hp,
  learners = list(lnr_mean, lnr_lm)
)
predict(sl_fit, newdata = mtcars)
```

---

nadir_supported_types *Outcome types supported by* {nadir}

Description

The following outcome types are supported in the {nadir} package:

Usage

```
nadir_supported_types
```

Details

- continuous
- binary
- multiclass
- density

Value

A character vector of the outcome types nadir supports: "continuous", "binary", "multiclass", "density".

See Also

super_learner

Examples

```
nadir_supported_types
```

negative_log_loss *Negative Log Loss*

Description

Negative Log Loss

Usage

```
negative_log_loss(predicted_densities, ...)
```

Arguments

`predicted_densities`
: The predicted densities from a learner predicted at `newdata`.

`...`
: Because `nadir::compare_learners()` passes `estimates, truth` to the `loss_metric` passed to it, `negative_log_loss` accepts ... but doesn't do anything with it.

Details

`negative_log_loss` encodes the logic: if $\hat{p}_n$ is a good model of the conditional densities, then it should minimize:

$$-\sum(\log(\hat{p}_n(X_i))$$

Value

A sum of the negative log loss given a vector of predicted probabilities/densities for some observed outcome.

Examples

```
# suppose we have some prediction probabilities _at the true values_:
predicted_probabilities <- lnr_logistic(mtcars, am ~ hp)(mtcars)
# we can calculate the -log(loss) for binary predicted probabilities like so:
negative_log_loss(predicted_probabilities)
```

negative_log_loss_for_binary *Negative Log Loss for Binary*

Description

Negative Log Loss for Binary

Usage

```
negative_log_loss_for_binary(predicted_probabilities, true_outcomes)
```

Arguments

`predicted_probabilities`
: The predicted probabilities from a learner predicted at `newdata`.

`true_outcomes`
: A vector of true outcomes to use in calculating the negative log loss of the relevant predicted probabilities.

Value

A sum of the negative log loss given a vector of predicted probabilities for `outcome == 1` or equivalently a 'success'.

Examples

```
# summed negative log loss of predicted P(y = 1) against 0/1 outcomes;
# confident-and-right predictions contribute little loss ...
negative_log_loss_for_binary(c(0.9, 0.2, 0.8), c(1, 0, 1))
# ... while a confident-and-wrong prediction is penalized heavily
negative_log_loss_for_binary(c(0.05, 0.2, 0.8), c(1, 0, 1))
```

`plot.nadir_cv_sl` *Plot a Cross-Validated Super Learner*

Description

Three plot types are provided:

`type = "comparison"` **(default)** A *fair* comparison of the candidate learners against the super learner ensemble. For each learner — candidates and `super_learner` alike — bars and filled points show the mean held-out loss across the *outer* cross-validation folds, open circles show each outer fold's held-out loss, and ranges show ± 1 standard deviation across folds. Learners are ordered best (lowest mean loss) at top. This automates the figure shown in the package README.

`type = "weights"` Ensemble-weight stability across the outer folds; delegated to `plot.nadir_crossfit_sl(x$crossfit, type = "weights")`.

`type = "fitted"` Out-of-fold ensemble predictions against observed outcomes; delegated to `plot.nadir_crossfit_sl(x$ type = "fitted")`. Not defined for `outcome_type = "density"` or `"multiclass"`.

Usage

```
## S3 method for class 'nadir_cv_sl'
plot(x, type = c("comparison", "weights", "fitted"), ...)

## S3 method for class 'nadir_cv_sl'
print(x, ...)
```

Arguments

x	An object of class nadir_cv_sl as returned by cv_super_learner().
type	One of "comparison", "weights", or "fitted".
...	Ignored; included for compatibility with the generic.

Details

Why is this the fair comparison, rather than plot(super_learner(...))? A super_learner() fit's ensemble weights are chosen *using* its candidates' holdout predictions, so scoring the ensemble on those same holdouts is optimistically biased — which is why plot.nadir_sl_model deliberately omits the ensemble from its comparison. cv_super_learner() adds the outer layer of cross-validation needed to score the ensemble honestly, and this plot method scores every candidate on those *same* outer validation folds: within each outer fold, the candidate fits and the ensemble weights were all estimated entirely without the validation data they are evaluated on, and every learner is trained on the same outer training data. All quantities are derived from the fitted object (the per-fold super_learner() fits stored in $crossfit); nothing is re-fit, though each candidate is re-*predicted* once per outer fold.

The loss is the loss_metric stored at fit time (by default, mean squared error for continuous outcomes and negative log loss otherwise). Learners that errored on an outer training fold contribute no loss for that fold.

Requires the **ggplot2** package (listed in Suggests).

Value

A ggplot object, which prints when returned to the console.

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  cv_sl <- cv_super_learner(
    data = mtcars,
    formula = mpg ~ cyl + hp,
    n_folds = 3, inner_n_folds = 2,
    learners = list(mean = lnr_mean, lm = lnr_lm)
  )
  plot(cv_sl) # fair candidates-vs-ensemble comparison
  plot(cv_sl, type = "weights") # weight stability across outer folds
  plot(cv_sl, type = "fitted") # out-of-fold predictions vs. observed
}
```

