Package 'rtemis'

November 3, 2026

Version 1.4.3

Title Schema-Driven Machine Learning and Visualization

Date 2026-11-03

Description Machine learning and visualization package with an 'S7' backend featuring comprehensive type checking and validation, observable runs, and auditable outputs. The functional API provides one-call access to supervised and unsupervised learning through train(), cluster(), and decomp(). All configuration steps are performed using setup functions and validated. A single call to train() can handle preprocessing, decomposition, hyperparameter tuning, model stacking, and testing with nested resampling. Supports 'data.frame', 'data.table', and 'tibble' inputs and parallel execution. Plot methods for trained and resampled supervised models draw interactive figures through 'rtemis.draw'. Includes native implementations of SuperLearner, RuleFit, and Linear Additive Trees, and supports conformal prediction. The package first appeared in E.D. Gennatas (2017) <https://repository.upenn.edu/entities/publication/d81892ea-3087-4b71-a6f5-739c58626d64>.

License BSD_3_clause + file LICENSE

URL https://www.rtemis.org, https://docs.rtemis.org/r/ml, https://docs.rtemis.org/r/ml-api/

BugReports https://github.com/rtemis-org/rtemis/issues

ByteCompile yes

Depends R (>= 4.4.0)

Imports data.table, future, grDevices, graphics, methods, nanoparquet, openssl, parallel, parallelly, rtemis.core (>= 0.4.7), rtemis.draw (>= 0.5.4), stats, S7, utils

Suggests arrow, bit64, car, cluster, DBI, dbscan, duckdb, e1071, earth, farff, fastICA, flexclust, fpc, future.mirai, glmnet, grid, gsubfn, hal9001, haven, hopach, htmltools, jsonlite, jsonvalidate, kernlab, kknn, later, leaps, lightAUC, lightgbm, matrixStats, mclust, mgcv, mice, mirai, missRanger, mori, nanonext, NMF, nnls, partykit, pROC, ranger, readxl, ROCR, rpart, rtemis.llm (>= 0.8.7), Rtsne, seqinr, shapr, spls,

stochtree, survival, tabnet, testthat (>= 3.0.0), tibble, timeDate, torch, uwot, vegan, visNetwork, vroom, withr

Encoding UTF-8

Language en-US

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author E.D. Gennatas [aut, cre, cph] (ORCID: <https://orcid.org/0000-0001-9280-3609>)

Maintainer E.D. Gennatas <gennatas@gmail.com>

Repository CRAN

Date/Publication 2026-11-03 18:40:02 UTC

Contents

rtemis-package . . . 7
ai_review . . . 8
applicable_metrics . . . 10
apply_decomp . . . 10
apply_decomposition . . . 11
apply_preprocessor . . . 12
assign_clusters . . . 13
available_supervised . . . 14
bias_variance . . . 15
calibrate . . . 17
check_data . . . 20
classification_metrics . . . 21
class_imbalance . . . 22
cluster . . . 23
clustering_metrics . . . 24
clusters . . . 25
conformal . . . 26
conformal_metrics . . . 28
data_fingerprint . . . 29
data_profile . . . 30
ddb_collect . . . 31
ddb_data . . . 31
decomp . . . 33
decomposition_metrics . . . 34
decomposition_metric_traits . . . 36
decomposition_traits . . . 37
decomp_metrics . . . 38
describe . . . 39
df_movecolumn . . . 40
df_nunique_perfeat . . . 41

draw_linad . . . 41
dt_describe . . . 44
dt_inspect_types . . . 45
dt_keybin_reshape . . . 46
dt_merge . . . 47
dt_names_by_attr . . . 48
dt_nunique_perfeat . . . 49
dt_pctmatch . . . 50
dt_pctmissing . . . 51
dt_set_autotypes . . . 51
dt_set_cleanfactorlevels . . . 52
dt_set_clean_all . . . 53
dt_set_logical2factor . . . 54
dt_set_one_hot . . . 55
exc . . . 56
explain . . . 57
explanation_methods . . . 59
explanation_traits . . . 59
features . . . 60
feature_matrix . . . 61
feature_names . . . 62
fitted_config . . . 62
fit_predict . . . 63
getnames . . . 64
getnamesandtypes . . . 65
get_factor_names . . . 66
get_fitted_config . . . 67
get_learning_curve . . . 68
get_loadings . . . 69
get_membership . . . 70
get_metrics . . . 71
get_mode . . . 72
get_predictions . . . 73
get_varimp . . . 74
HyperparameterCandidates . . . 75
inc . . . 75
index_col_by_attr . . . 76
ingest . . . 77
init_project_dir . . . 78
inspect . . . 79
inspect_type . . . 79
is_constant . . . 80
is_same_data . . . 81
JSONSchema_to_S7 . . . 82
massGLM . . . 83
matchcases . . . 84
mgetnames . . . 86
mode_value . . . 87

names_by_class . . . 88
numeric_features . . . 88
one_hot2factor . . . 89
outcome . . . 90
outcome_name . . . 91
partition . . . 91
plot_learning . . . 92
plot_manhattan . . . 93
plot_metric . . . 94
plot_roc . . . 95
plot_session . . . 96
plot_true_pred . . . 97
plot_varimp . . . 98
preprocess . . . 99
preprocessed . . . 101
present . . . 101
read . . . 103
read_config . . . 105
read_result . . . 106
reconstruct . . . 107
record . . . 108
regression_metrics . . . 109
resample . . . 109
review . . . 111
rnormmat . . . 113
roc_curve . . . 114
roc_table . . . 115
rtemis_conditions . . . 116
rtversion . . . 117
runifmat . . . 118
S7_dispatcher_JSONSchema . . . 119
S7_to_JSONSchema . . . 121
same_data . . . 123
se . . . 123
session_timeline . . . 124
setdiffsym . . . 125
setup_ARFFIngest . . . 126
setup_ASWCriterion . . . 127
setup_Autoencoder . . . 127
setup_BART . . . 130
setup_Bootstrap . . . 132
setup_CART . . . 133
setup_CHCriterion . . . 134
setup_ClusterConfig . . . 135
setup_CMeans . . . 136
setup_ConditionalSuperLearner . . . 137
setup_CQR . . . 138
setup_CUDA . . . 139

setup_Custom . . . 140
setup_CVPlus . . . 141
setup_DBSCAN . . . 142
setup_DecomposeConfig . . . 143
setup_DelimitedIngest . . . 144
setup_DTAIngest . . . 145
setup_FutureExecution . . . 146
setup_GAM . . . 148
setup_GLM . . . 149
setup_GLMNET . . . 149
setup_GLMTree . . . 150
setup_GMM . . . 153
setup_GridSearch . . . 154
setup_GroupPartition . . . 155
setup_HAL . . . 156
setup_HardCL . . . 158
setup_HOPACH . . . 159
setup_ICA . . . 161
setup_Isomap . . . 162
setup_Isotonic . . . 163
setup_KFold . . . 163
setup_KMeans . . . 164
setup_KNN . . . 165
setup_LightCART . . . 166
setup_LightGBM . . . 170
setup_LightRF . . . 176
setup_LightRuleFit . . . 181
setup_LINAD . . . 187
setup_LINADForest . . . 193
setup_LinearSVM . . . 198
setup_LOOCV . . . 198
setup_MARS . . . 199
setup_MDS . . . 201
setup_MiraiExecution . . . 202
setup_MLP . . . 204
setup_ModalityStacking . . . 207
setup_MonotonicHAL . . . 209
setup_MultiASWCriterion . . . 211
setup_NeuralGas . . . 211
setup_NMF . . . 212
setup_NNLS . . . 213
setup_Nystrom . . . 214
setup_PAM . . . 214
setup_PAMK . . . 216
setup_ParquetIngest . . . 217
setup_PCA . . . 218
setup_PCoA . . . 219
setup_PredefinedPartition . . . 220

setup_Preprocessor . . . 220
setup_RadialSVM . . . 226
setup_RandomPartition . . . 227
setup_Ranger . . . 227
setup_RDSIngest . . . 231
setup_Resampler . . . 232
setup_SerialExecution . . . 233
setup_SHAP . . . 234
setup_SpectralLaplace . . . 236
setup_SpectralLocal . . . 237
setup_SpectralRBF . . . 238
setup_SplitConformal . . . 239
setup_SPLS . . . 241
setup_StratBoot . . . 242
setup_StratSub . . . 243
setup_SuperConfig . . . 244
setup_SuperConfigLive . . . 246
setup_SuperLearner . . . 248
setup_SupervisedPreprocessor . . . 249
setup_TabNet . . . 254
setup_TimePartition . . . 256
setup_tSNE . . . 257
setup_UMAP . . . 259
setup_VariationalAutoencoder . . . 260
setup_XLSXIngest . . . 262
set_outcome . . . 263
set_positive_class . . . 264
shap_by_level . . . 265
shap_case . . . 266
shap_long . . . 267
show_color_key . . . 268
size . . . 268
table_column_attr . . . 269
train . . . 270
transformed . . . 273
tune_over . . . 274
tuning_grid . . . 275
validate_config . . . 276
varimp_ranking . . . 277
varimp_table . . . 278
writeup . . . 279
write_config . . . 281
write_record . . . 282
write_result . . . 283
write_text . . . 284
%BC% . . . 285

Index **287**

rtemis-package **rtemis**: *Advanced Machine Learning and Visualization*

Description

Schema-driven machine learning for validated, observable, auditable, and reproducible machine learning workflows.

Online Documentation and Vignettes

https://docs.rtemis.org/r/ml

Visualization

Model plots (`plot_true_pred()`, `plot_roc()`, `plot_varimp()`, `plot_metric()`, `plot_learning()`, `plot_session()`, `plot_manhattan()`) and `present()` render interactive charts through 'rtemis.draw', which also provides the `draw_*()` functions for data. `draw_linad()` draws a LINAD tree.

Supervised Learning

By convention, the last column of the data is the outcome variable, and all other columns are predictors. Convenience function set_outcome can be used to move a specified column to the end of the data. Regression and Classification is performed using `train()`, which preprocesses, trains, tunes, and tests models on multiple resamples. Use available_supervised to get a list of available algorithms

Classification

For training of binary classification models, the outcome should be provided as a factor, with the *second* level of the factor being the 'positive' class.

Clustering

Clustering is performed using `cluster()`. Use available_clustering to get a list of available algorithms.

Decomposition

Decomposition is performed using `decomp()`. Use available_decomposition to get a list of available algorithms.

Type Documentation

Function documentation includes input type (e.g. "Character", "Integer", "Float"/"Numeric", etc). When applicable, value ranges are provided in interval notation. For example, Float: [0, 1) means floats between 0 and 1 including 0, but excluding 1. Categorical variables may include set of allowed values using curly braces. For example, Character: {"future", "mirai", "none"}.

Tabular Data

rtemis internally uses methods for efficient handling of tabular data, with support for `data.frame`, `data.table`, and `tibble`. If a function is documented as accepting "tabular data", it accepts any of these data structures. A function documented as accepting only one of these accepts only that structure. For example, some optimized `data.table` operations that perform in-place modifications only work with `data.table` objects.

Error Conditions

Errors are signalled as classed conditions that can be handled with `tryCatch()`. See rtemis_conditions for the class hierarchy.

Author(s)

Maintainer: E.D. Gennatas <gennatas@gmail.com> (ORCID) [copyright holder]

Authors:

- E.D. Gennatas <gennatas@gmail.com> (ORCID) [copyright holder]

See Also

Useful links:

- https://www.rtemis.org
- https://docs.rtemis.org/r/ml
- https://docs.rtemis.org/r/ml-api/
- Report bugs at https://github.com/rtemis-org/rtemis/issues

---

ai_review *Write an assessment of a model review with a language model*

Description

Ask a language model to write a summary, an evaluation, next steps and caveats from a review of a trained supervised model. Every statement cites the codes of the review findings it rests on, and the result keeps the review and a record of how the text was produced.

Usage

```
ai_review(x, llm, context = NULL, temperature = NULL, verbosity = 1L, ...)
```

Arguments

x	SupervisedReview, Supervised or SupervisedRes object: A review, or a trained model to review with default settings.
llm	rtemis.llm LLM or Agent object: The model to write the assessment, for example from rtemis.llm::create_Ollama() or rtemis.llm::create_Anthropic().
context	Optional Character: Domain context for the assessment.
temperature	Optional Numeric [0, Inf): Sampling temperature for this call. NULL uses the model's own setting.
verbosity	Integer: Verbosity level.
...	Not used.

Details

The model receives the review as JSON and, if given, context: the question the model addresses, the costs of different errors, how its predictions will be used. It never receives the data. Without context, the assessment states that usefulness cannot be judged.

The model answers in a declared structure whose codes are restricted to the review's findings, and the answer is checked: an answer that does not match the structure, or cites a code that is not a finding of the review, is an error. The returned object records the model, the provider, the temperature, the full prompt, a SHA-256 hash of the review sent and the time, so the assessment can be audited and, as far as the model allows, reproduced.

The written assessment is an interpretation of the review, which remains the evidence; read them together. write_text writes both as Markdown.

Requires the rtemis.llm package and access to a model.

Value

AISupervisedReview object.

Author(s)

EDG

Examples

```
## Not run:
# Requires a running Ollama server with the model pulled.
mod <- train(
  iris,
  hyperparameters = setup_CART(),
  outer_resampling_config = setup_KFold(5L),
  verbosity = 0L
)
llm <- rtemis.llm::create_Ollama(model_name = "gemma4:e4b")
ai_review(
  mod,
  llm = llm,
```

```
  context = "Teaching example; no decisions depend on the predictions."
)

## End(Not run)
```

`applicable_metrics` *applicable_metrics() (Deprecated)*

Description

`applicable_metrics()` is deprecated. Use `decomposition_metric_traits()`, which takes the same arguments.

Usage

```
applicable_metrics(algorithm = NULL)
```

Arguments

`algorithm`	Optional Character: Name of a decomposition algorithm, matched case-insensitively. `NULL` returns the full derived matrix.

Value

As `decomposition_metric_traits()`.

Author(s)

EDG

Examples

```
suppressWarnings(applicable_metrics("UMAP"))
```

`apply_decomp` *Apply Decomposition to New Data (Deprecated)*

Description

`apply_decomp()` is deprecated. Use `apply_decomposition()`, which takes the same arguments.

Usage

```
apply_decomp(decom, new_data, execution_config = NULL, verbosity = 1L)
```

Arguments

decom	Decomposition object.
new_data	Tabular data (data.frame, data.table, or tibble): New data to which the decomposition will be applied.
execution_config	Optional ExecutionConfig object: Threads for an algorithm whose threaded trait is TRUE are its n_workers_algorithm, or its worker count. NULL uses the host's default worker count.
verbosity	Integer: Verbosity level.

Value

As apply_decomposition().

Author(s)

EDG

Examples

```
iris_pca <- decomp(exc(iris, "Species"), algorithm = "PCA", verbosity = 0L)
suppressWarnings(apply_decomp(iris_pca, exc(iris, "Species")))
```

---

apply_decomposition *Apply a Decomposition to New Data*

Description

Apply a fitted decomposition to new data for algorithms that support this.

Usage

```
apply_decomposition(decom, new_data, execution_config = NULL, verbosity = 1L)
```

Arguments

decom	Decomposition object.
new_data	Tabular data (data.frame, data.table, or tibble): New data to which the decomposition will be applied.
execution_config	Optional ExecutionConfig object: Threads for an algorithm whose threaded trait is TRUE are its n_workers_algorithm, or its worker count. NULL uses the host's default worker count.
verbosity	Integer: Verbosity level.

Details

A fit is learned on the columns `decom@config@features` names, so only those columns of `new_data` are transformed; the remaining columns are returned unchanged, alongside the learned components, in the layout `[kept features, components]`. `decomp()` resolves an unset `features` to the numeric columns and records them, so a fit it produced always names them. A `Decomposition` assembled by hand with no `features` transforms every column of `new_data` and returns only the components.

Available for algorithms whose `can_apply` trait is TRUE; see decomposition_traits. `transformed()` returns the components of the data the decomposition was fitted on.

Value

A data.frame of the learned components for `new_data`, preceded by any feature columns that were not decomposed.

Author(s)

EDG

Examples

```
iris_pca <- decomp(exc(iris, "Species"), algorithm = "PCA")
apply_decomposition(iris_pca, exc(iris, "Species"))
```

`apply_preprocessor` *Apply trained Preprocessor to new data*

Description

Apply a trained `Preprocessor` to new data, reusing the values learned from the training data. For example, the same mean and mode imputation values, scale centers and coefficients, one-hot levels, and removed features will be applied to the new data.

Usage

```
apply_preprocessor(preprocessor, new_data, verbosity = 1L)
```

Arguments

`preprocessor`	`Preprocessor`: Trained preprocessor, i.e. the output of preprocess.
`new_data`	Tabular data, i.e. data.frame, data.table, or tbl_df (tibble): New data to preprocess.
`verbosity`	Integer: Verbosity level.

Details

Imputation follows the preprocessor's `impute_type`. "meanMode" imputes the stored training values. "missRanger" applies the forests fitted on the training data with `predict()`, and imputes a feature that had no missing training values with its stored mean or mode; `predict()` stops on a factor level of an imputed feature that is absent from the training data. A preprocessor using "micePMM" cannot be applied to new data and is refused.

Value

Preprocessed data, in the same structure as `new_data`.

Author(s)

EDG

See Also

preprocess, setup_Preprocessor

Examples

```
res <- resample(iris, setup_KFold(seed = 2026))
iris_train <- iris[res[[1]], ]
iris_test <- iris[-res[[1]], ]

# Preprocess training data
iris_pre <- preprocess(iris_train, setup_Preprocessor(scale = TRUE, center = TRUE))

# Apply the same preprocessing to test data
iris_test_pre <- apply_preprocessor(iris_pre, iris_test)
```

`assign_clusters` *Assign New Cases to Fitted Clusters*

Description

Assigns each case of `newdata` to one of the clusters of a fitted clustering, by the algorithm's own assignment rule.

Usage

```
assign_clusters(clust, newdata, verbosity = 1L)
```

Arguments

`clust`	`Clustering` object.
`newdata`	Tabular data (data.frame, data.table, or tibble): Cases to assign.
`verbosity`	Integer: Verbosity level.

Details

The clustering was fitted on the columns `clust@config@features` names, so only those columns of `newdata` are read; `newdata` must hold every one of them, numeric and free of missing values. KMeans, HardCL, and NeuralGas assign a case to the cluster with the closest centroid under the distance the fit used. An algorithm with no rule for cases it was not fitted on raises an `rtemis_unsupported_error` naming the algorithms that have one. `clusters()` returns the assignments of the data the clustering was fitted on.

Value

Integer vector: One cluster label per row of `newdata`.

Author(s)

EDG

Examples

```
x <- exc(iris, "Species")
iris_km <- cluster(x[1:100, ], algorithm = "KMeans", verbosity = 0L)
assign_clusters(iris_km, x[101:150, ], verbosity = 0L)
```

`available_supervised` *Available Algorithms*

Description

Print available algorithms for supervised learning, clustering, and decomposition.