---

plot.nadir_sl_model *Plot a* nadir_sl_model

Description

Two plot types are provided:

`type = "comparison"` **(default)** For each candidate learner and for the weighted super learner ensemble: bars show the mean cross-validated held-out loss, open circles show each fold's held-out loss, and point-ranges show ± 1 standard deviation across folds. This automates the figure shown in the package README, with the ensemble added as its own row. Learners are ordered best (lowest loss) at top.

`type = "fitted"` Cross-validated ensemble predictions on the held-out folds against the observed outcomes, with the identity line for reference. Each prediction comes from learners trained without that observation. Not defined for `outcome_type = "density"` or `"multiclass"`.

The loss is chosen by `outcome_type`: mean squared error for continuous outcomes and negative log loss otherwise.

Requires the **ggplot2** package (listed in `Suggests`).

Usage

```
## S3 method for class 'nadir_sl_model'
plot(x, type = c("comparison", "fitted"), ...)
```

Arguments

`x`	An object of class `nadir_sl_model` as returned by `super_learner()`.
`type`	One of `"comparison"` or `"fitted"`.
`...`	Ignored; included for compatibility with the generic.

Value

A `ggplot` object, which prints when returned to the console.

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  sl_model <- super_learner(
    data = mtcars,
    formula = mpg ~ cyl + hp,
    learners = list(mean = lnr_mean, lm = lnr_lm)
  )
  plot(sl_model) # learner comparison
  plot(sl_model, type = "fitted") # observed vs. CV ensemble predictions
}
```

screener_cor — *Correlation Threshold Based Screening*

Description

Correlation Threshold Based Screening

Usage

```
screener_cor(data, formula, threshold = 0.2, cor... = NULL)
```

Arguments

`data`	A dataframe intended to be used with `super_learner()`
`formula`	The formula specifying the regression to be done
`threshold`	The correlation coefficient cutoff, below which variables are screened out from the dataset and regression formula.
`cor...`	An optional list of extra arguments to pass to `cor`. Use `method = 'spearman'` for the Spearman rank based correlation coefficient.

Details

If a variable used has little correlation with the outcome being predicted, we might want to screen that variable out from the predictors.

In large datasets, this is quite important, as having a huge number of columns could be computationally intractable or frustratingly time-consuming to run `super_learner()` with.

Value

A list of `$data` with columns screened out, `$formula` with variables screened out, and `$failed_to_correlate_names` the names of variables that failed to correlate with the outcome at least at the threshold level.

Examples

```
screener_cor(
  data = mtcars,
  formula = mpg ~ .,
  threshold = 0.5
)

# We're also showing how to specify that you want the Spearman rank-based
# correlation coefficient, to get away from the assumption of linearity.

screener_cor(
  data = mtcars,
  formula = mpg ~ .,
  threshold = 0.5,
  cor... = list(method = "spearman")
)
```

screener_cor_top_n *Correlation Threshold Based Screening*

Description

Correlation Threshold Based Screening

Usage

```
screener_cor_top_n(data, formula, keep_n_terms, cor... = NULL)
```

Arguments

data	A dataframe intended to be used with `super_learner()`
formula	The formula specifying the regression to be done
keep_n_terms	Set to an integer value >=1, this indicates that the top n terms in the model frame with greatest absolute correlation with the outcome will be kept.
cor...	An optional list of extra arguments to pass to `cor`. Use `method = 'spearman'` for the Spearman rank based correlation coefficient.

Details

If a variable used has little correlation with the outcome being predicted, we might want to screen that variable out from the predictors.

In large datasets, this is quite important, as having a huge number of columns could be computationally intractable or frustratingly time-consuming to run `super_learner()` with.

Value

A list of `$data` with columns screened out, `$formula` with variables screened out, and `$failed_to_correlate_names` the names of variables that failed to correlate with the outcome at least at the threshold level.

Examples

```
screener_cor_top_n(
  data = mtcars,
  formula = mpg ~ .,
  keep_n_terms = 5
)

# We're also showing how to specify that you want the Spearman rank-based
# correlation coefficient, to get away from the assumption of linearity.

screener_cor_top_n(
  data = mtcars,
  formula = mpg ~ .,
  keep_n_terms = 5,
  cor... = list(method = "spearman")
)
```

screener_t_test — *t-test Based Screening: Thresholds on p.values and/or t statistics*

Description

Screens out variables from the formula and dataset based on a p.value and/or the absolute value of the t statistic from a univariate linear regression (with intercept and one term) comparing each predictor to the outcome (dependent) variable.

Usage

```
screener_t_test(
  data,
  formula,
  p_value_threshold = NULL,
  t_statistic_threshold = NULL
)
```

Arguments

`data` a dataset with variables mentioned in the `formula`

`formula` a `formula` with terms from `data`, intended to be used with a learner from `nadir`.

`p_value_threshold` A numeric scalar where terms pass if the t test for the linear model coefficient has p value lower than or equal to the `p_value_threshold` given.

`t_statistic_threshold` A numeric scalar where terms pass if they have a t test statistic greater than or equal to the `t_statistic_threshold` given.

Details

The intended use of `screener_t_test` and other screeners is for pragmatic purposes: when there are a very large number of candidate predictors, such that `super_learner` is very slow to run, predictor variables that fail to have a detectable association with the dependent variable of a formula should be dropped from the learner.

Value

A list of `$data` with columns screened out, `$formula` with variables screened out, and `$failed_to_pass_threshold` the names of variables that failed to associate with the outcome at least at the threshold level.

See Also

screeners, add_screener, screener_cor_top_n

Examples

```
# screen mtcars predictors of mpg, keeping those with t-test p < 0.05
screened <- screener_t_test(mtcars, mpg ~ ., p_value_threshold = 0.05)
screened$formula                    # the screened formula
screened$failed_to_pass_threshold  # predictors that were dropped

# screeners compose with super_learner(): fit on the screened output

sl <- super_learner(
  data = screened$data,
  formulas = screened$formula,
  learners = list(lm = lnr_lm, mean = lnr_mean),
  n_folds = 2
)
sl$learner_weights
```

`screeners` *Wrapping Learners with a Screener*

Description

Screeners work off of the principle that they should take the same arguments that a learner does and return a modified dataset and formula in which variables that have failed to meet some threshold have been screened out.

Details

A screener can be added to a learner by using the `add_screener(learner, screener)` function provided. This returns a modified learner that implements screening based on the data and formula passed.

So far, the screeners implemented rely on being able to call `model.matrix` and therefore only support standard (generalized) linear model syntax like those mentioned in `?formula`.

Value

Screener functions have the same structure: call them with `(data, formula, ...)` and optionally any threshold arguments, then they return a list with `$formula` (the screened formula, retaining only predictors that passed), `$data` (the correspondingly screened data), and, when any predictors were screened out, `$failed_to_pass_threshold` (their names).

See Also

screener_cor, screener_cor_top_n, screener_t_test, add_screener

Examples

```
# examples for setting up a screened regression problem:
#
# users can just run a screener to see what data and formula terms pass the
# given screener conditions:

screened_regression_problem <- screener_cor(
  data = mtcars,
  formula = mpg ~ ., threshold = 0.5
)
screened_regression_problem

screened_regression_problem2 <- screener_cor(
  data = mtcars,
  formula = mpg ~ ., threshold = 0.5, cor... = list(method = "spearman")
)
screened_regression_problem2

screened_regression_problem3 <- screener_t_test(
  data = mtcars,
  formula = mpg ~ ., t_statistic_threshold = 10
)
screened_regression_problem3

# build a new learner with screening builtin:
lnr_rf_screener_top_5_cor_terms <- add_screener(
  learner = lnr_rf,
  screener = screener_cor_top_n,
  screener_extra_args = list(
    cor... = list(method = "spearman"),
    keep_n_terms = 5
  )
)

# train learner
trained_learner <- lnr_rf_screener_top_5_cor_terms(data = mtcars, formula = mpg ~ .)
mtcars_modified <- mtcars
mtcars_modified["gear"] <- 1 # gear is one of the least correlated variables with mpg
identical(trained_learner(mtcars), trained_learner(mtcars_modified))
```

---

super_learner *Super Learner: Cross-Validation Based Ensemble Learning*

Description

Super learning with functional programming!

Usage