Usage

```
available_supervised(verbosity = 1L)

available_clustering(verbosity = 1L)

available_calibration(verbosity = 1L)

available_decomposition(verbosity = 1L, traits = FALSE)
```

Arguments

`verbosity`	Integer: Verbosity level.
`traits`	Logical: If TRUE, `available_decomposition()` prints and returns the full trait table from decomposition_traits instead of the named list of descriptions.

Details

Each algorithm is set up with `setup_{Algorithm}()`, using the name printed here: `setup_LightGBM()`, `setup_KMeans()`, `setup_PCA()`. Pass the result to train, cluster, or decomp.

Value

Named list of algorithm descriptions, invisibly.

Author(s)

EDG

Examples

```
available_supervised()
# Train with one of them, at its default hyperparameters:
# train(iris, hyperparameters = setup_LightGBM())
available_clustering()
available_calibration()
# Calibrate with one of them:
# calibrate(mod, hyperparameters = setup_Isotonic())
available_decomposition()
available_decomposition(traits = TRUE)
```

---

bias_variance *Estimate Bias and Variance*

Description

Decompose a learner's error into the part that comes from being systematically wrong and the part that comes from being unstable, by refitting it over many resamples of the training set and watching what its predictions do at a **fixed** test set.

Usage

```
bias_variance(
  x,
  hyperparameters,
  dat_test = NULL,
  true_values = NULL,
  resampler_config = NULL,
  test_p = 0.3,
  execution_config = setup_FutureExecution(),
  verbosity = 1L
)
```

Arguments

`x`	tabular data: Full dataset, outcome in the last column. Resamples are drawn from it after the test set is held out.
`hyperparameters`	`Hyperparameters` object: Must be fixed, not a search space – this measures one model, not a tuning run.
`dat_test`	Optional tabular data: Test set. NULL holds one out of `x`.
`true_values`	Optional Numeric vector: The true function at the test cases, in their order. Regression: `f(x)`. Classification: the probability of the positive class. NULL measures bias against the observed outcome, which includes irreducible noise.
`resampler_config`	Optional `ResamplerConfig` object: How training sets are drawn. NULL uses 100 stratified bootstraps.
`test_p`	Numeric (0, 1): Fraction of `x` held out as the test set. Ignored when `dat_test` is given.
`execution_config`	`ExecutionConfig` object: Supplies the seed and the workers the resamples are fitted over.
`verbosity`	Integer: If > 0, print messages.

Details

For squared-error loss, the expected error at a point decomposes as

$$E[(y - \hat{f}(x))^2] = \sigma^2 + (E[\hat{f}(x)] - f(x))^2 + Var[\hat{f}(x)]$$

– irreducible noise, squared bias, and variance. This function estimates the second and third per test case, by fitting the model on `n_resamples` resamples of the training set and reading the spread and the average of the predictions each makes at the same test cases.

The test set is held out **once** and every resample is drawn from what is left, which is what makes the predictions comparable across resamples: a test set that moved would confound the model's variability with the sample's.

Value

`BiasVariance` object.

Bias against true values and against observed outcomes

Squared bias is measured against `true_values` when supplied, and against the observed outcome otherwise:

With `true_values` `bias_squared` is the bias, exactly. Only simulated data has true values, so a simulation study measures how a learner divides its error.

Without The observed outcome carries the irreducible noise, so what is reported is $bias^2 + \sigma^2$ and no amount of resampling separates them. It stays comparable **between learners on the same data** – they share the same σ^2 – so a ranking is still meaningful even though the level is inflated.

What the numbers are, exactly

`variance` is the sample variance across resamples (dividing by `n_resamples - 1`), which is unbiased for the variance of the fitted function. `bias_squared` is the plug-in estimate, and is inflated by roughly `variance / n_resamples` because the mean prediction is itself estimated – so prefer many resamples over few, and treat a `bias_squared` of the same order as `variance / n_resamples` as indistinguishable from zero.

Classification

The decomposition above is for squared error, and 0-1 loss does not decompose additively into bias and variance. For a binary outcome this function therefore decomposes the squared error **of the predicted probability** of the positive class, which does. `true_values` is then the true probability, not the class. Multiclass is not supported.

Author(s)

EDG

References

Geman S, Bienenstock E, Doursat R (1992). Neural Networks and the Bias/Variance Dilemma. *Neural Computation*, 4(1), 1-58. doi:10.1162/neco.1992.4.1.1

Examples

```
set.seed(2026)
n <- 200L
features <- data.frame(a = rnorm(n), b = rnorm(n))
dat <- data.frame(features, y = 2 * features[["a"]] + rnorm(n))
bv <- bias_variance(
  dat,
  hyperparameters = setup_CART(),
  resampler_config = setup_StratBoot(n_resamples = 10L),
  verbosity = 0L
)
bv
```

---

calibrate *Calibrate* `Classification` & `ClassificationRes` *Models*

Description

Generic function to calibrate binary classification models.

Usage

```
calibrate(
  x,
  hyperparameters = NULL,
  execution_config = NULL,
  verbosity = 1L,
  ...
)
```

Arguments

`x`	`Classification` or `ClassificationRes` object to calibrate.
`hyperparameters`	Optional `Hyperparameters` object: Setup using one of `setup_*` functions. Defines the algorithm used to train the calibration model. NULL uses setup_Isotonic.
`execution_config`	Optional `ExecutionConfig` object: Setup using setup_SerialExecution, setup_FutureExecution or setup_MiraiExecution. The calibration models are trained under it. NULL trains them serially.
`verbosity`	Integer: Verbosity level.
`...`	Additional arguments passed to specific methods.

Details

The goal of calibration is to adjust the predicted probabilities of a binary classification model so that they better reflect the true probabilities (i.e. empirical risk) of the positive class.

Value

Calibrated model object.

Method-specific parameters

For `Classification` objects:

- `predicted_probabilities`: Numeric vector of the positive class's predicted probabilities, one per case
- `true_labels`: Factor of true class labels

For `ClassificationRes` objects:

- `resampler_config`: `ResamplerConfig` object for calibration training
- `train_verbosity`: Integer controlling calibration model training output

Choosing a calibrator

A calibration map must be monotonic non-decreasing. A map that reorders scores changes the ranking of the predictions, and so changes AUC; a non-decreasing one cannot. available_calibration lists the algorithms that carry that guarantee. Any other `Hyperparameters` object is accepted and trained like any other model, and the map is then unconstrained.

- setup_Isotonic (the default) fits isotonic regression. The map is a step function, so it merges nearby scores into ties, which moves AUC slightly in either direction. Fitted probabilities are held at least `1 / (2 * n)` away from 0 and 1, `n` being the number of calibration cases, so a block of uniformly labelled cases does not assert certainty.
- setup_MonotonicHAL fits a monotonic Highly Adaptive Lasso on the logit scale. At its default `smoothness_orders = 1` the map is continuous and strictly increasing, so it preserves AUC exactly – but the constraint that achieves this also makes the map convex on the logit scale, which costs accuracy whenever the correction needed is concave. `smoothness_orders = 0` with `penalized = FALSE` lifts that restriction, at the cost of ties and a markedly slower fit.

In rtemis benchmarks, no monotonic lasso configuration was uniformly more accurate than isotonic regression, and each was several times slower. `data-raw/benchmark_calibrators.R` reproduces the comparison.

The calibrator that ran is recorded on the returned object's `@calibrator` property, is shown by `print()`, and is serialized by `to_json()`.

Author(s)

EDG

Examples

```
# --- Calibrate Classification ---
dat <- iris[51:150, ]
res <- resample(dat)
dat[["Species"]] <- factor(dat[["Species"]])
dat_train <- dat[res[[1]], ]
dat_test <- dat[-res[[1]], ]

# Train GLM on a training/test split
mod_c_glm <- train(
  x = dat_train,
  dat_test = dat_test,
  hyperparameters = setup_GLM()
)

# Calibrate the `Classification` by defining `predicted_probabilities` and `true_labels`,
# in this case using the training data, but it could be a separate calibration dataset.
mod_c_glm_cal <- calibrate(
  mod_c_glm,
  predicted_probabilities = mod_c_glm[["predicted_prob_training"]][, 1L],
  true_labels = mod_c_glm[["y_training"]]
)
mod_c_glm_cal
```

```
# --- Calibrate ClassificationRes ---

# Train GLM with cross-validation
resmod_c_glm <- train(
  x = dat,
  hyperparameters = setup_GLM(),
  outer_resampling_config = setup_KFold(n_resamples = 3L)
)

# Calibrate the `ClassificationRes` using the same resampling configuration as used for training.
resmod_c_glm_cal <- calibrate(resmod_c_glm)
resmod_c_glm_cal
```

`check_data` *Check Data*

Description

Check Data

Usage

```
check_data(
  x,
  name = NULL,
  get_duplicates = TRUE,
  get_na_case_pct = FALSE,
  get_na_feature_pct = FALSE
)
```

Arguments

`x`	tabular data: Input to be checked.
`name`	Character: Name of dataset.
`get_duplicates`	Logical: If TRUE, count duplicate cases. If FALSE, `n_duplicates` is `NULL` and nothing reports on duplicates.
`get_na_case_pct`	Logical: If TRUE, calculate percent of NA values per case.
`get_na_feature_pct`	Logical: If TRUE, calculate percent of NA values per feature.

Value

`CheckData` object.

Author(s)

EDG

Examples

```
n <- 1000
x <- rnormmat(n, 50, return_df = TRUE)
x[["char1"]] <- sample(letters, n, TRUE)
x[["char2"]] <- sample(letters, n, TRUE)
x[["fct"]] <- factor(sample(letters, n, TRUE))
x <- rbind(x, x[1, ])
x[["const"]] <- 99L
x[sample(nrow(x), 20), 3] <- NA
x[sample(nrow(x), 20), 10] <- NA
x[["fct"]][30:35] <- NA
check_data(x)
```

---

classification_metrics

Classification Metrics

Description

Classification Metrics

Usage

```
classification_metrics(
  true_labels,
  predicted_labels,
  predicted_prob = NULL,
  binclasspos = 2L,
  calc_auc = TRUE,
  calc_brier = TRUE,
  auc_method = "lightAUC",
  sample = NULL,
  verbosity = 0L
)
```

Arguments

true_labels	Factor: True labels.
predicted_labels	Factor: predicted values.
predicted_prob	Optional Numeric vector or matrix [0, 1]: Predicted probabilities, one column per class. A single column is the positive class's probability and enables the probability-based metrics (AUC, Brier score); a multiclass matrix has no single score per case, so those are not computed.
binclasspos	Integer: Factor level position of the positive class in binary classification.
calc_auc	Logical: If TRUE, calculate AUC. May be slow in very large datasets.

calc_brier	Logical: If TRUE, calculate Brier score.
auc_method	Character: "lightAUC", "pROC", "ROCR".
sample	Optional Character {"Training", "Validation", "Test", "Calibrated Training", "Calibrated Validation", "Calibrated Test"}: Sample the metrics describe.
verbosity	Integer: Verbosity level.

Details

auc_method = "pROC" is the only method that outputs an AUC when one or more predicted probabilities are NA.

Value

ClassificationMetrics object.

Author(s)

EDG

Examples

```
# Assume positive class is "b"
true_labels <- factor(c("a", "a", "a", "b", "b", "b", "b", "b", "b", "b"))
predicted_labels <- factor(c("a", "b", "a", "b", "b", "a", "b", "b", "b", "a"))
predicted_prob <- c(0.3, 0.55, 0.45, 0.75, 0.57, 0.3, 0.8, 0.63, 0.62, 0.39)

classification_metrics(true_labels, predicted_labels, predicted_prob)
classification_metrics(true_labels, predicted_labels, 1 - predicted_prob, binclasspos = 1L)
```

---

class_imbalance *Class Imbalance*

Description

Calculate class imbalance as given by:

$$I = K \cdot \sum_{i=1}^{K} (n_i/N - 1/K)^2$$

where K is the number of classes, and n_i is the number of instances of class i

Usage

```
class_imbalance(x)
```

Arguments

x	Vector, factor: Outcome.

Value

Numeric.

Author(s)

EDG

Examples

```
# iris is perfectly balanced
class_imbalance(iris[["Species"]])
# Simulate imbalanced outcome
x <- factor(sample(c("A", "B"), size = 500L, replace = TRUE, prob = c(0.9, 0.1)))
class_imbalance(x)
```

`cluster` *Perform Clustering*

Description

Perform clustering on the rows (usually cases) of a dataset.

Usage

```
cluster(
  x,
  algorithm = "KMeans",
  config = NULL,
  execution_config = setup_FutureExecution(),
  outdir = NULL,
  verbosity = 1L
)
```

Arguments

`x`	Matrix, data.frame, or `ClusterConfig` object: Data to cluster (rows are cases to be clustered), or a `ClusterConfig` recipe (from setup_ClusterConfig) carrying the data path, algorithm config, and output directory.
`algorithm`	Character: Clustering algorithm. Not needed when `config` is supplied, which names its own; an explicit `algorithm` that disagrees with `config` is an error.
`config`	`ClusteringConfig`, optional: Algorithm-specific config from a clustering `setup_*` function. Its `features` selects the columns to cluster on; `NULL` selects every numeric column of `x`, since a clustering backend reads numbers. The returned object's config carries the resolved names.

`execution_config`
: `ExecutionConfig` object: Execution settings, e.g. setup_FutureExecution or setup_SerialExecution. A clustering dispatches no work to other processes and no registered algorithm threads, so the fit runs on one thread; the config's `seed` seeds it.

`outdir`
: Character, optional: Output directory. If not NULL, the returned `Clustering` object is saved there as an `.rds` file, alongside a run record (`cluster_<algorithm>.record.json`) stating what the run resolved. See write_record.

`verbosity`
: Integer: Verbosity level.

Details

See docs.rtemis.org/r for detailed documentation.

Value

`Clustering` object.

Author(s)

EDG

Examples

```
iris_km <- cluster(exc(iris, "Species"), algorithm = "KMeans")
```

---

clustering_metrics *Clustering Metrics*

Description

Quantify a fitted clustering on the data it was fitted on: how many cases and clusters, the fraction labeled noise, how concentrated the memberships of a soft clustering are, and how well separated its clusters are.

Usage

```
clustering_metrics(clust, x, verbosity = 1L)
```

Arguments

`clust`
: `Clustering` object.

`x`
: Tabular data (data.frame, data.table, or tibble): The data `clust` was fitted on.

`verbosity`
: Integer: Verbosity level.

Details

`cluster()` stores the measures that cost no more than one pass over the assignments on its result, where `get_metrics()` returns them, and reports the mean silhouette as not requested. This function computes the same measures and the mean silhouette, whose cost is quadratic in the number of cases. The silhouette uses Euclidean distances over the clustered features, `clust@config@features`, and excludes cases labeled noise (label 0).

Each measure carries a status: `computed`, `unsupported` (the clustering does not have what the measure needs, such as a membership matrix), or `not_applicable` (the silhouette when fewer than two clusters hold a case).

`x` must be the data `clust` was fitted on, with one row per case, since the measures pair each row with its fitted assignment. Data whose fingerprint differs from the one `cluster()` recorded raises an error.

Value

`ClusteringMetrics` object.

Author(s)

EDG

Examples

```
x <- exc(iris, "Species")
iris_km <- cluster(x, algorithm = "KMeans", verbosity = 0L)
clustering_metrics(iris_km, x, verbosity = 0L)
```

---

clusters *Get cluster assignments from* `Clustering`

Description

Returns the cluster each case of the data a `Clustering` was fitted on is assigned to.

Usage

```
clusters(x)
```

Arguments

`x`	`Clustering`: A `Clustering` object.

Details

Cluster labels are integers from 1 to the number of clusters fitted. An algorithm with a noise label, such as DBSCAN, labels noise cases 0. A soft clustering assigns each case to the cluster of its highest membership; `get_membership()` returns the membership matrix.

Value

Integer vector: One cluster label per case.

Author(s)

EDG

Examples

```
iris_km <- cluster(exc(iris, "Species"), algorithm = "KMeans", verbosity = 0L)
clusters(iris_km)
```

`conformal` *Conformal prediction regions*

Description

A prediction interval for a regression or a set of labels for a classification, covering the truth with probability at least `1 - alpha` under exchangeability alone – in finite samples, for any model, including a misspecified one.

Usage

```
conformal(x, newdata, calibration = NULL, config = NULL, verbosity = 1L, ...)
```

Arguments

`x`	`Supervised` or `SupervisedRes` object.
`newdata`	tabular data: Cases to bound. Predictors only, in training order, as `stats::predict()` requires.
`calibration`	Optional tabular data: Calibration cases, predictors and outcome, in the shape `train()` was given. NULL uses the model's stored test split where it has one.
`config`	Optional `ConformalConfig` object: Built by setup_SplitConformal, setup_CVPlus or setup_CQR. NULL takes the method the object supports.
`verbosity`	Integer: Verbosity level.
`...`	Additional arguments passed to methods.

Details

The guarantee is marginal. `P(Y in C(X)) >= 1 - alpha` averages over the draw of both the calibration set and the test case. It is *not* conditional: 90% marginal coverage is compatible with 99% coverage on easy cases and 60% on a hard subgroup. The guarantee rests on exchangeability and does not hold under distribution shift.

Width is the quality measure; coverage is the correctness check. A model with no predictive value attains valid coverage with intervals wide enough to be uninformative, so the two are read together. The region reports its widths (or set sizes); `conformal_metrics()` scores coverage against outcomes the region did not see.

Which data calibrates. `calibration` is a tabular dataset holding the predictors and the outcome, in training shape. Left NULL, a model trained with `dat_test` calibrates on that split, whose residuals it already stores: `train()` never fits, tunes or early-stops on `dat_test`. Nothing else is assumed clean – a validation split may have driven early stopping, and a model chosen by its test metric has used that split for selection, which nothing in the object records.

Conformal calibration is unrelated to the probability calibration calibrate performs, and the two must not share rows. rtemis raises an error where it detects shared rows.

Which method. `config` NULL takes the method the object supports: split conformal for a `Supervised`, CV+ for a `SupervisedRes`. See setup_SplitConformal, setup_CVPlus and setup_CQR.

Value

`PredictionInterval` object for a regression, `PredictionSet` for a classification.

Author(s)

EDG

Examples

```
x <- data.frame(age = rnorm(300), bmi = rnorm(300))
x[["y"]] <- x[["age"]] * 2 + rnorm(300, sd = 0.3)
# Three splits: one fits, one calibrates, one scores what the other two did.
mod <- train(
  x[1:200, ],
  dat_test = x[201:250, ],
  hyperparameters = setup_GLM(),
  verbosity = 0L
)

# The stored test split calibrates; 90% intervals for five fresh cases.
region <- conformal(mod, x[1:5, c("age", "bmi")], verbosity = 0L)
region
region@lower
region@upper

# Coverage, against outcomes nothing in the pipeline has seen.
held_out <- x[251:300, ]
conformal_metrics(
  conformal(mod, held_out[, c("age", "bmi")], verbosity = 0L),
  held_out[["y"]]
)
```

`conformal_metrics` *Score a Conformal Prediction Region*

Description

Empirical coverage and region size for a `PredictionRegion`, against the outcomes of the cases it bounds.

Usage

```
conformal_metrics(region, true_outcome)
```

Arguments

`region`	`PredictionRegion` object, from conformal.
`true_outcome`	Numeric or Factor: The outcomes of the cases `region` bounds, in the same order.

Details

Coverage on the calibration data is not a check. The threshold is chosen as the `1 - alpha` empirical quantile of the calibration scores, so coverage measured back on those same cases is `1 - alpha` by construction. Coverage is scored against outcomes nothing in the pipeline has seen.

Read coverage and size together. A model with no signal attains valid coverage by returning uninformatively wide intervals, so coverage alone does not distinguish an accurate model from a wide one. Width – or set size, and the rate of singleton and empty sets – is the part that carries information about the model.

Coverage is a proportion over the cases supplied, so it carries the Monte Carlo error of that sample: 100 cases at a nominal 90% have a standard error near 3 points, and a single run landing at 87% is not evidence of a broken guarantee.

Value

One-row data.frame. `coverage` and `n` for either shape, plus `mean_width` and `median_width` for an interval, or `mean_set_size`, `singleton_rate` and `empty_rate` for a set.

Author(s)

EDG

Examples

```
x <- data.frame(age = rnorm(300), bmi = rnorm(300))
x[["y"]] <- x[["age"]] * 2 + rnorm(300, sd = 0.3)
mod <- train(
  x[1:200, ],
  dat_test = x[201:250, ],
```

```
  hyperparameters = setup_GLM(),
  verbosity = 0L
)
held_out <- x[251:300, ]
region <- conformal(mod, held_out[, c("age", "bmi")], verbosity = 0L)
conformal_metrics(region, held_out[["y"]])
```

`data_fingerprint` *Fingerprint a dataset*

Description

Computes a `DataFingerprint`: a content hash of `x` plus the structural facts (dimensions, column names) that make a mismatch between two runs diagnosable. Used to record which data a run used, so undocumented drift between experiments is visible.

Usage

```
data_fingerprint(x, method = "object", algorithm = "sha256", source = NULL)
```

Arguments

`x`	tabular data: The dataset. Required for every method, so that dimensions and column names are always recorded, even when hashing file bytes.
`method`	Character {"file", "object", "table"}: What to hash. "object" hashes the serialized R object (cheap, R-only). "file" hashes the raw bytes of `source`. "table" hashes the canonical Arrow IPC representation (cross-language; requires the `arrow` package), and is the only method under which one table held as a data.frame, a data.table, a tibble or a matrix gives one hash.
`algorithm`	Character {"sha256", "sha512", "sha384", "sha224", "sha3-256", "sha3-512", "blake2b", "blake2s", "sha1", "md5"}: Hash algorithm. "blake2b" and "blake2s" are the fast options; "sha1" and "md5" are for interoperability with systems that record those.
`source`	Optional Character: Path `x` was read from. Required when `method` is "file".

Value

`DataFingerprint` object.

Author(s)

EDG

Examples

```
fp <- data_fingerprint(iris)
fp
# A logically identical table in a different R container hashes differently
# under "object", which is what `@data_structure` records. FALSE:
data_fingerprint(iris)@hash ==
  data_fingerprint(data.table::as.data.table(iris))@hash
```

`data_profile` *Profile a dataset*

Description

Measures the facts validate_config checks a config against: dimensions, column types, distinct and missing counts per column, level counts for low-cardinality categorical columns, and the number of complete cases.

Usage

```
data_profile(x, n_duplicates = TRUE, fingerprint = FALSE)
```

Arguments

`x`	tabular data: The dataset.
`n_duplicates`	Logical: If TRUE, count rows that repeat an earlier row. The one whole-table operation here; FALSE leaves the field NULL.
`fingerprint`	Logical: If TRUE, record a data_fingerprint of `x`. Hashes the whole dataset, so it is off by default.

Details

One pass per column and no copy of the data, so profiling is cheap enough to do per validation.

Value

`DataProfile` object.

Author(s)

EDG

Examples

```
p <- data_profile(iris)
p@columns
p@level_counts
```

ddb_collect *Collect a lazy-read duckdb table*

Description

Collect a table read with `ddb_data(x, collect = FALSE)`

Usage

```
ddb_collect(sql, progress = TRUE, returnobj = c("data.frame", "data.table"))
```

Arguments

`sql`	Character: DuckDB SQL query, usually output of ddb_data with `collect = FALSE`
`progress`	Logical: If TRUE, show progress bar
`returnobj`	Character: data.frame or data.table: class of object to return

Value

`data.frame` or `data.table`.

Author(s)

EDG

Examples

```
## Not run:
# Requires local CSV file; replace with your own path
sql <- ddb_data("/Data/iris.csv", collect = FALSE)
ir <- ddb_collect(sql)

## End(Not run)
```

ddb_data *Read CSV using DuckDB*

Description

Lazy-read a CSV file, optionally: filter rows, remove duplicates, clean column names, convert character to factor, collect.

Usage

```
ddb_data(
  filename,
  datadir = NULL,
  sep = ",",
  header = TRUE,
  quotechar = "",
  ignore_errors = TRUE,
  make_unique = TRUE,
  select_columns = NULL,
  filter_column = NULL,
  filter_vals = NULL,
  character2factor = FALSE,
  collect = TRUE,
  progress = TRUE,
  returnobj = c("data.table", "data.frame"),
  data.table.key = NULL,
  clean_colnames = TRUE,
  verbosity = 1L
)
```

Arguments

`filename`	Character: file name; either full path or just the file name, if `datadir` is also provided.
`datadir`	Character: Optional path if `filename` is not full path.
`sep`	Character: Field delimiter/separator.
`header`	Logical: If TRUE, first line will be read as column names.
`quotechar`	Character: Quote character.
`ignore_errors`	Logical: If TRUE, ignore parsing errors.
`make_unique`	Logical: If TRUE, keep only unique rows.
`select_columns`	Character vector: Column names to select.
`filter_column`	Character: Name of column to filter on, e.g. "ID".
`filter_vals`	Numeric or Character vector: Values in `filter_column` to keep.
`character2factor`	Logical: If TRUE, convert character columns to factors.
`collect`	Logical: If TRUE, collect data and return structure class as defined by `returnobj`.
`progress`	Logical: If TRUE, enable the DuckDB progress bar.
`returnobj`	Character: "data.frame" or "data.table" object class to return. If "data.table", data.frame object returned from `DBI::dbGetQuery` is passed to `data.table::setDT`.
`data.table.key`	Character: If set, this corresponds to a column name in the dataset. This column will be set as key in the data.table output.
`clean_colnames`	Logical: If TRUE, clean colnames with clean_colnames.
`verbosity`	Integer: Verbosity level.

Value

data.frame or data.table if `collect` is TRUE, otherwise a character with the SQL query

Author(s)

EDG

Examples

```
## Not run:
# Requires local CSV file; replace with your own path
ir <- ddb_data("/Data/massive_dataset.csv",
  filter_column = "ID",
  filter_vals = 8001:9999
)

## End(Not run)
```

---

decomp *Perform Data Decomposition*

Description

Perform linear or non-linear decomposition of numeric data.

Usage

```
decomp(
  x,
  algorithm = "ICA",
  config = NULL,
  execution_config = setup_FutureExecution(),
  outdir = NULL,
  verbosity = 1L
)
```

Arguments

`x`	Matrix, data frame, or `DecomposeConfig` object: Input data, or a `DecomposeConfig` recipe (from setup_DecomposeConfig) carrying the data path, algorithm config, and output directory.
`algorithm`	Character: Decomposition algorithm. Not needed when `config` is supplied, which names its own; an explicit `algorithm` that disagrees with `config` is an error.
`config`	DecompositionConfig: Algorithm-specific config. Its `features` selects the columns of `x` to decompose; `NULL` selects every numeric column, since a decomposition reads a numeric matrix. The returned object's config carries the resolved names, and `apply_decomposition()` replays that selection.

`execution_config`
: `ExecutionConfig` object: Execution settings, e.g. setup_FutureExecution or setup_SerialExecution. A decomposition dispatches no work to other processes: an algorithm whose `threaded` trait is TRUE (see decomposition_traits) runs on `n_workers_algorithm` threads, or on the config's worker count when that is unset. Its `device` is the compute device of an algorithm that can use one (setup_Autoencoder); every other algorithm runs on the CPU. The config's `seed` seeds the fit, torch's generator included.

`outdir`
: Character, optional: Output directory. If not NULL, the returned `Decomposition` object is saved there as an `.rds` file, alongside a run record (`decomp_<algorithm>.record.json`) stating what the run resolved. See write_record.

`verbosity`
: Integer: Verbosity level.

Details

See docs.rtemis.org/r for detailed documentation.

Value

`Decomposition` object.

Author(s)

EDG

Examples

```
iris_pca <- decomp(exc(iris, "Species"), algorithm = "PCA")
```

`decomposition_metrics` *Decomposition Metrics*

Description

Quantify a fitted decomposition: how much of the data its components reconstruct, how redundant they are, and – given data the fit never saw – whether it generalizes.

Usage

```
decomposition_metrics(
  decom,
  x,
  new_data = NULL,
  execution_config = NULL,
  verbosity = 1L
)
```

Arguments

decom	Decomposition object.
x	Tabular data (data.frame, data.table, or tibble): The data decom was fitted on.
new_data	Optional tabular data: Data the fit never saw. Its columns must match those decom was fitted on.
execution_config	Optional ExecutionConfig object: Threads and device for encoding and reconstructing, as in apply_decomposition(). NULL uses the host's default worker count.
verbosity	Integer: Verbosity level.

Details

Which metrics are computed follows from the algorithm's traits; see decomposition_metric_traits. A metric its algorithm cannot support is NA.

The unprefixed metrics describe the fitted data, so x must be that data: they are computed against the components decom already holds. A different number of cases is an error, and x failing to match decom@data_fingerprint is reported without raising, since scoring a fit against an altered copy of its training data is a valid analysis.

decomp() already stores this on its result for the data it was fitted on, so calling this without new_data recomputes what decom@metrics holds. With new_data it computes reconstruction_gap, the difference between out-of-sample and in-sample relative reconstruction error, which indicates a decomposition that has fitted its training cases rather than their structure.

Reconstruction is measured in the units of the data as it was passed to decomp(), so the numbers are comparable across algorithms and across configurations of one algorithm. One caveat: ICA with row_norm = TRUE standardizes each case across features, and inverting that needs each case's own mean and standard deviation, taken from the data being reconstructed. Its reconstruction therefore draws on information the components do not carry, and its error is not comparable with an algorithm that reconstructs from components alone.

Value

DecompositionMetrics object.

Author(s)

EDG

Examples

```
x <- exc(iris, "Species")
iris_pca <- decomp(x, algorithm = "PCA", verbosity = 0L)
decomposition_metrics(iris_pca, x, verbosity = 0L)
```

`decomposition_metric_traits` *Which Decomposition Metrics Apply to Which Algorithm*

Description

Derived from decomposition_traits and the metric registry: a metric applies to an algorithm when every trait it requires is TRUE for that algorithm.

Usage

```
decomposition_metric_traits(algorithm = NULL)
```

Arguments

`algorithm`	Optional Character: Name of a decomposition algorithm, matched case-insensitively. `NULL` returns the full derived matrix.

Details

Nothing here is maintained per algorithm. `reconstruction_rmse` requires `invertible`, so it is defined for PCA, ICA and NMF and undefined for tSNE, UMAP and Isomap; `max_abs_component_correlation` requires nothing, so it is defined everywhere. An algorithm added to `decom_algorithms` with a correct traits row inherits the right metric set without any edit to the metrics code.

Value

data.frame. With `algorithm`, the registry rows that apply to it: one row per metric, with its group, requirements, whether it needs new data, direction and range. With `NULL`, one row per metric and one logical column per algorithm.

Author(s)

EDG

Examples

```
# Every metric, and which algorithms support it
decomposition_metric_traits()
# What can be computed for a UMAP embedding, which does not invert
decomposition_metric_traits("UMAP")
```

`decomposition_traits` *Decomposition Algorithm Traits*

Description

Machine-readable properties of each decomposition algorithm available to decomp: whether the map is linear, whether it can be applied to new data, whether it inverts, and what its objective preserves.

Usage

```
decomposition_traits(algorithm = NULL)
```

Arguments

`algorithm`	Optional Character: Name of a decomposition algorithm, matched case-insensitively. `NULL` returns every algorithm.

Details

The columns:

`name` The name used by `setup_<name>()` and `decomp(algorithm =)`.

`description` Full name of the algorithm.

`linear` The data is modeled as the components times a loadings matrix, which get_loadings returns.

`can_apply` A fitted result can be applied to new data with `apply_decomposition()`.

`invertible` There is an inverse map from component space back to input space, so reconstruction error is defined.

`orthogonal` Components are mutually orthogonal.

`ordered` Components come in a meaningful order, so "the first j" is well-defined.

`deterministic` The same input and configuration give the same result without setting a seed.

`preserves` What the objective preserves: `"variance"`, `"global"`, `"local"`, or `"reconstruction"`.

`nonneg` Requires non-negative input and produces non-negative factors.

`threaded` The fit runs on as many threads as the execution config gives the algorithm (`n_workers_algorithm`, or the worker count). Like `can_apply`, it states what rtemis implements.

`package` Package supplying the backend implementation.

`can_apply` states what rtemis implements, not what is mathematically possible. Parametric tSNE exists and Isomap admits a Nystrom-style out-of-sample extension; neither is implemented here, so both are `FALSE`. The column's contract is that `apply_decomposition()` on a fitted result of this algorithm returns components.

Value

data.frame: One row per algorithm, one column per trait.

Author(s)

EDG

Examples

```
decomposition_traits()
decomposition_traits("PCA")
# Which algorithms support out-of-sample projection?
decomposition_traits()[["name"]][decomposition_traits()[["can_apply"]]]
```

---

decomp_metrics *Decomposition Metrics (Deprecated)*

Description

`decomp_metrics()` is deprecated. Use `decomposition_metrics()`, which takes the same arguments.

Usage

```
decomp_metrics(
  decom,
  x,
  new_data = NULL,
  execution_config = NULL,
  verbosity = 1L
)
```

Arguments

`decom`	`Decomposition` object.
`x`	Tabular data (data.frame, data.table, or tibble): The data `decom` was fitted on.
`new_data`	Optional tabular data: Data the fit never saw. Its columns must match those `decom` was fitted on.
`execution_config`	Optional `ExecutionConfig` object: Threads and device for encoding and reconstructing, as in `apply_decomposition()`. `NULL` uses the host's default worker count.
`verbosity`	Integer: Verbosity level.

Value

As `decomposition_metrics()`.

Author(s)

EDG

Examples

```
x <- exc(iris, "Species")
iris_pca <- decomp(x, algorithm = "PCA", verbosity = 0L)
suppressWarnings(decomp_metrics(iris_pca, x, verbosity = 0L))
```

---

describe *Describe object*

Description

Describe object

Usage

```
describe(x, verbosity = 1L, ...)
```

Arguments

x	R object to describe. See method documentation for supported classes.
verbosity	Integer: Verbosity level.
...	Additional arguments passed to methods.

Details

Extra arguments for `factor` method:

- `max_n`: Integer: Return counts for up to this many levels.
- `return_ordered`: Logical: If TRUE, return levels ordered by count, otherwise return in level order.
- `verbosity`: Integer: Verbosity level.

Value

Character, invisibly.

Author(s)

EDG

Examples

```
# --- For `Supervised` objects ---
species_cart <- train(iris, hyperparameters = setup_CART())
describe(species_cart)

# --- For `SupervisedRes` objects ---
mod <- train(
  iris,
  hyperparameters = setup_CART(),
  outer_resampling_config = setup_KFold(n_resamples = 3L)
)
describe(mod)

# --- For factors ---
# Small number of levels
describe(iris[["Species"]])

# Large number of levels: show top n by count
x <- factor(sample(letters, 1000, TRUE))
describe(x)
describe(x, 3)
describe(x, 3, return_ordered = FALSE)
```

`df_movecolumn` *Move data frame column*

Description

Move data frame column

Usage

```
df_movecolumn(x, colname, to = ncol(x))
```

Arguments

`x`	data.frame.
`colname`	Character: Name of the column to move.
`to`	Integer: Which column position to move the vector to.

Value

data.frame

Author(s)

EDG

Examples

```
ir <- df_movecolumn(iris, colname = "Species", to = 1L)
```

`df_nunique_perfeat` *Unique values per feature*

Description

Get number of unique values per features

Usage

```
df_nunique_perfeat(x, excludeNA = FALSE)
```

Arguments

`x`	matrix or data frame input
`excludeNA`	Logical: If TRUE, exclude NA values from unique count.

Value

Vector, integer of length `NCOL(x)` with number of unique values per column/feature

Author(s)

EDG

Examples

```
df_nunique_perfeat(iris)
```

`draw_linad` *Plot a Linear Additive Tree*

Description

Draw a fitted LINAD model as an interactive tree: the partition as a hierarchy, each node's value on its label, and each node's linear coefficients in a table on hover.

Usage

```
draw_linad(
  x,
  n_leaves = NULL,
  top = 15L,
  sort_coefs = TRUE,
  direction = "UD",
  node_width = 180L,
  node_gap = 16L,
  level_separation = NULL,
  font_size = 14L,
  node_labels = TRUE,
  show_ncases = TRUE,
  show_node_value = TRUE,
  show_changes = TRUE,
  lo_col = sign_colors()[["negative"]],
  mid_col = NULL,
  hi_col = sign_colors()[["positive"]],
  root_col = NULL,
  node_col = NULL,
  leaf_col = NULL,
  hover_col = NULL,
  height = "600px",
  theme = NULL,
  verbosity = 1L
)
```

Arguments

x	`Regression` or `Classification` object holding a LINAD model, or a fitted LINAD model itself.
n_leaves	Optional Integer: Draw the tree at this size. NULL draws the size selected at training.
top	Optional Integer: Show this many coefficients per node, largest in absolute value first. NULL shows every one, which on a wide model is neither readable nor quick to render.
sort_coefs	Logical: If TRUE, order each table by absolute coefficient.
direction	Character {"UD", "DU", "LR", "RL"}: Tree direction.
node_width	Integer: Largest node width in pixels. Labels wrap to stay within it, which is what keeps a long feature name from widening a node past the space the layout reserves for it.
node_gap	Integer: Clear pixels to leave between two boxes on the same level. Separation is computed per pair from the two nodes' own widths, so a narrow node sits close to its neighbor.
level_separation	Optional Integer: Pixels between levels.

font_size	Integer: Label font size in pixels.
node_labels	Logical: If TRUE, label nodes with their rule.
show_ncases	Logical: If TRUE, include the case count in node labels.
show_node_value	Logical: If TRUE, include the node's own value in its label.
show_changes	Logical: If TRUE, mark each coefficient in the node's table with where it came from: a global effect, shared by every leaf, is marked with an identity sign, and one this node changed with a delta. A term with neither was inherited from the parent unchanged. The root carries no marks: it has no parent to differ from.
lo_col	Character: Color for the most negative coefficient. Defaults to the package-wide negative color, the same one `plot()` and `plot_manhattan()` use on a `MassGLM`.
mid_col	Optional Character: Color at zero. NULL uses the coefficient table's own background, so a coefficient of no consequence shows as no cell at all.
hi_col	Character: Color for the most positive coefficient. Defaults to the package-wide positive color.
root_col	Optional Character: Root node color. NULL follows the theme.
node_col	Optional Character: Internal node color. NULL follows the theme.
leaf_col	Optional Character: Terminal node color. NULL follows the theme. Kept off the two ends of the coefficient scale, so a hue in this plot means a coefficient's sign and nothing else.
hover_col	Optional Character: Border color marking the node the coefficient table belongs to. NULL uses the theme foreground.
height	Character: CSS height, as a length: a percentage of a container with no height of its own collapses.
theme	Optional rtemis.draw::Theme: Its background and text colors set the canvas and the foreground. Node fills, the coefficient table and the hover marker all follow them, chosen by the background's luminance so that a custom theme is handled too. NULL uses `rtemis.draw::theme_light()`. The two ends of the coefficient scale do not follow it: both ends read on either background, so a sign has the same color under every theme.
verbosity	Integer: If > 0, print messages.