```
super_learner(
  data,
  learners,
  formulas,
  y_variable = NULL,
  n_folds = 5,
  determine_super_learner_weights = NULL,
  ensemble_or_discrete = c("ensemble", "discrete"),
  cv_schema,
  outcome_type = c("continuous", "binary", "density", "multiclass"),
  extra_learner_args = NULL,
  cluster_ids = NULL,
  strata_ids = NULL,
  weights = NULL,
  rowids = NULL,
  use_complete_cases = FALSE,
  train_on_whole_dataset = TRUE
)
```

Arguments

`data` Data to use in training a `super_learner`.

`learners` A list of predictor/closure-returning-functions. See Details.

`formulas` Either a single regression formula or a vector of regression formulas.

`y_variable` Typically `y_variable` can be inferred automatically from the `formulas`, but if needed, the y_variable can be specified explicitly.

`n_folds` The number of cross-validation folds to use in constructing the `super_learner`. Must be a scalar integer of value >= 2.

`determine_super_learner_weights` A function/method to determine the weights for each of the candidate `learners`. The default is to use `determine_super_learner_weights_nnls`.

`ensemble_or_discrete` Defaults to `'ensemble'`, but can be set to `'discrete'`. Discrete `super_learner()` chooses only one of the candidate learners to have weight 1 in the resulting prediction algorithm, while `ensemble` `super_learner()` combines predictions from 1 or more candidate learners, with respective weights adding up to 1.

`cv_schema` A function that takes `data`, `n_folds` and returns a list containing `training_data` and `validation_data`, each of which are lists of `n_folds` data frames.

`outcome_type` One of 'continuous', 'binary', 'multiclass', or 'density'. `outcome_type` is used to infer the correct `determine_super_learner_weights` function if it is not explicitly passed.

`extra_learner_args` A list of equal length to the `learners` with additional arguments to pass to each of the specified learners.

`cluster_ids` (default: null) If specified, clusters will either be entirely assigned to training or validation (not both) in each cross-validation split.

`strata_ids` (default: null) If specified, strata are balanced across training and validation splits so that strata appear in both the training and validation splits.

`weights` If specified, (per observation) weights are used to indicate that risk minimization across models (i.e., the meta-learning step) should be targeted to higher weight observations.

`rowids` (default: null) If specified, rowids are stored for use in out-of-fold prediction (`$oof_predict()`) and can also be used to validate/observe the behavior of the cross-validation folds construction.

`use_complete_cases` (default: FALSE) If the `data` passed have any NA or NaN missing data, restrict the `data` to `data[complete.cases(data),]`.

`train_on_whole_dataset` (default: TRUE) If `TRUE`, after the cross-validation stages complete, each candidate learner is additionally fit on the whole dataset; these whole-dataset fits are combined with the meta-learned weights to form the `$predict()` function returned. If `FALSE`, the whole-dataset fits are skipped entirely, saving roughly `1/n_folds` of the total learner-fitting compute. The resulting model retains all out-of-fold prediction interfaces (`$oof_predictions`, `$oof_predict()`, `$oof_predict_modified()`, `$oof_predict_fold()`) as well as `fitted()`, `residuals()`, `coef()`, `summary()`, `plot()`, and `compare_learners()`, since these depend only on the per-fold fits and held-out predictions – but `$predict()` / `predict.nadir_sl_model()` error informatively and `$fit_learners` is `NULL`. This is intended for workflows in which only out-of-fold predictions, or counterfactual modifications thereof, are needed – e.g. in nuisance estimation for causal inference estimators or simulation studies.

Details

The goal of any super learner is to use cross-validation and a set of candidate learners to 1) evaluate how the learners perform on held out data and 2) to use that evaluation to produce a weighted average (for continuous super learner) or to pick a best learner (for discrete super learner) of the specified candidate learners.

Super learner and its statistically desirable properties have been written about at length, including at least the following references:

- https://biostats.bepress.com/ucbbiostat/paper222/
- https://www.stat.berkeley.edu/users/laan/Class/Class_subpages/BASS_sec1_3.1.pdf

`nadir::super_learner` adopts several user-interface design-perspectives that will be useful to know in understanding what it does and how it works:

- The specification of learners should be *very flexible*, really only constrained by the fact that candidate learners should be designed for the same prediction problem but their details can wildly vary from learner to learner.
- It should be easy to specify a customized or new learner.

nadir::super_learner at its core accepts data, a formula (a single one passed to formulas is fine), and a list of learners.

learners are taken to be lists of functions of the following specification:

- a learner must accept a data and formula argument,
- a learner may accept more arguments, and
- a learner must return a prediction function that accepts newdata and produces a vector of prediction values given newdata.

In essence, a learner is specified to be a function taking (data, formula, ...) and returning a *closure* (see http://adv-r.had.co.nz/Functional-programming.html#closures for an introduction to closures) which is a function accepting newdata returning predictions.

Since many candidate learners will have hyperparameters that should be tuned, like depth of trees in random forests, or the lambda parameter for glmnet, extra arguments can be passed to each learner via the extra_learner_args argument. extra_learner_args should be a list of lists, one list of extra arguments for each learner. If no additional arguments are needed for some learners, but some learners you're using do require additional arguments, you can just put a NULL value into the extra_learner_args. See the examples.

In order to seamlessly support using features implemented by extensions to the formula syntax (like random effects formatted like random intercepts or slopes that use the (age | strata) syntax in lme4 or splines like s(age | strata) in mgcv), we allow for the formulas argument to either be one fixed formula that super_learner will use for all the models, or a vector of formulas, one for each learner specified. Most learners receive the formula directly, but some learners specifically require matrix input. nadir automatically does the conversion of formula and data to matrix input for those learners like lnr_glmnet, lnr_hal, lnr_xgboost and others.

Note that in the examples a mean-squared-error (mse) is calculated on the same training/test set, and this is only useful as a crude diagnostic to see that super_learner is working. A more rigorous performance metric to evaluate super_learner on is the cv-rmse produced by cv_super_learner.

Value

An object of class inheriting from nadir_sl_model. This is an S3 object, with elements including a $predict(newdata) method, and some information about the fit model including y_variable, outcome_type, learner_weights, holdout_predictions and optionally information about any errors or warnings thrown by the learner fitting process. holdout_predictions rows are in cross-validation fold order, while the .sl_rowid column maps each row to its row index in the input data (after any complete-case filtering is applied if use_complete_cases is enabled). If any learners signaled warnings, they are captured (not printed) and returned in $warnings_from_training_cv_stage1, $warnings_from_predicting_cv_stage2, and $warnings_from_training_on_entire_data, each a list of warning conditions named by learner with user-legible $calls; $warning_learners lists the learners that warned. If train_on_whole_dataset = FALSE was used, $fit_learners is NULL and $predict() errors informatively; see that parameter's documentation.

See Also

predict.nadir_sl_model compare_learners

cv_super_learner

Examples