Details

A LINAD leaf's coefficients are the accumulated sum along its path, so the same feature can carry a different slope, and a different sign, in each region of the data. The coefficient tables show this when compared across nodes. The color scale is diverging, centered on zero, and computed once across every table, so a sign change shows as a color flip.

Each node's label carries `node_value`, what the **tree alone** predicts there, while the table on hover carries what the **linear model** adds. The two are separate quantities in a fitted LINAD, so the plot shows them separately: with `node_model = "constant"` the tables are empty of slopes and the node values are the entire model, which is what a plain decision tree looks like.

The intercept is excluded from the color range. It is typically far larger than any slope and would compress every other coefficient into the middle of the scale.

This is the reference implementation for the LINAD tree plot, built on `visNetwork` because a tooltip carrying a real HTML table needs a backend whose tooltip is an HTML element. Plotly, which every other `draw_*` uses, renders hover labels as SVG and cannot express a table at all.

Value

`visNetwork` htmlwidget.

Author(s)

EDG

Examples

```
x <- data.frame(a = rnorm(300), b = rnorm(300))
x[["y"]] <- 2 * x[["a"]] + ifelse(x[["b"]] < 0, -3, 3) + rnorm(300)
mod <- train(x, hyperparameters = setup_LINAD(max_leaves = 5L))
draw_linad(mod)
```

---

dt_describe *Describe data.table*

Description

Describe data.table

Usage

```
dt_describe(x, verbosity = 1L)
```

Arguments

`x`	data.table: Input data.table.
`verbosity`	Integer: If > 0, print output to console.

Value

List with three data.tables: Numeric, Categorical, and Date.

Author(s)

EDG

Examples

```
library(data.table)
origin <- as.POSIXct("2022-01-01 00:00:00", tz = "America/Los_Angeles")
x <- data.table(
  ID = paste0("ID", 1:10),
  V1 = rnorm(10),
  V2 = rnorm(10, 20, 3),
  V1_datetime = as.POSIXct(
    seq(
      1, 1e7,
      length.out = 10
    ),
    origin = origin
  ),
  V2_datetime = as.POSIXct(
    seq(
      1, 1e7,
      length.out = 10
    ),
    origin = origin
  ),
  C1 = sample(c("alpha", "beta", "gamma"), 10, TRUE),
  F1 = factor(sample(c("delta", "epsilon", "zeta"), 10, TRUE))
)
```

---

dt_inspect_types *Inspect column types*

Description

Identifies columns that may be numeric but are stored as character or factor, by running inspect_type on each column.

Usage

```
dt_inspect_types(x, cols = NULL, verbosity = 1L)
```

Arguments

x	data.table: Input data.table.
cols	Character vector: columns to inspect.
verbosity	Integer: Verbosity level.

Value

Character vector.

Author(s)

EDG

Examples

```
library(data.table)
x <- data.table(
  id = 8001:8006,
  a = c("3", "5", "undefined", "21", "4", NA),
  b = c("mango", "banana", "tangerine", NA, "apple", "kiwi"),
  c = c(1, 2, 3, 4, 5, 6)
)
dt_inspect_types(x)
```

---

dt_keybin_reshape *Long to wide key-value reshaping*

Description

Reshape a long format `data.table` using key-value pairs with `data.table::dcast`

Usage

```
dt_keybin_reshape(
  x,
  id_name,
  key_name,
  positive = 1,
  negative = 0,
  xname = NULL,
  verbosity = 1L
)
```

Arguments

`x`	`data.table` object.
`id_name`	Character: Name of column in `x` that defines the IDs identifying individual rows.
`key_name`	Character: Name of column in `x` that holds the key.
`positive`	Numeric or Character: Used to fill id ~ key combination present in the long format input `x`.
`negative`	Numeric or Character: Used to fill id ~ key combination NOT present in the long format input `x`.
`xname`	Character: Name of `x` to be used in messages.
`verbosity`	Integer: Verbosity level.

Value

data.table in wide format.

Author(s)

EDG

Examples

```
library(data.table)
x <- data.table(
  ID = rep(1:3, each = 2),
  Dx = c("A", "C", "B", "C", "D", "A")
)
dt_keybin_reshape(x, id_name = "ID", key_name = "Dx")
```

---

dt_merge *Merge data.tables*

Description

Merge data.tables

Usage

```
dt_merge(
  left,
  right,
  on = NULL,
  left_on = NULL,
  right_on = NULL,
  how = "left",
  left_name = NULL,
  right_name = NULL,
  left_suffix = NULL,
  right_suffix = NULL,
  verbosity = 1L,
  ...
)
```

Arguments

left	data.table
right	data.table
on	Character: Name of column to join on.
left_on	Character: Name of column on left table.

right_on	Character: Name of column on right table.
how	Character: Type of join: "inner", "left", "right", "outer".
left_name	Character: Name of left table.
right_name	Character: Name of right table.
left_suffix	Character: If provided, add this suffix to all left column names, excluding on/left_on.
right_suffix	Character: If provided, add this suffix to all right column names, excluding on/right_on.
verbosity	Integer: Verbosity level.
...	Additional arguments to be passed to data.table::merge.

Value

Merged data.table.

Author(s)

EDG

Examples

```
library(data.table)
xleft <- data.table(ID = 1:5, Alpha = letters[1:5])
xright <- data.table(ID = c(3, 4, 5, 6), Beta = LETTERS[3:6])
xlr_inner <- dt_merge(xleft, xright, on = "ID", how = "inner")
```

---

dt_names_by_attr *List column names by attribute*

Description

List column names by attribute

Usage

```
dt_names_by_attr(x, attribute, exact = TRUE, sorted = TRUE)
```

Arguments

x	data.table: Input data.table.
attribute	Character: name of attribute.
exact	Logical: If TRUE, use exact matching.
sorted	Logical: If TRUE, sort the output.

Value

Character vector.

Author(s)

EDG

Examples

```
library(data.table)
x <- data.table(
  id = 1:5,
  sbp = rnorm(5, 120, 15),
  dbp = rnorm(5, 80, 10),
  paO2 = rnorm(5, 90, 10),
  paCO2 = rnorm(5, 40, 5)
)
setattr(x[["id"]], "source", "demographics")
setattr(x[["sbp"]], "source", "outpatient")
setattr(x[["dbp"]], "source", "outpatient")
setattr(x[["paO2"]], "source", "icu")
setattr(x[["paCO2"]], "source", "icu")

dt_names_by_attr(x, "source", "outpatient")
```

---

dt_nunique_perfeat *Number of unique values per feature*

Description

Number of unique values per feature

Usage

```
dt_nunique_perfeat(x, excludeNA = FALSE, limit = 20L, verbosity = 1L)
```

Arguments

x	data.table: Input data.table.
excludeNA	Logical: If TRUE, exclude NA values.
limit	Integer: Print up to this many features. Set to -1L to print all.
verbosity	Integer: If > 0, print output to console.

Value

Named integer vector of length `NCOL(x)` with number of unique values per column/feature, invisibly.

Author(s)

EDG

Examples

```
library(data.table)
ir <- as.data.table(iris)
dt_nunique_perfeat(ir)
```

`dt_pctmatch` *Get N and percent match of values between two columns of two data.tables*

Description

Get N and percent match of values between two columns of two data.tables

Usage

```
dt_pctmatch(x, y, on = NULL, left_on = NULL, right_on = NULL, verbosity = 1L)
```

Arguments

`x`	data.table: First input data.table.
`y`	data.table: Second input data.table.
`on`	Integer or character: column to read in `x` and `y`, if it is the same
`left_on`	Integer or character: column to read in `x`
`right_on`	Integer or character: column to read in `y`
`verbosity`	Integer: Verbosity level.

Value

list.

Author(s)

EDG

Examples

```
library(data.table)
x <- data.table(ID = 1:5, Alpha = letters[1:5])
y <- data.table(ID = c(3, 4, 5, 6), Beta = LETTERS[3:6])
dt_pctmatch(x, y, on = "ID")
```

dt_pctmissing *Get percent of missing values from every column*

Description

Get percent of missing values from every column

Usage

```
dt_pctmissing(x, verbosity = 1L)
```

Arguments

x	data.frame or data.table
verbosity	Integer: Verbosity level.

Value

list

Author(s)

EDG

Examples

```
library(data.table)
x <- data.table(a = c(1, 2, NA, 4), b = c(NA, NA, 3, 4), c = c("A", "B", "C", NA))
dt_pctmissing(x)
```

dt_set_autotypes *Set column types automatically*

Description

Inspects a data.table, identifies columns that hold numbers but were read as character, and converts their type **in-place**. Such columns arise, for example, when one or more fields contain non-numeric characters.

Usage

```
dt_set_autotypes(x, cols = NULL, verbosity = 1L)
```

Arguments

x	data.table: Input data.table. Will be modified **in-place**, if needed.
cols	Character vector: columns to work on. If not defined, will work on all columns
verbosity	Integer: Verbosity level.

Value

data.table, invisibly.

Author(s)

EDG

Examples

```
library(data.table)
x <- data.table(
  id = 8001:8006,
  a = c("3", "5", "undefined", "21", "4", NA),
  b = c("mango", "banana", "tangerine", NA, "apple", "kiwi"),
  c = c(1, 2, 3, 4, 5, 6)
)
str(x)
# ***in-place*** operation means no assignment is needed
dt_set_autotypes(x)
str(x)

# Try excluding column 'a' from autotyping
x <- data.table(
  id = 8001:8006,
  a = c("3", "5", "undefined", "21", "4", NA),
  b = c("mango", "banana", "tangerine", NA, "apple", "kiwi"),
  c = c(1, 2, 3, 4, 5, 6)
)
str(x)
# exclude column 'a' from autotyping
dt_set_autotypes(x, cols = setdiff(names(x), "a"))
str(x)
```

---

dt_set_cleanfactorlevels

Clean factor levels of data.table **in-place**

Description

Finds all factors in a data.table and cleans factor levels to include only underscore symbols

Usage

```
dt_set_cleanfactorlevels(x, prefix_digits = NA)
```

Arguments

x	data.table: Input data.table. Will be modified **in-place**.
prefix_digits	Character: If not NA, add this prefix to all factor levels that are numbers

Value

Nothing, modifies x **in-place**.

Author(s)

EDG

Examples

```
library(data.table)
x <- as.data.table(iris)
levels(x[["Species"]]) <- c("setosa:iris", "versicolor$iris", "virginica iris")
levels(x[["Species"]])
dt_set_cleanfactorlevels(x)
levels(x[["Species"]])
```

---

dt_set_clean_all *Clean column names and factor levels* **in-place**

Description

Clean column names and factor levels **in-place**

Usage

```
dt_set_clean_all(x, prefix_digits = NA)
```

Arguments

x	data.table: Input data.table. Will be modified **in-place**, if needed.
prefix_digits	Character: prefix to add to names beginning with a digit. Set to NA to skip

Value

Nothing, modifies x **in-place**.

Author(s)

EDG

Examples

```
library(data.table)
x <- as.data.table(iris)
levels(x[["Species"]]) <- c("setosa:iris", "versicolor$iris", "virginica iris")
names(x)
levels(x[["Species"]])
# ***in-place*** operation means no assignment is needed
dt_set_clean_all(x)
names(x)
levels(x[["Species"]])
```

---

dt_set_logical2factor *Convert data.table logical columns to factors*

Description

Convert data.table logical columns to factors with custom labels **in-place**

Usage

```
dt_set_logical2factor(
  x,
  cols = NULL,
  labels = c("False", "True"),
  maintain_attributes = TRUE,
  fillNA = NULL
)
```

Arguments

x	data.table: Input data.table. Will be modified **in-place**.
cols	Optional Integer or character: columns to convert. If NULL, operates on all logical columns.
labels	Character: labels for factor levels.
maintain_attributes	Logical: If TRUE, maintain column attributes.
fillNA	Optional Character: If not NULL, fill NA values with this constant.

Value

data.table, invisibly.

Author(s)

EDG

Examples

```
library(data.table)
x <- data.table(a = 1:5, b = c(TRUE, FALSE, FALSE, FALSE, TRUE))
x
dt_set_logical2factor(x)
x
z <- data.table(
  alpha = 1:5,
  beta = c(TRUE, FALSE, TRUE, NA, TRUE),
  gamma = c(FALSE, FALSE, TRUE, FALSE, NA)
)
# `fillNA` fills NA values with a constant
dt_set_logical2factor(z, cols = "beta", labels = c("No", "Yes"), fillNA = "No")
z
w <- data.table(mango = 1:5, banana = c(FALSE, FALSE, TRUE, TRUE, FALSE))
w
dt_set_logical2factor(w, cols = 2, labels = c("Ugh", "Huh"))
w
# Column attributes are maintained by default:
z <- data.table(
  alpha = 1:5,
  beta = c(TRUE, FALSE, TRUE, NA, TRUE),
  gamma = c(FALSE, FALSE, TRUE, FALSE, NA)
)
for (i in seq_along(z)) setattr(z[[i]], "source", "Guava")
str(z)
dt_set_logical2factor(z, cols = "beta", labels = c("No", "Yes"))
str(z)
```

---

dt_set_one_hot *Convert data.table's factor to one-hot encoding* **in-place**

Description

Convert data.table's factor to one-hot encoding **in-place**

Usage

```
dt_set_one_hot(x, xname = NULL, verbosity = 1L)
```

Arguments

x	data.table: Input data.table. Will be modified **in-place**.
xname	Character, optional: Dataset name.
verbosity	Integer: Verbosity level.

Value

The input, invisibly, after it has been modified **in-place**.

Author(s)

EDG

Examples

```
ir <- data.table::as.data.table(iris)
# dt_set_one_hot operates ***in-place***; therefore no assignment is used:
dt_set_one_hot(ir)
ir
```

---

exc *Exclude columns by character or numeric vector.*

Description

Exclude columns by character or numeric vector.

Usage

```
exc(x, idx)
```

Arguments

x	tabular data.
idx	Character or numeric vector: Column names or indices to exclude.

Value

data.frame, tibble, or data.table.

Author(s)

EDG

Examples

```
exc(iris, "Species") |> head()
exc(iris, c(1, 3)) |> head()
```

`explain` *Explain a prediction*

Description

Per-case explanation of a fitted model: why it predicted what it did, for each case in `newdata`. Where `get_varimp()` ranks features across a whole model, this attributes one prediction – a feature can rank third overall and be the entire reason for one case.

Usage

```
explain(x, newdata, background = NULL, config = NULL, verbosity = 1L, ...)
```

Arguments

`x`	`Supervised` or `SupervisedRes` object.
`newdata`	tabular data: Cases to explain. Predictors only, in training order, as `stats::predict()` requires.
`background`	Optional tabular data: Cases the contributions are measured against, in the same shape as `newdata`. Required by every estimator that needs `E[x]` or a reference set, which is most of them; those abort with a message naming it when it is missing.
`config`	Optional `ExplanationConfig` object: Built by setup_SHAP. NULL uses `setup_SHAP()`, which resolves the estimator per algorithm.
`verbosity`	Integer: Verbosity level.
`...`	Additional arguments passed to methods, such as `type` for a `SupervisedRes`.

Details

`background` **is what the contributions are measured against**, and most estimators cannot work without one. A contribution says how far a feature moved this case's prediction away from `E[f(x)]`, and that expectation is a property of the background data, and the model does not store the data it was trained on. Pass the training features, or a representative sample of them. The exceptions are the estimators that take their baseline from the model itself, such as the LightGBM family's; those ignore it, and everything else aborts without it.

Two explanations of one model against different backgrounds are not comparable. The result carries a fingerprint of the background it used.

`config` **chooses the method**, built by setup_SHAP: `setup_SHAP(perturbation = "conditional")` picks the value function, `setup_SHAP(estimator = "kernel")` overrides the estimator the algorithm would otherwise get. explanation_traits reports which estimator applies to which algorithm, whether it is exact, and why.

A `SupervisedRes` additionally takes `type`: `"avg"` (the default) averages the resamples' explanations, which decomposes the averaged prediction exactly, and `"all"` returns one explanation per resample.

The contributions plus the baseline reconstruct the prediction exactly, on the scale they were computed on. For classification that scale is the model's margin, **not** the probability `stats::predict()` returns: probability is a nonlinear transform of the margin, so contributions do not sum to it.

Read the result with shap_case for one case, shap_long across cases, shap_by_level for the levels of a categorical, and `get_varimp()` for `mean(|phi|)` per feature.

A SHAP value is not a causal effect. It attributes a *prediction* under the model's own logic; intervening on a feature in the world does not move the outcome by its attribution. Correlated features split credit between themselves, and how they split it depends on the value function – so a small attribution does not show a feature to be unimportant, and this is not a feature-selection criterion.

Value

`SHAP` object, or a named list of them for a `SupervisedRes` with `type = "all"`.

Author(s)

EDG

Examples

```
x <- data.frame(age = rnorm(100), bmi = rnorm(100))
x[["y"]] <- x[["age"]] * 2 + rnorm(100, sd = 0.3)
features <- x[, c("age", "bmi")]
mod <- train(x, hyperparameters = setup_GLM(), verbosity = 0L)

# `background` is what the contributions are measured against.
contributions <- explain(
  mod,
  features[1:5, ],
  background = features,
  verbosity = 0L
)
contributions

# Contributions plus the baseline reconstruct the prediction, exactly:
rowSums(contributions@phi[["outcome"]]) + contributions@baseline[, 1L]
predict(mod, features[1:5, ], verbosity = 0L)

# One case, ordered the way a waterfall reads:
shap_case(contributions, features[1:5, ], case = 1L)

# Which features mattered across these cases:
get_varimp(contributions)

# `config` chooses the method; here, a subsampled background.
explain(
  mod,
  features[1:5, ],
  background = features,
  config = setup_SHAP(background_n = 50L),
  verbosity = 0L
```

```
)
```

`explanation_methods` *explanation_methods() (Deprecated)*

Description

`explanation_methods()` is deprecated. Use `explanation_traits()`, which takes the same arguments.

Usage

```
explanation_methods(algorithm = NULL)
```

Arguments

`algorithm`	Optional Character: Name of a supervised algorithm, matched case-insensitively. NULL returns every algorithm.

Value

As `explanation_traits()`.

Author(s)

EDG

Examples

```
suppressWarnings(explanation_methods("Ranger"))
```

`explanation_traits` *Explanation Methods per Algorithm*

Description

What explain does to a model fitted by each supervised algorithm: which estimator `setup_SHAP(estimator = "auto")` resolves to, whether that answer is exact, which value function it computes, and why.

Usage

```
explanation_traits(algorithm = NULL)
```

Arguments

`algorithm`	Optional Character: Name of a supervised algorithm, matched case-insensitively. NULL returns every algorithm.

Details

The columns:

`name` The algorithm, as named by `setup_<name>()`.

`estimator` The estimator `"auto"` resolves to. `"KernelSHAP"` is the model-agnostic fallback and applies to every algorithm; the others are exact and algorithm-specific.

`exact` TRUE if the estimator returns Shapley values rather than an estimate of them, NA where that depends on the fitted model rather than on the algorithm – an ensemble whose meta-learner turned out to be linear, or a model that has one linear map for a regression or binary outcome but not for a multiclass one. An NA row explains what it did when asked, and refuses with a message naming the reason when it cannot.

`perturbation` The value function computed when `setup_SHAP(perturbation = )` is left NULL.

`rationale` Why this estimator, in one line.

`estimator = "kernel"` is always available, so a row whose `exact` is FALSE is saying that `estimator = "exact"` has nothing to offer for it – not that the algorithm cannot be explained.

Value

data.frame: One row per algorithm.

Author(s)

EDG

Examples

```
explanation_traits()
explanation_traits("Ranger")
# Which algorithms have an exact estimator?
em <- explanation_traits()
em[["name"]][!is.na(em[["exact"]]) & em[["exact"]]]
```

---

features *Get features from tabular data*

Description

Returns all columns except the last one.

Usage

```
features(x)
```

Arguments

`x`	tabular data: Input data to get features from.

Details

This can be applied to tabular datasets used for supervised learning in **rtemis**, where, by convention, the last column is the outcome variable and all other columns are features.

Value

Object of the same class as the input, after removing the last column.

Author(s)

EDG

Examples

```
features(iris) |> head()
```

---

feature_matrix *Convert tabular data to feature matrix*

Description

Convert a tabular dataset to a matrix, one-hot encoding factors, if present.

Usage

```
feature_matrix(x)
```

Arguments

x	tabular data: Input data to convert to a feature matrix.

Details

This is a convenience function that uses `features()`, `preprocess()`, `as.matrix()`.

Value

Matrix with features. Factors are one-hot encoded, if present.

Author(s)

EDG

Examples

```
# reorder columns so that we have a categorical feature
x <- set_outcome(iris, "Sepal.Length")
feature_matrix(x) |> head()
```

feature_names *Get feature names*

Description

Returns all column names except the last one

Usage

```
feature_names(x)
```

Arguments

x	tabular data.

Details

This applied to tabular datasets used for supervised learning in rtemis, where, by convention, the last column is the outcome variable and all other columns are features.

Value

Character vector of feature names.

Author(s)

EDG

Examples

```
feature_names(iris)
```

fitted_config *fitted_config() (Deprecated)*

Description

`fitted_config()` is deprecated. Use `get_fitted_config()`, which takes the same arguments.

Usage

```
fitted_config(preprocessor)
```

Arguments

preprocessor	`Preprocessor` object.

Value

As get_fitted_config().

Author(s)

EDG

Examples

```
iris_pre <- preprocess(iris, setup_Preprocessor(scale = TRUE), verbosity = 0L)
suppressWarnings(fitted_config(iris_pre))
```

---

fit_predict *Fit a Learner and Predict New Data*

Description

Train one regression model on x and y with an algorithm named as a string, then return its predictions for newdata, their standard errors when the algorithm provides them, and the training R-squared.

Usage

```
fit_predict(
  x,
  y,
  newdata,
  algorithm,
  params = NULL,
  se = FALSE,
  verbosity = 0L
)
```

Arguments

x	data.frame: Features, one column per predictor.
y	Numeric: Outcome, one value per row of x.
newdata	data.frame: Features to predict, with the columns of x.
algorithm	Character: Supervised learning algorithm name, matched case-insensitively, e.g. "LINAD", "CART", "LightGBM". See available_supervised().
params	Optional Named list: Arguments to the algorithm's setup function, e.g. list(max_leaves = 8) for setup_LINAD(). NULL uses the algorithm's defaults.
se	Logical: If TRUE, compute standard errors of the fit for newdata.
verbosity	Integer: Verbosity level.

Details

This is the bridge behind `fit = "<algorithm>"` in `rtemis.draw::draw_scatter()`, `rtemis.draw::draw_fit()` and `rtemis.draw::draw_scatter3d()`, which need a fitted curve or surface from a name alone. For full control over training, use `train()`.

Value

Named list: `fitted` (Numeric predictions for `newdata`), `se` (Numeric standard errors, or NULL when `se = FALSE` or the algorithm has none), `rsq` (Numeric training R-squared), and `algorithm` (Character canonical algorithm name).

Author(s)

EDG

Examples

```
fit <- fit_predict(
  mtcars["wt"], mtcars[["mpg"]],
  newdata = data.frame(wt = c(2, 3, 4)),
  algorithm = "GLM",
  se = TRUE
)
fit[["fitted"]]
fit_predict(
  mtcars["wt"], mtcars[["mpg"]],
  newdata = data.frame(wt = c(2, 3, 4)),
  algorithm = "LINAD",
  params = list(max_leaves = 2)
)[["fitted"]]
```

---

getnames *Get names by string matching or class*

Description

Get names by string matching or class

Usage

```
getnames(
  x,
  pattern = NULL,
  starts_with = NULL,
  ends_with = NULL,
  ignore_case = TRUE
)
```

```
getfactornames(x)

getnumericnames(x)

getlogicalnames(x)

getcharacternames(x)

getdatenames(x)
```

Arguments

x	object with names() method.
pattern	Character: pattern to match anywhere in names of x.
starts_with	Character: pattern to match in the beginning of names of x.
ends_with	Character: pattern to match at the end of names of x.
ignore_case	Logical: If TRUE, well, ignore case.

Details

For getnames() only: pattern, starts_with, and ends_with are applied sequentially. If more than one is provided, the result will be the intersection of all matches.

Value

Character vector of matched names.

Author(s)

EDG

Examples

```
getnames(iris, starts_with = "Sepal")
getnames(iris, ends_with = "Width")
getfactornames(iris)
getnumericnames(iris)
```

---

getnamesandtypes *Get data.frame names and types*

Description

Get data.frame names and types

Usage

```
getnamesandtypes(x)
```

Arguments

x	data.frame / data.table or similar

Value

character vector of column names with attribute "type" holding the class of each column

Author(s)

EDG

Examples

```
getnamesandtypes(iris)
```

---

get_factor_names *Get factor names (Deprecated)*

Description

`get_factor_names()` is deprecated. Use `getfactornames()`, which returns the names of every factor column of `x`.

Usage

```
get_factor_names(x)
```

Arguments

x	tabular data.

Value

Character vector of the names of the factor columns of `x`.

Author(s)

EDG

Examples

```
suppressWarnings(get_factor_names(iris))
```

get_fitted_config *A* Preprocessor*'s config with its learned values filled in*

Description

`@config` holds what the user asked for and `@values` what preprocessing learned, which keeps the input intact – but re-applying to new data needs the two merged, and so does a run record, which must report the values used. One merge, so the two cannot disagree.

Usage

```
get_fitted_config(preprocessor)
```

Arguments

`preprocessor`	`Preprocessor` object.

Details

The merged fields are settable inputs: supplying `scale_centers` makes `preprocess()` use it instead of computing one. That is why the merge is `@values` over `@config` and not the reverse – a learned value only fills a slot the user left empty.

Value

`PreprocessorConfig` with the learned values applied.

Author(s)

EDG

See Also

apply_preprocessor, setup_Preprocessor

Examples

```
iris_pre <- preprocess(iris, setup_Preprocessor(scale = TRUE, center = TRUE))
# The config as supplied carries no centers; the fitted one carries the
# centers `preprocess()` computed.
get_fitted_config(iris_pre)@scale_centers
```

`get_learning_curve` *Learning curve of a fitted model*

Description

The loss recorded at every step of training, as data: one row per step, with the training and validation loss where the algorithm records them.

Returns NULL for an algorithm that records no learning curve, so it is safe to call on any model.

Usage

```
get_learning_curve(x, ...)
```

Arguments

`x`	`Supervised` object.
`...`	Additional arguments passed to methods.

Value

data.frame with columns `iteration`, `loss_training` and `loss_validation`, carrying attributes `unit` and `selected`; or NULL.

Author(s)

EDG

See Also

plot_learning, which draws it

Examples

```
dat <- set_outcome(iris[, 1:4], "Sepal.Length")
mod <- train(
  dat[1:100, ],
  dat_validation = dat[101:150, ],
  hyperparameters = setup_LINAD(max_leaves = 12L)
)
get_learning_curve(mod)
```

`get_loadings` *Get Decomposition Loadings*

Description

The loadings of a linear decomposition: the weight of each feature on each component.

Usage

```
get_loadings(decom)
```

Arguments

`decom`	`Decomposition` object.

Details

Loadings are defined for algorithms whose `linear` trait is TRUE; see decomposition_traits. The loadings matrix `L` is the one through which the components map back to the data, `transformed %*% t(L)`, in the space the backend fits in:

PCA The rotation, whose columns are the principal axes.

ICA The transposed mixing matrix.

NMF The basis matrix, non-negative.

Rows are named by the features the decomposition was fitted on and columns by its components, as in `decom@transformed`.

Value

Numeric matrix, features by components, or NULL when the algorithm has no loadings.

Author(s)

EDG

Examples

```
iris_pca <- decomp(exc(iris, "Species"), algorithm = "PCA", verbosity = 0L)
get_loadings(iris_pca)
```

`get_membership` *Get cluster membership from a soft* `Clustering`

Description

Returns, for each case of the data a soft clustering was fitted on, its weight on each cluster: a fuzzy membership degree for c-means, a posterior probability for a Gaussian mixture model.

Usage

```
get_membership(x)
```

Arguments

`x`	`Clustering`: A soft `Clustering` object, produced by an algorithm such as CMeans or GMM.

Details

Column `j` is named `"j"` and corresponds to cluster label `j` in the output of `clusters()`. Each row sums to 1. Each call returns a new data.table, so modifying the result by reference leaves `x` unchanged. A hard clustering, which assigns each case to one cluster with no weights, raises an error.

Value

data.table: One row per case, one column per cluster.

Author(s)

EDG

Examples

```
iris_cm <- cluster(exc(iris, "Species"), algorithm = "CMeans", verbosity = 0L)
get_membership(iris_cm)
```

`get_metrics` *Get metrics*

Description

The performance metrics stored on a fitted model, clustering, or decomposition, for one sample.

Usage

```
get_metrics(x, sample = NULL, calibrated = FALSE)
```

Arguments

`x`	`Supervised`, `SupervisedRes`, `Clustering`, or `Decomposition` object.
`sample`	Optional Character {"training", "validation", "test"}: The sample whose metrics to return. Required for `Supervised` and `SupervisedRes` objects.
`calibrated`	Logical: If TRUE, return the metrics of the calibrated predictions of a calibrated classifier.

Details

A `Supervised` model stores metrics for the training sample, and for the validation and test samples when train received `dat_validation` and `dat_test`. A `SupervisedRes` model stores per-resample metrics for the training and test samples of each outer resample, with their mean and standard deviation. A calibrated classifier also stores the metrics of its calibrated predictions, selected with `calibrated = TRUE`.

A `Clustering` or `Decomposition` stores metrics for the data it was fitted on, which `sample = NULL` and `sample = "training"` both select. For a decomposition, `decomposition_metrics()` computes the metrics of new data.

Requesting a sample the object does not store raises an error that names the samples it does store.

The returned object supports `$` and `[[`: for example, `get_metrics(mod, "test")$rsq`.

Value

`Metrics` object for a `Supervised` model; `MetricsRes` object for a `SupervisedRes` model; `ClusteringMetrics` or `DecompositionMetrics` object, or NULL when the object holds no metrics, for a `Clustering` or `Decomposition`.

Author(s)

EDG

Examples

```
x <- data.frame(age = rnorm(100), bmi = rnorm(100))
x[["y"]] <- x[["age"]] * 2 + rnorm(100, sd = 0.3)
mod <- train(
  x[1:80, ],
  dat_test = x[81:100, ],
  hyperparameters = setup_GLM(),
  verbosity = 0L
)
get_metrics(mod, "training")
get_metrics(mod, "test")$rsq
```

---

get_mode *Mode of a vector (Deprecated)*

Description

`get_mode()` is deprecated. Use `mode_value()`, which takes the same arguments.

Usage

```
get_mode(x, na.rm = TRUE, getlast = TRUE, retain_class = TRUE)
```

Arguments

`x`	Vector, factor or integer: Input data.
`na.rm`	Logical: If TRUE, exclude NAs (using `na.exclude(x)`).
`getlast`	Logical: If TRUE, get the last value in case of ties.
`retain_class`	Logical: If TRUE, output is always same class as input.

Value

As `mode_value()`.

Author(s)

EDG

Examples

```
suppressWarnings(get_mode(c(9, 3, 4, 4, 0, 2, 2, NA)))
```

`get_predictions` *Get predictions*

Description

The true and predicted values stored on a fitted supervised model, with the predicted class probabilities of a classifier, one row per case.

Usage

```
get_predictions(x, sample = NULL, calibrated = FALSE)
```

Arguments

`x`	`Supervised` or `SupervisedRes` object.
`sample`	Optional Character {"training", "validation", "test"}: The sample whose predictions to return.
`calibrated`	Logical: If TRUE, return the calibrated predictions of a calibrated classifier.

Details

A `Supervised` model stores predictions for the training sample, and for the validation and test samples when train received `dat_validation` and `dat_test`. A `SupervisedRes` model stores the training and test predictions of each outer resample. The training predictions are the model's fitted values, `stats::fitted()`. A calibrated classifier also stores its calibrated predictions, selected with `calibrated = TRUE`.

`sample = NULL` returns every stored sample, stacked in the order training, validation, test. The samples are the same ones `get_metrics()` stores metrics for. Requesting a sample the model does not store raises an error that names the samples it does store.

Value

data.table with one row per case of each selected sample and the columns `sample`; `resample`, the outer resample identifier, for a `SupervisedRes` model; `true`; `predicted`; and, for a classifier, one `prob_<class>` column per stored probability column: the positive class of a binary classifier, every class of a multiclass one.

Author(s)

EDG

Examples

```
mod <- train(
  iris[1:120, ],
  dat_test = iris[121:150, ],
  hyperparameters = setup_CART(),
  verbosity = 0L
)
head(get_predictions(mod, "test"))
```

`get_varimp` *Get variable importance*

Description

The variable importance measures a fitted model reports, or the mean absolute contribution of each predictor in a `SHAP` explanation.

Usage

```
get_varimp(x, ...)
```

Arguments

`x`	`Supervised`, `SupervisedRes` or `SHAP` object.
`...`	Additional arguments passed to methods.

Details

A `VariableImportance` object holds its measures by name in `@measures`. Each measure states what it quantifies (`@kind`, for example split gain, permutation, or coefficient), the cases it was computed on (`@computed_on`), whether its values can be negative (`@signed`) or depend on the units of the predictors (`@scale_dependent`), how it ranks predictors (`@direction`), and what its values are for this model (`@description`); `@values` holds the value for each predictor. varimp_table combines the measures into one data.frame.

Value

`VariableImportance` object, a list of them (one per resample) for a `SupervisedRes`, or NULL when the algorithm reports no importance.

Author(s)

EDG

Examples

```
mod <- train(iris, hyperparameters = setup_LightRF())
get_varimp(mod)
```

HyperparameterCandidates *The values a tuner may choose among for one hyperparameter*

Description

Built by `tune_over()` and stored in the hyperparameter it is assigned to, so that a search space is a distinct type, never inferred from a vector's length.

Usage

```
HyperparameterCandidates(candidates = list(), from_vector = FALSE)
```

Arguments

`candidates`	List: The values to search over, one per element.
`from_vector`	Logical: TRUE when the candidates were read out of a single bare vector. A vector is also how one value of a vector-valued hyperparameter is written, so that one combination cannot be inferred; this records it so the hyperparameter can reject it with a correction.

Value

`HyperparameterCandidates` object.

Author(s)

EDG

Examples

```
# Built by tune_over(), not directly.
d <- tune_over(3L, 4L, 5L)
d@candidates
```

inc *Select (include) columns by character or numeric vector.*

Description

Select (include) columns by character or numeric vector.

Usage

```
inc(x, idx)
```

Arguments

x	tabular data.
idx	Character or numeric vector: Column names or indices to include.

Value

data.frame, tibble, or data.table.

Author(s)

EDG

Examples

```
inc(iris, c(3, 4)) |> head()
inc(iris, c("Sepal.Length", "Species")) |> head()
```

---

index_col_by_attr *Index columns by attribute name & value*

Description

Index columns by attribute name & value

Usage

```
index_col_by_attr(x, name, value, exact = TRUE)
```

Arguments

x	tabular data.
name	Character: Name of attribute.
value	Character: Value of attribute.
exact	Logical: Passed to `attr` when retrieving attribute value. If `TRUE`, attribute name must match `name` exactly, otherwise, partial match is allowed.

Value

Integer vector.

Author(s)

EDG

Examples

```
library(data.table)
x <- data.table(
  id = 1:5,
  sbp = rnorm(5, 120, 15),
  dbp = rnorm(5, 80, 10),
  paO2 = rnorm(5, 90, 10),
  paCO2 = rnorm(5, 40, 5)
)
setattr(x[["sbp"]], "source", "outpatient")
setattr(x[["dbp"]], "source", "outpatient")
setattr(x[["paO2"]], "source", "icu")
setattr(x[["paCO2"]], "source", "icu")
index_col_by_attr(x, "source", "icu")
```

---

ingest *Read a data file and normalize it to Parquet*

Description

Reads a data file with the settings of an `IngestConfig`, converts the columns it declares types for, writes the result to a Parquet file, and returns a manifest of the operation.

The file extension names the format. A `config` whose `format` differs from it is an error.

`path` may be an `http://` or `https://` URL or an `hf://datasets/` path, as `read()` accepts them, downloaded to a temporary file. The manifest records the path as given in `input`, the time the download started in `retrieved_at`, and, for an `hf://` path, the path pinned to the resolved commit and file in `input_resolved`.

The manifest also records the input and output files, the SHA-256 of the input, a `DataFingerprint` of the data, the config, and the engine that produced the Parquet file.