```
# most basic usage:
sl <- super_learner(mtcars, list(lm = lnr_lm, mean = lnr_mean),
                    mpg ~ hp + wt, n_folds = 2)
coef(sl)        # ensemble weights
head(fitted(sl))
head(residuals(sl))
formula(sl)
nobs(sl)
summary(sl)

# here's a slightly more in depth example with fancy formula features.
# note all these packages (glmnet, lme4, randomForest) are highly
# Suggested (among others) with {nadir}.

if (requireNamespace("glmnet", quietly = TRUE) &&
    requireNamespace("lme4", quietly = TRUE) &&
    requireNamespace("randomForest", quietly = TRUE)) {

learners <- list(
  glm = lnr_glm,
  rf = lnr_rf,
  glmnet = lnr_glmnet,
  lmer = lnr_lmer
)

# mtcars example ---
formulas <- c(
  .default = mpg ~ cyl + hp, # first three models use same formula
  lmer = mpg ~ (1 | cyl) + hp # lme4 uses different language features
)

# fit a super_learner
sl_model <- super_learner(
  data = mtcars,
  formula = formulas,
  learners = learners
)

# We recommend taking a look at this object to see what's contained inside it:
sl_model

compare_learners(sl_model)

# iris example ---
sl_model <- super_learner(
  data = iris,
  formula = list(
    .default = Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width,
    lmer = Sepal.Length ~ (Sepal.Width | Species) + Petal.Length
  ),
  learners = learners
```

```
)

# produce super_learner predictions and compare against the individual learners
compare_learners(sl_model)
}
```

`truncate_lnr` *Helper to Truncate a Learner's Predictions*

Description

Take in a learner, and return a learner that produces predictor functions which are truncated to the (min, max) region (boundary inclusive).

Usage