Usage

```
ingest(path, outfile, config = NULL, overwrite = FALSE, verbosity = 1L)
```

Arguments

`path`	Character: Path to the input file, an `http://` or `https://` URL, or an `hf://datasets/` path.
`outfile`	Character: Path to write the Parquet file to.
`config`	Optional `IngestConfig`: How to read it. If `NULL`, the defaults for the file's own format.
`overwrite`	Logical: If TRUE, overwrite `outfile` if it exists.
`verbosity`	Integer: Verbosity level.

Value

Named list: the ingest manifest, with `input`, `input_resolved` (`hf://datasets/<repo>@<commit>/<file>` for an `hf://` input, otherwise NULL), `input_sha256` (SHA-256 of the input file's bytes), `retrieved_at` (ISO 8601 UTC time the download started, or NULL for a local file), `output`, `format`, `config`, `engine`, `rtemis_version`, `data_input`, `n_rows`, and `n_cols`.

Author(s)

EDG

Examples

```
## Not run:
ingest("data.csv", "data.parquet")
ingest("data.csv", "data.parquet", setup_DelimitedIngest(sep = ";"))
ingest("https://example.org/data/iris.csv", "iris.parquet")
ingest("hf://datasets/rtemis/tcga-pancan-hiseq", "tcga.parquet")

## End(Not run)
```

`init_project_dir` *Initialize Project Directory*

Description

Initializes Directory Structure: "R", "Data", "Results"

Usage

```
init_project_dir(path, output_dir = "Out", verbosity = 1L)
```

Arguments

`path`	Character: Path to initialize project directory in.
`output_dir`	Character: Name of output directory to create.
`verbosity`	Integer: Verbosity level.

Value

Character: the path where the project directory was initialized, invisibly.

Author(s)

EDG

Examples

```
## Not run:
# Creates the "my_project" directory and its subdirectories
init_project_dir("my_project")

## End(Not run)
```

`inspect` *Inspect rtemis object*

Description

Inspect rtemis object

Usage

```
inspect(x)
```

Arguments

`x`	R object to inspect.

Value

Called for side effect of printing information to console; returns character string invisibly.

Author(s)

EDG

Examples

```
inspect(iris)
```

`inspect_type` *Inspect character and factor vector*

Description

Checks character or factor vector to determine whether it might be best to convert to numeric.

Usage

```
inspect_type(x, xname = NULL, verbosity = 1L, thresh = 0.5, na.omit = TRUE)
```

Arguments

x	Character or factor vector.
xname	Character: Name of input vector x.
verbosity	Integer: Verbosity level.
thresh	Numeric: Threshold for determining whether to convert to numeric.
na.omit	Logical: If TRUE, remove NA values before checking.

Details

All data can be represented as a character string. A numeric variable may be read as a character variable if there are non-numeric characters in the data. Such variables are detected so they can be converted to numeric, which introduces NA values.

Value

Character.

Author(s)

EDG

Examples

```
x <- c("3", "5", "undefined", "21", "4", NA)
inspect_type(x)
z <- c("mango", "banana", "tangerine", NA)
inspect_type(z)
```

---

is_constant *Check if vector is constant*

Description

Check if vector is constant

Usage

```
is_constant(x, skip_missing = FALSE)
```

Arguments

x	Vector: Input
skip_missing	Logical: If TRUE, skip NA values before test

Value

Logical.

Author(s)

EDG

Examples

```
x <- rep(9, 1000000)
is_constant(x)
x[10] <- NA
is_constant(x)
is_constant(x, skip_missing = TRUE)
```

---

is_same_data *Do two fingerprints identify the same dataset?*

Description

Two fingerprints are comparable only if they were produced the same way, so this is FALSE when `encoding` or `algorithm` differ.

Usage

```
is_same_data(x, y)
```

Arguments

x, y	`DataFingerprint` objects.

Value

Logical.

Author(s)

EDG

Examples

```
is_same_data(data_fingerprint(iris), data_fingerprint(iris))
```

`JSONSchema_to_S7` *Build an S7 class from a JSON Schema*

Description

The read direction of S7_to_JSONSchema: reads each property's `PropertySpec` back out of the schema and assembles a live S7 class whose properties carry the same types, bounds, enums, containers, and validators as the class the schema was generated from.

Usage

```
JSONSchema_to_S7(
  schema,
  defaults = NULL,
  refs = NULL,
  name = NULL,
  package = NULL,
  authoring = NULL,
  declarations = NULL,
  schemas = NULL,
  decode_reference = NULL,
  parent = NULL,
  policies = NULL
)
```

Arguments

`schema`	Named list: A JSON Schema, as produced by S7_to_JSONSchema or parsed from the published tree with `jsonlite::fromJSON(simplifyVector = FALSE)`.
`defaults`	Optional named list: Format version 1 defaults artifact, or explicit property declaration values for an independently supplied schema.
`refs`	Optional named list: Property name to S7 class, for properties published as a `$ref` to another schema.
`name`	Optional Character: Class name. Defaults to the schema `title`.
`package`	Optional Character: Package name recorded on the class.
`authoring`	Optional named list: Property name to logical authorship policy, from the separate authoring artifact under this schema's `$id`.
`declarations`	Optional List: Declaration descriptors keyed by schema pointer.
`schemas`	Optional List: Complete parsed schema graph keyed by canonical ID. Used with a format version 1 artifact to reconstruct typed references without source classes.
`decode_reference`	Optional Function: Reference decoder for a custom artifact graph.
`parent`	Optional S7 class: Reconstructed parent type.
`policies`	Optional List: Serialized input policy descriptors.

Details

Defaults are supplied in a separate versioned artifact keyed by schema `$id`. The published schemas carry no `default` keywords. Absent declaration defaults remain absent; constructing a non-nullable property without a default requires a value.

Properties holding a nested config are published as a `$ref` to that config's own schema. Supply the parsed graph through `schemas` and the format version 1 artifact through `defaults` to reconstruct these references. Independently supplied S7 reference classes can be passed through `refs`. Reconstructed constructors use declaration defaults; input resolution policies remain separate metadata on the class. Complex defaults are reconstructed from their serialized values for each instance, including references handled by `decode_reference`.

Value

S7 class.

Author(s)

EDG

Examples

```
# A schema as `S7_to_JSONSchema()` emits it: standard keywords carry the
# bounds, `x-rtemis` carries the leaf type.
schema <- list(
  title = "Demo",
  properties = list(
    k = list(
      type = "integer",
      minimum = 1L,
      maximum = 10L,
      description = "Number of components.",
      `x-rtemis` = list(type = "integer")
    )
  )
)
Demo <- JSONSchema_to_S7(schema, defaults = list(k = 2L))
Demo()@k
Demo(k = 5L)@k
# The reconstructed class enforces the schema's bounds.
tryCatch(Demo(k = 99L), error = conditionMessage)
```

---

massGLM *Mass-univariate GLM Analysis*

Description

Mass-univariate GLM Analysis

Usage

```
massGLM(x, y, scale_y = NULL, center_y = NULL, verbosity = 1L)
```

Arguments

x	tabular data: Predictor variables. Usually a small number of covariates.
y	data.frame or similar: Each column is a different outcome. The function will train one GLM for each column of y. Usually a large number of features.
scale_y	Logical: If TRUE, scale each column of y to have mean 0 and sd 1. If NULL, set to TRUE if y is numeric, FALSE otherwise.
center_y	Logical: If TRUE, center each column of y to have mean 0. If NULL, set to TRUE if scale_y is TRUE, FALSE otherwise.
verbosity	Integer: Verbosity level.

Value

MassGLM object.

Author(s)

EDG

Examples

```
set.seed(2022)
y <- rnormmat(500, 40, return_df = TRUE)
x <- data.frame(
  x1 = y[[3]] - y[[5]] + y[[14]] + rnorm(500),
  x2 = y[[21]] + rnorm(500),
  g = factor(sample(letters[1:3], 500, replace = TRUE))
)
massmod <- massGLM(x, y)
# Print table of coefficients, p-values, etc. for all models
summary(massmod)
```

---

matchcases *Match cases by covariates*

Description

Find one or more cases from a pool data.frame that match cases in a target data.frame. Match exactly and/or by distance (sum of squared distances).

Usage

```
matchcases(
  target,
  pool,
  n_matches = 1,
  target_id = NULL,
  pool_id = NULL,
  exactmatch_factors = TRUE,
  exactmatch_cols = NULL,
  distmatch_cols = NULL,
  norepeats = TRUE,
  ignore_na = FALSE,
  verbosity = 1L
)
```

Arguments

target	data.frame: Cases to match against.
pool	data.frame: Cases to draw matches from.
n_matches	Integer: Number of matches to return.
target_id	Character: Column name in target that holds unique cases IDs. If NULL, integer case numbers will be used.
pool_id	Character: Same as target_id for pool.
exactmatch_factors	Logical: If TRUE, selected cases will have to exactly match factors available in target.
exactmatch_cols	Character: Names of columns that should be matched exactly.
distmatch_cols	Character: Names of columns that should be distance-matched.
norepeats	Logical: If TRUE, cases in pool can only be chosen once.
ignore_na	Logical: If TRUE, ignore NA values during exact matching.
verbosity	Integer: Verbosity level.

Value

data.frame

Author(s)

EDG

Examples

```
set.seed(2021)
cases <- data.frame(
  PID = paste0("PID", seq(4)),
```

```
  Sex = factor(c(1, 1, 0, 0)),
  Handedness = factor(c(1, 1, 0, 1)),
  Age = c(21, 27, 39, 24),
  Var = c(.7, .8, .9, .6),
  Varx = rnorm(4)
)
controls <- data.frame(
  CID = paste0("CID", seq(50)),
  Sex = factor(sample(c(0, 1), 50, TRUE)),
  Handedness = factor(sample(c(0, 1), 50, TRUE, c(.1, .9))),
  Age = sample(16:42, 50, TRUE),
  Var = rnorm(50),
  Vary = rnorm(50)
)

mc <- matchcases(cases, controls, 2, "PID", "CID")
```

`mgetnames` *Get names by string matching multiple patterns*

Description

Get names by string matching multiple patterns

Usage

```
mgetnames(
  x,
  pattern = NULL,
  starts_with = NULL,
  ends_with = NULL,
  ignore_case = TRUE,
  return_index = FALSE
)
```

Arguments

`x`	Character vector or object with `names()` method.
`pattern`	Character vector: pattern(s) to match anywhere in names of x.
`starts_with`	Character: pattern to match in the beginning of names of x.
`ends_with`	Character: pattern to match at the end of names of x.
`ignore_case`	Logical: If TRUE, well, ignore case.
`return_index`	Logical: If TRUE, return integer index of matches instead of names.

Details

`pattern`, `starts_with`, and `ends_with` are applied and the union of all matches is returned. `pattern` can be a character vector of multiple patterns to match.

Value

Character vector of matched names or integer index.

Author(s)

EDG

Examples

```
mgetnames(iris, pattern = c("Sepal", "Petal"))
mgetnames(iris, starts_with = "Sepal")
mgetnames(iris, ends_with = "Width")
```

---

mode_value *Mode of a vector*

Description

Returns the most frequent value of a vector.

Usage

```
mode_value(x, na.rm = TRUE, getlast = TRUE, retain_class = TRUE)
```

Arguments

x	Vector, factor or integer: Input data.
na.rm	Logical: If TRUE, exclude NAs (using na.exclude(x)).
getlast	Logical: If TRUE, get the last value in case of ties.
retain_class	Logical: If TRUE, output is always same class as input.

Value

The mode of x

Author(s)

EDG

Examples

```
x <- c(9, 3, 4, 4, 0, 2, 2, NA)
mode_value(x)
x <- c(9, 3, 2, 2, 0, 4, 4, NA)
mode_value(x)
mode_value(x, getlast = FALSE)
```

names_by_class *List column names by class*

Description

List column names by class

Usage

```
names_by_class(x, sorted = TRUE, item_format = highlight, maxlength = 24)
```

Arguments

x	tabular data.
sorted	Logical: If TRUE, sort the output
item_format	Function: Function to format each item
maxlength	Integer: Maximum number of items to print

Value

NULL, invisibly.

Author(s)

EDG

Examples

```
names_by_class(iris)
```

numeric_features *Get numeric features from tabular data*

Description

Returns the numeric columns among the features (all columns except the last).

Usage

```
numeric_features(x)
```

Arguments

x	tabular data: Input data to get numeric features from.

Details

Mirrors `features()`: by **rtemis** convention the last column is the outcome variable and all other columns are features. This drops the outcome, then keeps the numeric features (both double and integer). Useful, for example, to feed only the continuous features to a decomposition: `decomp(numeric_features(iris), ...)`.

Value

Object of the same class as the input, containing only the numeric feature columns.

Author(s)

EDG

Examples

```
numeric_features(iris) |> head()
```

---

one_hot2factor *Convert one-hot encoded matrix to factor*

Description

Convert one-hot encoded matrix to factor

Usage

```
one_hot2factor(x, labels = colnames(x))
```

Arguments

`x`	one-hot encoded matrix or data.frame.
`labels`	Character vector of level names.

Details

If input has a single column, it will be converted to factor and returned

Value

A factor.

Author(s)

EDG

Examples

```
x <- data.frame(matrix(FALSE, 10, 3))
colnames(x) <- c("Dx1", "Dx2", "Dx3")
x[["Dx1"]][1:3] <- x[["Dx2"]][4:6] <- x[["Dx3"]][7:10] <- TRUE
one_hot2factor(x)
```

`outcome` *Get the outcome as a vector*

Description

Returns the last column of `x`, which is by convention the outcome variable.

Usage

```
outcome(x)
```

Arguments

`x`	tabular data.

Details

This applied to tabular datasets used for supervised learning in rtemis, where, by convention, the last column is the outcome variable and all other columns are features.

Value

Vector containing the last column of `x`.

Author(s)

EDG

Examples

```
outcome(iris)
```

outcome_name *Get the name of the last column*

Description

Get the name of the last column

Usage

```
outcome_name(x)
```

Arguments

x	tabular data.

Details

This applied to tabular datasets used for supervised learning in rtemis, where, by convention, the last column is the outcome variable and all other columns are features.

Value

Name of the last column.

Author(s)

EDG

Examples

```
outcome_name(iris)
```

partition *Split a dataset into training and held-out test partitions*

Description

Splits `dat` by the rule in a `PartitionConfig` and returns a manifest of the split, including a `DataFingerprint` of the source and of each partition. With `outdir`, writes `training.parquet` and `test.parquet`, which a `SuperConfig` references through `dat_training_path` and `dat_test_path`.

Usage

```
partition(dat, config, outdir = NULL, overwrite = FALSE, verbosity = 1L)
```

Arguments

dat	data.frame or data.table: The dataset to split.
config	PartitionConfig: The split rule.
outdir	Optional Character: Directory to write training.parquet and test.parquet to; created if it does not exist. If NULL, nothing is written and each outputs entry carries its fingerprint and row count without a path.
overwrite	Logical: If TRUE, overwrite existing output files.
verbosity	Integer: Verbosity level.

Value

Named list: the partition manifest – config, source (a DataFingerprint of dat), and outputs (a list with training and test, each holding a fingerprint and, when outdir is given, the path written to).

Author(s)

EDG

Examples

```
partition(iris, setup_RandomPartition(train_p = 0.8, seed = 2026L))
```

---

plot_learning *Plot a Model's Learning Curve*

Description

Draw training and validation losses recorded by a rtemis Supervised object using the ECharts renderer in rtemis.draw.

Usage

```
plot_learning(x, ...)
```

Arguments

x	rtemis::Supervised: Fitted regression or classification result.
...	Additional arguments to rtemis.draw::draw_learning_curve().

Details

The method uses get_learning_curve() to extract algorithm-specific progress, losses, and the selected step, then rtemis.draw::draw_learning_curve() to render them. Forest curves are averaged at each step over available trees. Models without recorded learning losses produce an informative error. The default title names the algorithm; pass title = NULL to omit it or title = "..." to supply another title.

Value

htmlwidget: ECharts learning curve.

Examples

```
model <- train(
  mtcars[, c("wt", "mpg")],
  hyperparameters = setup_LINAD(max_leaves = 3L),
  verbosity = 0L
)
plot_learning(model)
```

`plot_manhattan` *Plot a MassGLM Result*

Description

`plot(model)` draws a volcano plot using rtemis.draw. `plot_manhattan(model)` draws categorical significance bars, preserving outcome order. Both select one coefficient and use the shared rtemis.draw::SignificanceConfig statistical contract via `rtemis.draw::draw_volcano()` or `rtemis.draw::draw_manhatta`

Usage

```
plot_manhattan(x, coefname = NULL, ...)

plot_manhattan.MassGLM(x, coefname = NULL, ...)

## S3 method for class 'MassGLM'
plot(x, ...)
```

Arguments

`x`	`rtemis::MassGLM`: Fitted mass-univariate model.
`coefname`	Optional Character scalar: Coefficient to plot. Unset selects the first entry in the object's `coefnames`.
`...`	Named settings for `rtemis.draw::draw_volcano()` or `rtemis.draw::draw_manhattan()`.

Details

`plot_manhattan.MassGLM()` is deprecated; use `plot_manhattan()`.

Value

An ECharts htmlwidget.

Examples

```
model <- massGLM(mtcars["wt"], mtcars[c("mpg", "hp", "disp")],
                         verbosity = 0L)
plot(model, coefname = "wt")
plot_manhattan(model, coefname = "wt")
```

`plot_metric` *Plot Stored Metrics Across Model Resamples*

Description

S7 generic for `SupervisedRes` results, including regression and classification. Uses recorded per-resample metrics in `resample_ids` order; list names do not change that producer contract. The default metric is `balanced_accuracy` for classification and `rsq` for regression. Classification selects overall metrics, not per-class scores.

Usage

```
plot_metric(x, ...)
```

Arguments

`x`	rtemis::SupervisedRes: Resampled fitted model.
`...`	Method arguments, including `what` (Character: `"all"` or unique `"training"` and/or `"test"` samples) and `metric` (Optional Character: stored metric name), followed by additional arguments to `rtemis.draw::draw_metric()`.

Details

All available training and test reports are selected by default. An explicitly requested unavailable report is an error. Missing metric values remain missing; a metric absent from every selected report is an error. No predictions or aggregate means are used to reconstruct fold values.

Value

htmlwidget: Metric distributions with all observations by default.

Examples

```
model <- train(mtcars[, c("wt", "mpg")],
  hyperparameters = setup_GLM(),
  outer_resampling_config = setup_KFold(n_resamples = 3L),
  execution_config = setup_SerialExecution(),
  verbosity = 0L
)
plot_metric(model)
```

`plot_roc` *Plot Classification ROC Curves*

Description

S7 generic for ordinary and resampled rtemis classification objects using ECharts. All available training, validation, and test probability samples are selected by default. Resampled results use training and test. An explicit unavailable sample is an error; absence of training probabilities does not hide test data.

Usage

```
plot_roc(x, ...)
```

Arguments

`x`	rtemis classification object: Ordinary or resampled result.
`...`	Method arguments, including `what` (Character: `"all"` or unique sample names), `variant` (Character: `"aggregate"` or `"per_resample"`), `positive` (Optional Character: binary positive class), and `labelify` (Logical: capitalize sample labels), followed by additional named arguments to `rtemis.draw::draw_roc()`.

Details

`variant = "aggregate"` pools aligned prediction rows, retaining repeated observations. `"per_resample"` draws individual curves with mean and sample SD of fold AUCs. Missing probability folds remain undefined curves and are disclosed; they are not zero AUCs. All curves use the same named classes and binary positive class, defaulting to the first fold's second outcome level.

Value

An ECharts htmlwidget.

Examples

```
model <- train(iris, hyperparameters = setup_CART(), verbosity = 0L)
plot_roc(model)
```

`plot_session` *Plot a Supervised object's session timeline*

Description

Plots the session timeline of a `Supervised` or `SupervisedRes` object using `rtemis.draw::draw_gantt()`.

Usage

```
plot_session(x, ...)
```

Arguments

`x`	`rtemis::Supervised` or `rtemis::SupervisedRes` object.
`...`	Additional arguments passed to `rtemis.draw::draw_gantt()`.

Details

Dispatches on ordinary and resampled supervised results.

Value

htmlwidget: Widget object.

Author(s)

EDG

Examples

```
# GLM keeps the example dependency-free: rtemis fits it with stats::glm().
  mod <- train(
    mtcars[, c("wt", "hp", "mpg")],
    hyperparameters = setup_GLM(),
    verbosity = 0L
  )
  plot_session(mod)
```

`plot_true_pred` *Plot a Model's True Versus Predicted Values*

Description

S7 generic for ECharts plots of rtemis regression and classification results.

Usage

```
plot_true_pred(x, ...)
```

Arguments

`x`	rtemis regression or classification object: Ordinary or resampled result.
`...`	Method arguments, including `what` (Optional Character: `"all"` or a vector of `"training"`, `"validation"`, `"test"` sample names) and `labelify` (Logical: capitalize sample names in the legend or panel titles), followed by additional arguments to `rtemis.draw::draw_fit()` or `rtemis.draw::draw_confusion()`. Resampled results do not have validation pairs.

Details

`what = "all"` selects available training, validation, and test pairs for a regression model, or training and test pairs for a resampled model. An explicitly requested missing sample is an error. Resampled observations are pooled in fold order, keeping repeated observations. Pair lengths and names are checked within each fold before pooling.

Rendering is provided by `rtemis.draw::draw_fit()`. The API uses draw argument names, including `se`, `square`, `equal_axes`, and `title`, and a draw rtemis.draw::Theme. Supported fits are `"glm"`, `"gam"`, or `NULL`.

Classification uses `rtemis.draw::draw_confusion()` with stored confusion frequencies when available, preserving the producer's class order. Otherwise it counts aligned reference/predicted pairs. An omitted `what` selects test, validation, then training for an ordinary classifier, or all available samples for a resampled classifier. Multiple samples share one widget and one export. Balanced accuracy in this view is macro recall, not the multiclass average of one-versus-rest balanced accuracies. See rtemis.draw::ConfusionConfig for zero-denominator and pooled-count semantics.

Value

htmlwidget: ECharts drawing.

Examples

```
model <- train(
  mtcars[, c("wt", "hp", "mpg")],
  hyperparameters = setup_GLM(), verbosity = 0L
)
plot_true_pred(model)
```

plot_varimp *Plot a Model's Variable Importance*

Description

Draw importance recorded by a rtemis Supervised or SupervisedRes object, or computed from a SHAP explanation by get_varimp, using rtemis.draw::draw_varimp().

Usage

```
plot_varimp(x, ...)
```

Arguments

x	rtemis::Supervised, rtemis::SupervisedRes or rtemis::SHAP: Fitted model result or explanation.
...	Additional arguments to rtemis.draw::draw_varimp(), including measure, type, top_n, rank_by, decreasing, bar_width, summary, and title. Fold IDs come from the model and cannot be overridden.

Details

Named measures and variable identities are preserved. Resampled records retain their fold IDs before summarizing. The first measure in the first available fold is selected when measure is omitted; columns missing in other folds are unavailable scores. The full fold list is passed to the renderer, including folds with no importance result.

Every producer reports a value for each predictor it scores, including zeros, so an NA value or a wholly unavailable fold is an unavailable score and is excluded from the summary. varimp_ranking() returns the summary and ranking drawn here.

Use type = "boxplot" for resampled fold distributions with all contributing scores overlaid. Summary bars remain the default. Both views share selection, ranking, and missing-score handling. Ordinary models have no fold distribution and require the bar view.

A SHAP explanation is drawn from the mean absolute contribution of each feature, with one measure per class for a multiclass outcome.

Models without any importance records produce an informative error. The default title names the algorithm; title = NULL omits it. See rtemis.draw::draw_varimp() for selection, ranking, and missing-score semantics.

Value

htmlwidget: ECharts importance bars or fold distributions.

Examples

```
model <- train(
  mtcars[, c("wt", "hp", "mpg")],
  hyperparameters = setup_GLM(), verbosity = 0L
)
plot_varimp(model)
features <- mtcars[, c("wt", "hp")]
plot_varimp(explain(model, features, background = features, verbosity = 0L))
```

`preprocess` *Preprocess Data*

Description

Preprocess data for analysis and visualization.

Usage

```
preprocess(
  x,
  config,
  dat_validation = NULL,
  dat_test = NULL,
  verbosity = 1L,
  ...
)
```

Arguments

`x`	Tabular data, i.e. data.frame, data.table, or tbl_df (tibble): Training set data to preprocess.
`config`	`PreprocessorConfig`: Preprocessing configuration created by setup_Preprocessor. May be omitted, in which case setup_Preprocessor arguments are passed directly via `...`.
`dat_validation`	Optional tabular data: Validation set data. Preprocessed using the values learned from the training set data.
`dat_test`	Optional tabular data: Test set data. Preprocessed using the values learned from the training set data.
`verbosity`	Integer: Verbosity level.
`...`	setup_Preprocessor arguments: Only used when `config` is not provided.

Details

`preprocess()` preprocesses training data and learns any data-dependent values (e.g. scale centers and coefficients, one-hot levels). Optional `dat_validation` and `dat_test` data are preprocessed using the values learned from the training data. To apply a trained `Preprocessor` to new data, use apply_preprocessor.

For interactive use, `config` may be omitted and setup_Preprocessor arguments passed directly instead, e.g. `preprocess(x, scale = TRUE, center = TRUE)`. At least one preprocessing parameter must be specified: `preprocess(x)` is an error.

The preprocessed data comes back in the structure it was given – data.frame, data.table or tibble – and the object passed in is never modified. It carries no row names; case identifiers are carried in columns.

Value

`Preprocessor` object.

Author(s)

EDG

See Also

apply_preprocessor, setup_Preprocessor

Examples

```
# Setup a `Preprocessor`: this outputs a `PreprocessorConfig` object.
prp <- setup_Preprocessor(remove_duplicates = TRUE, scale = TRUE, center = TRUE)

# Includes a long list of parameters
prp

# Resample iris to get train and test data
res <- resample(iris, setup_KFold(seed = 2026))
iris_train <- iris[res[[1]], ]
iris_test <- iris[-res[[1]], ]

# Preprocess training data
iris_pre <- preprocess(iris_train, prp)

# Alternatively, for interactive use, pass `setup_Preprocessor()` arguments directly
iris_pre <- preprocess(iris_train, remove_duplicates = TRUE, scale = TRUE, center = TRUE)

# Access preprocessed training data with `preprocessed()`
preprocessed(iris_pre)

# Apply the same preprocessing to test data with `apply_preprocessor()`,
# which returns the preprocessed data directly.
# The scale and center values learned from the training data will be used.
iris_test_pre <- apply_preprocessor(iris_pre, iris_test)
```

preprocessed *Get preprocessed data from* Preprocessor.

Description

Returns the preprocessed data from a Preprocessor object.

Usage

```
preprocessed(x)
```

Arguments

x	Preprocessor: A Preprocessor object.

Details

A data.table is returned as a copy made with data.table::copy(), so modifying the result by reference, with := or set(), leaves x unchanged.

Value

The preprocessed data, as a data.frame, data.table, or tibble, matching the data passed to preprocess(). When validation or test data were also preprocessed, a named list with elements training, validation, and test, as supplied.

Author(s)

EDG

Examples

```
prp <- preprocess(iris, setup_Preprocessor(scale = TRUE, center = TRUE))
preprocessed(prp)
```

present *Describe and Plot Fitted Models*

Description

A single model prints its describe() output, unless verbosity = 0, and returns its default plot() view: true/predicted values for regression, confusion matrices for classification, and stored metric distributions for resampled models. plot_roc() draws ROC curves for classification models with predicted probabilities.

Usage

```
present(x, ...)
```

Arguments

x	rtemis model or List: Fitted ordinary/resampled models.
...	Arguments for the selected presentation: what, verbosity, and drawing arguments. Resampled models accept metric. Lists accept metric, model_names, what, ylim, boxpoints, quartiles, whisker, palette, theme, width, height, and filename. Single models forward drawing arguments to their plot() method. Comparison lists use training/test samples; individual ordinary models also support validation. Explicit ylim is a finite increasing pair for comparisons.

Details

A nonempty homogeneous list compares models of the same task and sampling kind. Ordinary models use grouped metric bars. Resampled models use one boxplot panel per sample with common value limits and aligned model labels. Fold counts may differ; fold scores are neither paired nor pooled across models. Missing scores stay missing. Wholly unavailable samples are omitted under what = "all"; explicit unavailable samples are errors.

Comparison labels use model_names, complete list names, or algorithm names, made unique in that order. A data-fingerprint mismatch notice is printed, including when descriptions are disabled.

Value

htmlwidget: One chart or one complete rtemis.draw::draw_panels() figure.

Default plots

plot() on a fitted model draws the view present() returns, without the description:

- Regression: plot_true_pred() for all available samples.
- Classification: confusion matrices for all available samples, as drawn by plot_true_pred().
- RegressionRes and ClassificationRes: plot_metric() for all available samples.

Arguments after x are passed to the corresponding plot_*() method, for example plot(model, what = "test") or plot(model, metric = "rmse").

Model plots color each data split the same way: training teal, validation purple, test magenta. A supplied palette replaces these colors. Confusion matrices shade correct predictions blue and incorrect predictions orange.

Examples

```
model <- train(mtcars[, c("wt", "mpg")],
  hyperparameters = setup_GLM(), verbosity = 0L)
present(model, verbosity = 0L)
plot(model)
```

read *Read tabular data from a variety of formats*

Description

Read data and optionally clean column names, keep unique rows, and convert characters to factors

Usage

```
read(
  filename,
  datadir = NULL,
  remove_duplicates = FALSE,
  character2factor = FALSE,
  clean_colnames = TRUE,
  delim_reader = c("data.table", "vroom", "duckdb", "arrow"),
  parquet_reader = c("nanoparquet", "arrow"),
  xlsx_sheet = 1,
  sep = NULL,
  quote = "\"",
  na_strings = c(""),
  output = c("data.table", "tibble", "data.frame"),
  attr = NULL,
  value = NULL,
  verbosity = 1L,
  fread_verbosity = 0L,
  timed = verbosity > 0L,
  ...
)
```

Arguments

filename	Character: filename, full path if datadir = NULL, an http:// or https:// URL, or an hf://datasets/ path.
datadir	Character: Optional path to directory where filename is located. If not specified, filename must be the full path. Must be NULL when filename is a URL or an hf:// path.
remove_duplicates	Logical: If TRUE, keep unique rows only.
character2factor	Logical: If TRUE, convert character variables to factors.
clean_colnames	Logical: If TRUE, clean columns names using clean_colnames.
delim_reader	Character: package to use for reading delimited data.
parquet_reader	Character: package to use for reading Parquet files. If undefined and system is WASM, uses "nanoparquet", otherwise "arrow".

xlsx_sheet	Integer or character: Name or number of XLSX sheet to read.
sep	Single character: field separator. If delim_reader = "fread" and sep = NULL, this is "auto", otherwise ",".
quote	Single character: quote character.
na_strings	Character vector: Strings to be interpreted as NA values. For delim_reader = "duckdb", this must be a single string.
output	Character: "default" or "data.table", If default, return the delim_reader's default data structure, otherwise convert to data.table.
attr	Character: Attribute to set (Optional).
value	Character: Value to set (if attr is not NULL).
verbosity	Integer: Verbosity level.
fread_verbosity	Integer: Verbosity level. Passed to data.table::fread
timed	Logical: If TRUE, time the process and print to console
...	Additional arguments to pass to data.table::fread, arrow::read_delim_arrow(), vroom::vroom(), or readxl::read_excel().

Details

read is a convenience function to read:

- **Delimited** files using data.table::fread(), arrow::read_delim_arrow(), vroom::vroom(), or duckdb::duckdb_read_csv()
- **ARFF** files using farff::readARFF()
- **Parquet** files using nanoparquet::read_parquet or arrow::read_parquet()
- **XLSX** files using readxl::read_excel()
- **DTA** files from Stata using haven::read_dta()
- **FASTA** files using seqinr::read.fasta()
- **RDS** files using readRDS()

An http:// or https:// URL is downloaded with utils::download.file() to a temporary file named after the last segment of the URL path, read like a local file, and deleted. The file extension in the URL path selects the reader, as it does for a local file. The download is bounded by getOption("timeout"), in seconds. A URL with user credentials (user:password@host) and any scheme other than http, https, and hf is rejected.

An hf://datasets/<owner>/<name>[@<revision>][/<file>] path reads a file from a public dataset repository on the Hugging Face Hub, in the path form of huggingface_hub's HfFileSystem. The revision is a branch, tag, or commit, main when none is given; a revision containing / is written URL-encoded, as refs%2Fconvert%2Fparquet. A path that names the repository only reads the repository's one data file (.parquet, .csv, .tsv, .csv.gz, .tsv.gz, .rds, .xlsx, .dta, .arff, or .fasta) and is an error when the repository holds more than one. The revision is resolved to a commit through the Hub API, the file is downloaded at that commit, and its size and hash are checked against the Hub's listing. The HF_ENDPOINT environment variable sets the Hub endpoint. Reading hf:// paths requires the jsonlite package.

Value

data.frame, data.table, or tibble.

Author(s)

EDG

Examples

```
## Not run:
# Replace with your own data directory and filename
datadir <- "/Data"
dat <- read("iris.csv", datadir)
# Read from a URL
dat <- read("https://example.org/data/iris.csv")
# Read from a Hugging Face dataset
dat <- read("hf://datasets/rtemis/tcga-pancan-hiseq")

## End(Not run)
```

---

read_config *Read an rtemis config from file*

Description

Read a schema.rtemis.org JSON config and return the appropriate rtemis object. Two families of config are supported: pipeline recipes that bundle a data path, algorithm config, and output directory (`SuperConfig`, `DecomposeConfig`, `ClusterConfig`), and the bare algorithm configs they wrap (`DecompositionConfig`, `ClusteringConfig`). The file is JSON, as written by write_config and consumed by rtemislive and the `rtemis` CLI.

Usage

```
read_config(file)
```

Arguments

`file`	Character: Path to input JSON config file.

Details

The config family is taken from the instance's `$schema`, which must be one of the supported schemas; a missing or unrecognized `$schema` is an error.

A **run record** is rejected, and named as such. A record declares the same fields as the config it came from, so it would read as one – but every value in it is resolved, including values a run derived from data this call has never seen. Accepting it would silently pin settings the new call should decide for itself. Records are written by write_record and identify themselves with a `record.json` schema.

Validation is entirely rtemis's own: `check_wire_keys()` rejects an unknown key at every level of the document, and each `setup_*` then constructs an S7 object whose properties enforce their declared types, bounds, enums and arities. Those properties are the same `PropertySpecs` the published JSON Schemas are generated from, so reading a config in R applies the published contract by construction.

Value

A `SuperConfig`, `DecomposeConfig`, `ClusterConfig`, `DecompositionConfig`, or `ClusteringConfig` object.

Author(s)

EDG

Examples

```
# Create a SuperConfig object
x <- setup_SuperConfig(
  dat_training_path = "~/Data/iris.csv",
  hyperparameters = setup_LightRF()
)
# Write JSON config file
tmpdir <- tempdir()
tmpfile <- file.path(tmpdir, "rtemis_test.json")
write_config(x, tmpfile, overwrite = TRUE)
# Read config back from JSON file
x_read <- read_config(tmpfile)
```

---

read_result *Read a portable result from a local schema registry*

Description

Reconstruct the report's classes from the supplied schema and defaults artifacts. Parsing does not open referenced files. With `load_data = TRUE`, verify file integrity and dimensions, then validate the materialized values. Native fitted objects are not restored.

Usage

```
read_result(file, registry, load_data = FALSE)
```

Arguments

`file`	Character: Result JSON path.
`registry`	Character: Local registry directory containing versioned schema documents and `defaults/v1/defaults.json`.
`load_data`	Logical: Materialize Parquet references.

Value

An S7 report reconstructed from registry declarations.

Examples

```
# A local registry is supplied explicitly; reading never fetches schemas.
reader <- function(file, registry) read_result(file, registry, load_data = TRUE)
```

`reconstruct` *Reconstruct Data from a Decomposition*

Description

Encode data with a fitted decomposition and decode it back, giving the rank-k approximation of the input in its original units and column layout.

Usage

```
reconstruct(decom, x, execution_config = NULL, verbosity = 1L)
```

Arguments

`decom`	`Decomposition` object.
`x`	Tabular data (data.frame, data.table, or tibble): Data to reconstruct. Its columns must match those `decom` was fitted on.
`execution_config`	Optional `ExecutionConfig` object: Threads for encoding `x`, as in `apply_decomposition()`. `NULL` uses the host's default worker count.
`verbosity`	Integer: Verbosity level.

Details

The round trip `apply_decomposition()` leaves half-done. The reconstruction is `x` with everything outside the first `k` components removed, i.e. a denoised `x`. The residual, `x - reconstruct(decom, x)`, is what the decomposition could not represent: `rowMeans(residual^2)` scores how anomalous each case is, and `colMeans(residual^2)` shows which variables the decomposition fails to capture.

Available for algorithms whose `invertible` trait is TRUE; see decomposition_traits. tSNE, UMAP and Isomap embed without an inverse, so there is no map back for them at all.

The reconstruction is in the units of the data as it was passed to decomp, with any centering or scaling the backend applied internally undone, so it can be compared with `x` directly. Columns the fit did not decompose (when `decom@config@features` is set) pass through unchanged and column order is preserved, so the result lines up cell for cell with `x`.

Value

A data.frame with the same columns as `x`, holding the reconstruction of the decomposed features.

Author(s)

EDG

Examples

```
x <- exc(iris, "Species")
iris_pca <- decomp(x, algorithm = "PCA", config = setup_PCA(k = 2L))
reconstructed <- reconstruct(iris_pca, x)
# What the two components could not represent, per case:
head(rowMeans((as.matrix(x) - as.matrix(reconstructed))^2))
```

---

record *Derive a run record from a fitted model*

Description

The record of what ran: every config value resolved, an `origin` saying where each came from, and a provenance block. Derived on demand from the configs the object already holds.