```
truncate_lnr(lnr, min, max)
```

Arguments

`lnr`	a nadir learner
`min`	the numeric minimum scalar value defining the minimum that can be predicted; Could be -Inf
`max`	the numeric maximum scalar value defining the maximum that can be predicted; Can be Inf

Value

A learner that produces predictor functions which trim their own output before returning

Examples

```
lnr_truncated <- truncate_lnr(lnr_glm, min = 20, max = 22)
head(lnr_glm(mtcars, mpg ~ cyl + am + hp)(mtcars))
head(lnr_truncated(mtcars, mpg ~ cyl + am + hp)(mtcars))
```

Index

∗ **binary_learners**
binary_learners, 8
∗ **density_learners**
density_learners, 23
∗ **learners**
learners, 28
∗ **multiclass_learners**
multiclass_learners, 65
∗ **screeners**
screeners, 76

add_screener, 3
add_stratification, 5
as_multi_predictor, 7

binary_learners, 8

coef.nadir_crossfit_sl
(crossfit_sl_methods), 10
coef.nadir_sl_model, *10*
coef.nadir_sl_model
(nadir_sl_model_methods), 66
compare_learners, 9
crossfit_sl_methods, 10
crossfit_super_learner, *11*, 12, *67*
cv_character_and_factors_schema, 16
cv_origami_schema, *13*, 18
cv_random_schema, *13*, 19
cv_super_learner, 20, *71*

density_learners, 23
determine_super_learner_weights_nnls,
24
determine_weights_for_binary_outcomes,
25
determine_weights_using_neg_log_loss,
26
df_to_survival_stacked, 27

fitted.nadir_crossfit_sl
(crossfit_sl_methods), 10
fitted.nadir_sl_model, *66*
fitted.nadir_sl_model
(nadir_sl_model_methods), 66
formula.nadir_crossfit_sl
(crossfit_sl_methods), 10
formula.nadir_sl_model
(nadir_sl_model_methods), 66

learners, 28
list_known_learners, 30
lnr_bart, 30
lnr_cvglmnet, 31
lnr_earth, 32
lnr_gam, 33
lnr_gausspr, 34
lnr_gbm, 35
lnr_glm, 36
lnr_glm_density, 36
lnr_glmer, 37
lnr_glmnet, 38
lnr_glmnet_grid, 39
lnr_hal, 40
lnr_hal_grid, 41
lnr_heteroskedastic_density, 42
lnr_homoskedastic_density, 43
lnr_knn, 45
lnr_knn_binary, 46
lnr_lightgbm, 47
lnr_lm, 48
lnr_lm_density, 49
lnr_lmer, 50
lnr_logistic, 50
lnr_mean, 51
lnr_multinomial_nnet, 52
lnr_multinomial_ranger, 52
lnr_multinomial_vglm, 53
lnr_nnet, 54
lnr_ppr, 55
lnr_ranger, 56
lnr_ranger_binary, 56

lnr_rf, 57
lnr_rf_binary, 58
lnr_rpart, 59
lnr_rpart_binary, 60
lnr_svm, 61
lnr_svm_binary, 62
lnr_xgboost, 63

make_learner_names_unique, 64
multiclass_learners, 65

nadir_sl_model_methods, 66
nadir_supported_types, 68
negative_log_loss, 69
negative_log_loss_for_binary, 69
nobs.nadir_crossfit_sl
(crossfit_sl_methods), 10
nobs.nadir_sl_model
(nadir_sl_model_methods), 66

plot.nadir_crossfit_sl, *70*
plot.nadir_crossfit_sl
(crossfit_sl_methods), 10
plot.nadir_cv_sl, 70
plot.nadir_sl_model, 71, *71*
predict.nadir_crossfit_sl
(crossfit_sl_methods), 10
predict.nadir_sl_model
(nadir_sl_model_methods), 66
print.nadir_cv_sl (plot.nadir_cv_sl), 70
print.nadir_sl_model
(nadir_sl_model_methods), 66
print.summary.nadir_crossfit_sl
(crossfit_sl_methods), 10

residuals.nadir_crossfit_sl
(crossfit_sl_methods), 10
residuals.nadir_sl_model
(nadir_sl_model_methods), 66

screener_cor, 73
screener_cor_top_n, 74
screener_t_test, 75
screeners, 76
summary.nadir_crossfit_sl, *10*
summary.nadir_crossfit_sl
(crossfit_sl_methods), 10
summary.nadir_sl_model
(nadir_sl_model_methods), 66

super_learner, *10*, *12*, *72*, 77

truncate_lnr, 82