Usage

```
record(x, ...)
```

Arguments

`x`	Fitted model object.
`...`	Passed to methods.

Value

Named list, conforming to the family's `record.json`.

Author(s)

EDG

Examples

```
mod <- train(iris, hyperparameters = setup_CART())
names(record(mod))
```

regression_metrics *Regression Metrics*

Description

Regression Metrics

Usage

```
regression_metrics(true, predicted, na.rm = TRUE, sample = NULL)
```

Arguments

true	Numeric vector: True values.
predicted	Numeric vector: Predicted values.
na.rm	Logical: If TRUE, remove NA values before computation.
sample	Optional Character {"Training", "Validation", "Test", "Calibrated Training", "Calibrated Validation", "Calibrated Test"}: Sample the metrics describe.

Value

RegressionMetrics object.

Author(s)

EDG

Examples

```
true <- rnorm(100)
predicted <- true + rnorm(100, sd = 0.5)
regression_metrics(true, predicted)
```

resample *Resample data*

Description

Create resamples of data, e.g. for model building or validation. "KFold" creates stratified folds, "StratSub" creates stratified subsamples, "Bootstrap" gives the standard bootstrap, i.e. random sampling with replacement, while "StratBoot" uses StratSub and then randomly duplicates some of the training cases to reach the original length of the input, or the length defined by target_length.

Usage

```
resample(x, config = setup_KFold(), verbosity = 1L)
```

Arguments

x	Vector or data.frame: Usually the outcome; NROW(x) defines the sample size. A config naming a column – stratify_var or id_strat – needs the data frame that column lives in, since a name cannot be looked up in a bare vector.
config	Resampler object created by one of the setup_* resampler functions, e.g. setup_KFold.
verbosity	Integer: Verbosity level.

Details

'KFold' can produce resamples of slightly different length, so resamples are returned as a list, not a data.frame.

Value

Resampler object.

Author(s)

EDG

Examples

```
y <- rnorm(200)
# 10-fold (stratified)
y_10fold <- resample(y, setup_KFold(10L))
y_10fold
# 25 stratified subsamples
y_25strat <- resample(y, setup_StratSub(25L))
y_25strat
# 100 stratified bootstraps
y_100strat <- resample(y, setup_StratBoot(100L))
y_100strat
# LOOCV
y_loocv <- resample(y, setup_LOOCV())
y_loocv
# User-supplied resamples
y_custom <- resample(
  y,
  setup_Custom(resamples = list(1:150, 51:200))
)
y_custom
```

review *Review a trained supervised model*

Description

Assess a trained model: whether its evaluation can be trusted given the sample size and the number of predictors, whether it performs better than a baseline that ignores the predictors, and whether it overfits. The review is deterministic: it draws no random numbers.

Usage

```
review(x, confidence_level = NULL, min_cases_per_predictor = NULL, ...)
```

Arguments

`x`	`Supervised` or `SupervisedRes` object: A trained model, as returned by train.
`confidence_level`	Optional Numeric (0, 1): Confidence level of every interval. NULL uses 0.95.
`min_cases_per_predictor`	Optional Numeric (0, Inf): Training cases per learner column – minority-class cases for classification – below which the review notes a shortfall. NULL uses 10, an events-per-variable rule of thumb from logistic regression (see References).
`...`	Not used.

Details

The review holds every value it rests on: `@sample` (sample sizes, input predictors and the columns the learner received), `@class_counts`, `@performance` (one row per metric: training, test, training minus test, the spread over resamples, and the test interval or pooled value), `@baseline` (one row per comparison with a reference that ignores the predictors, naming the reference and the method) and `@tuning` (tuned hyperparameters against their search range). Its findings state what it observes. Printing shows a summary: one row per metric – mean (SD) over resamples for a resampled model – the baseline comparisons, the findings and the limitations.

Single split. The test cases are independent of the fitted model, so the review computes confidence intervals and two-sided comparisons at `confidence_level`. Accuracy is compared with always predicting the most common training class by the exact McNemar test, which pairs the two predictors' results case by case. Balanced accuracy and AUC are compared with their chance levels (1/K, 0.5) through their intervals: per-class binomial variances for balanced accuracy, the DeLong method for AUC. The Brier score, MSE and MAE are compared with a constant baseline (training proportion, training mean) through a paired t interval of the per-case loss reduction; the skill score is reported as a point estimate. An interval that cannot be computed – too few cases, a class absent from the test set, an AUC of 0 or 1 – is left unset and no verdict is made from it. A training value of the headline metric (balanced accuracy, or mean squared error for regression) outside its test interval is reported as a diagnostic sign of possible overfitting; it is not a test of the train-test difference.

Resampled models. Resamples share training cases, so their test results are dependent, and the review makes no interval or test from them. Every metric is reported as its mean and standard deviation over the outer resamples, each baseline is fit to the training data of its resample, and the review counts the resamples in which the model beat its baseline. When every case is tested once, as with k-fold resampling, pooled out-of-sample values are reported as descriptions for metrics that average over cases; AUC is not pooled, since it would rank scores from different fitted models together.

A tuned hyperparameter selected at the edge of the values searched is reported, since a better value may lie beyond it.

The cases-per-predictor check counts the columns the learner received, after preprocessing and decomposition. Its threshold is a rule of thumb from logistic regression (events per variable; see References), reported for context.

Preprocessing, decomposition and tuning inside `train()` are fitted on the training cases of each split and only applied to its test cases. Steps taken before the data is passed to `train()` – selecting predictors or transforming cases using all the data, or choosing among models by their test performance – can bias evaluation, and the fitted model cannot show whether they happened. Performance metrics also cannot establish whether a model is useful. Every review states both.

Print the result to read it; write_text writes it as Markdown.

Value

`SupervisedReview` object, whose tables are data.frames.

Author(s)

EDG

References

Clopper CJ, Pearson ES (1934). The use of confidence or fiducial limits illustrated in the case of the binomial. Biometrika, 26(4), 404-413.

DeLong ER, DeLong DM, Clarke-Pearson DL (1988). Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach. Biometrics, 44(3), 837-845.

McNemar Q (1947). Note on the sampling error of the difference between correlated proportions or percentages. Psychometrika, 12(2), 153-157.

Peduzzi P, Concato J, Kemper E, Holford TR, Feinstein AR (1996). A simulation study of the number of events per variable in logistic regression analysis. Journal of Clinical Epidemiology, 49(12), 1373-1379.

Examples

```
idx <- c(1:40, 51:90, 101:140)
mod <- train(
  iris[idx, ],
  dat_test = iris[-idx, ],
  hyperparameters = setup_CART(),
  verbosity = 0L
```

```
)
review(mod)

# Resampled
mod_res <- train(
  iris,
  hyperparameters = setup_CART(),
  outer_resampling_config = setup_KFold(5L),
  verbosity = 0L
)
review(mod_res)
```

`rnormmat` *Random Normal Matrix*

Description

Create a matrix or data frame of defined dimensions, whose columns are random normal vectors

Usage

```
rnormmat(
  nrow = 10,
  ncol = 10,
  mean = 0,
  sd = 1,
  return_df = FALSE,
  seed = NULL
)
```

Arguments

`nrow`	Integer: Number of rows.
`ncol`	Integer: Number of columns.
`mean`	Float: Mean.
`sd`	Float: Standard deviation.
`return_df`	Logical: If TRUE, return data.frame, otherwise matrix.
`seed`	Integer: Set seed for `rnorm`.

Value

`matrix` or `data.frame`.

Author(s)

EDG

Examples

```
x <- rnormmat(20, 5, mean = 12, sd = 6, return_df = TRUE, seed = 2026)
x
```

`roc_curve` *ROC curve coordinates*

Description

Compute the points of one or more ROC curves from true labels and predicted probabilities, returning tidy `(class, fpr, tpr, auc)` rows. The rows serve any consumer of the curve data, such as a client that renders the curve.

Usage

```
roc_curve(true_labels, predicted_prob, max_points = NULL)
```

Arguments

`true_labels`	Factor: True outcome labels.
`predicted_prob`	Numeric vector or matrix [0, 1]: Predicted probabilities for the positive class (second level of `true_labels`), as a vector or a one-column matrix. For multiclass, a matrix with exactly one column per class, in factor-level order (rtemis's convention); the columns are labeled with the factor levels regardless of any names on the input.
`max_points`	Optional Integer [2, Inf): Cap on the number of vertices per curve. `NULL` keeps full resolution.

Details

Binary problems yield a single curve for the positive class (the second level of `true_labels`, rtemis convention); multiclass problems yield one one-vs-rest curve per class. Points are ordered along the curve; pass `max_points` to down-sample very long curves (one vertex per distinct score) to a compact, smooth line.

Value

`data.frame` with columns `class`, `fpr`, `tpr`, `auc` (the AUC is constant within a class group, repeated per vertex). Zero rows when no curve is defined.

Author(s)

EDG

Examples

```
true_labels <- factor(c("A", "B", "A", "A", "B", "A", "B", "B", "A", "B"))
predicted_prob <- c(0.1, 0.4, 0.35, 0.8, 0.65, 0.2, 0.9, 0.55, 0.3, 0.7)
roc_curve(true_labels, predicted_prob)
```

roc_table *ROC Curves of a Classification Model*

Description

The ROC curves of a fitted classification model for each sample it stores predicted probabilities for: the data `plot_roc()` draws, as a table.

Usage

```
roc_table(
  x,
  sample = NULL,
  variant = "aggregate",
  positive = NULL,
  max_points = NULL
)
```

Arguments

`x`	`Classification` or `ClassificationRes` object.
`sample`	Optional Character {"training", "validation", "test"}: The sample whose curves to return. `NULL` returns every sample with predicted probabilities.
`variant`	Character {"aggregate", "per_resample"}: For a resampled model, pooled curves or one curve per resample. An ordinary model has aggregate curves only.
`positive`	Optional Character: Positive class of a binary model. `NULL` uses the second level of the outcome.
`max_points`	Optional Integer [2, Inf): Cap on the number of vertices per curve. `NULL` keeps every vertex.

Details

Binary models have one curve, for the positive class; multiclass models have one one-vs-rest curve per class. For a resampled model, `variant = "aggregate"` pools the predictions of every outer resample into one curve per sample and class, and `variant = "per_resample"` gives one curve per resample, labeled in the `resample` column. A resample whose probabilities are missing has an undefined curve, with `auc` NA. `omitted` counts the cases whose probability or label is missing.

`roc_curve()` computes the curve of supplied labels and probabilities.

Value

data.table with columns `sample`; `resample`, for `variant = "per_resample"`; `class`; `fpr`; `tpr`; `auc`, constant within a curve; and `omitted`.

Author(s)

EDG

Examples

```
mod <- train(iris, hyperparameters = setup_CART(), verbosity = 0L)
roc_table(mod)
```

`rtemis_conditions` *rtemis error conditions*

Description

Errors signalled by **rtemis** are classed R conditions, so they can be handled selectively with `tryCatch()` or `withCallingHandlers()` by class. Each error carries a class vector ordered from the most specific failure mode to the most general, so a handler can catch narrowly (e.g. only type errors) or broadly (e.g. any invalid input).

Every rtemis error inherits, in order: its specific class (when applicable), its category class, and finally `"rtemis_error"`, `"error"`, `"condition"`. Catching `rtemis_error` therefore catches *all* rtemis errors.

Value

Not applicable. This page documents the condition classes; the functions that signal them do not return when they error.

Categories and specific classes

`rtemis_input_error`**:**

A function argument supplied by the caller is invalid. Specific classes:

`rtemis_type_error` Argument is of the wrong type or class.

`rtemis_value_error` Argument has a disallowed value or choice.

`rtemis_length_error` Argument has the wrong length or count.

`rtemis_range_error` Argument is outside its allowed range.

`rtemis_null_input` A required argument is `NULL` or empty.

`rtemis_data_error`**:**

The supplied dataset is unsuitable for the requested operation. Specific classes:

`rtemis_dim_error` Data has the wrong dimensions (e.g. too few columns).

`rtemis_level_mismatch` Factor levels do not match across data splits.

`rtemis_missing_data` Data contains missing values where none are allowed.

`rtemis_outcome_error` The outcome column has an invalid or mismatched type.

`rtemis_io_error`**:**

A filesystem operation failed. Specific classes:

`rtemis_file_not_found` A path or file does not exist.

`rtemis_file_exists` A file already exists and overwriting was not permitted.

`rtemis_unsupported_error`:
The requested algorithm, feature, or combination is not supported (e.g. multiclass classification for an algorithm that only handles binary outcomes). No specific subclass.

`rtemis_runtime_error`:
A model or algorithm call failed during execution. The original error is attached as the condition's `$parent`. No specific subclass.

Author(s)

EDG

See Also

rtemis-package

Examples

```
# Every rtemis error inherits `rtemis_error`. Catch a category broadly:
result <- tryCatch(
  set_outcome(iris, "nonexistent_column"),
  rtemis_input_error = function(e) {
    message("Caught an input error: ", conditionMessage(e))
    NULL
  }
)

# Or catch a specific failure mode only:
result <- tryCatch(
  set_outcome(iris, "nonexistent_column"),
  rtemis_value_error = function(e) {
    message("Caught a value error: ", conditionMessage(e))
    NULL
  }
)
```

---

rtversion *Get rtemis version and system info*

Description

Get rtemis version and system info

Usage

```
rtversion()
```

Value

List: rtemis version and system info, invisibly.

Author(s)

EDG

Examples

```
rtversion()
```

`runifmat` *Random Uniform Matrix*

Description

Create a matrix or data frame of defined dimensions, whose columns are random uniform vectors

Usage

```
runifmat(
  nrow = 10,
  ncol = 10,
  min = 0,
  max = 1,
  return_df = FALSE,
  seed = NULL
)
```

Arguments

`nrow`	Integer: Number of rows.
`ncol`	Integer: Number of columns.
`min`	Float: Min.
`max`	Float: Max.
`return_df`	Logical: If TRUE, return data.frame, otherwise matrix.
`seed`	Integer: Set seed for `rnorm`.

Value

`matrix` or `data.frame`.

Author(s)

EDG

Examples

```
x <- runifmat(20, 5, min = 12, max = 18, return_df = TRUE, seed = 2026)
x
```

`S7_dispatcher_JSONSchema` *Generate a config family's dispatcher JSON Schema*

Description

Assembles the `<family>/v1` schema for a config family: an object holding the discriminator (`algorithm`, `type`, ...), the properties the family base declares for every variant, and an `allOf` of `if/then` clauses that apply the selected variant's leaf schema (`<family>/<variant>/v1`) to the whole object. The discriminator enum, the leaf `$ref` URLs and the `allOf` table are all derived from the classes.

Usage

```
S7_dispatcher_JSONSchema(
  classes,
  id,
  discriminator = "algorithm",
  base = NULL,
  title = NULL,
  description = "",
  discriminator_description = "Algorithm name.",
  record = FALSE,
  instance_schema_url = NULL
)
```

Arguments

`classes`	List of S7 classes: the family's per-variant subclasses (each carries a computed constant discriminator property).
`id`	Character: Dispatcher `$id` URL (e.g. "https://schema.rtemis.org/decomposition/r/v1/schema.json").
`discriminator`	Character: Name of the property that selects the variant (e.g. "algorithm", "type").
`base`	Optional S7 class: The family base class. Its own `prop_*`-declared properties are shared by every variant, so they are emitted here, once (see Details).
`title`	Optional Character: Schema title.
`description`	Character: Schema description. If empty, omitted. The shape rule – settings are siblings of the discriminator – is appended, so every family states it the same way.
`discriminator_description`	Character: Description of the discriminator property.

`record` Logical: If TRUE, dispatch to the variants' **record** schemas (`<family>/<variant>/v1/record.json`), and require every property the dispatcher itself declares. The discriminator is required either way.

`instance_schema_url` Character or NULL: If set, adds a `$schema` const property so instances can self-identify.

Details

The leaf URLs are derived from `id`: for a dispatcher `.../<family>/v1/schema.json`, variant `A` maps to `.../<family>/<tolower(A)>/v1/schema.json` – matching S7_to_JSONSchema's `id` convention for the leaves.

One shape, matching how every family serializes (`dispatched_props()`): the variant's settings are **siblings of the discriminator**. A document holding only the discriminator is that variant with every default: "nothing set" is expressed by absence.

Each `then` is exactly the leaf `$ref`, applied to the whole object; readers key on that. `additionalProperties` is evaluated per schema and would not see the leaf's properties, so strictness comes from draft 2020-12's `unevaluatedProperties: false`, which does account for properties evaluated by the applied `$ref`. Leaves must therefore be generated open (`closed = FALSE` in S7_to_JSONSchema) so they compose.

The leaf's keys and the dispatcher's are disjoint by construction: `base` subtracts every base property, the discriminator included, from each leaf.

`base` pairs with S7_to_JSONSchema's `base` argument, which subtracts the family base's properties from every leaf: the dispatcher adds them back at the top level, from the same `PropertySpec`, so the shared fields are declared once. Base properties carrying no spec are class machinery (a computed settings list, run state) and are skipped, as is the discriminator, which is generated from the variant enum.

Value

Named list: the dispatcher JSON Schema. Serialize with write_JSONSchema.

Author(s)

EDG

Examples

```
## Not run:
schema <- S7_dispatcher_JSONSchema(
  classes = list(PCAConfig, ICAConfig),
  id = "https://schema.rtemis.org/decomposition/r/v1/schema.json",
  base = DecompositionConfig
)

## End(Not run)
```

S7_to_JSONSchema *Convert an S7 class built with* prop_* *factories to a JSON Schema*

Description

Walks the class's properties, reads each attached PropertySpec, and assembles a draft 2020-12 JSON Schema. Which properties take part is decided by their declared role (see prop_role()), not by a list kept here: "config" and "state" properties are generated from their spec, state being marked readOnly, while "computed" and "r_only" properties are omitted: the first is derived from published fields, and the second has no wire form. A spec-less property with no role is an error.

Usage

```
S7_to_JSONSchema(
  x,
  id,
  title = NULL,
  description = "",
  base = NULL,
  required = NULL,
  record = FALSE,
  asserted = FALSE,
  provenance_url = NULL,
  session_url = NULL,
  fold_refs = NULL,
  metrics_refs = NULL,
  metrics_ref = NULL,
  reference_urls = NULL,
  closed = TRUE,
  instance_schema_url = NULL
)
```

Arguments

x	S7 class (e.g. LightRFHyperparameters).
id	Character: Schema $id URL (e.g. "https://schema.rtemis.org/hyperparameters/r/lightrf/v1/schema.json")
title	Character: Schema title. Defaults to the class name.
description	Character: Schema description. If empty, the "description" keyword is omitted from the schema.
base	S7 class or NULL: The family base class, whose inherited properties are machinery (tuned, resampled, the computed payload list) and are omitted. NULL for a flat config that has no family base.
required	Character: Names of required properties. Default NULL: all optional, so omitted fields fall back to their setup_* defaults on read (matching write_config's compaction). Ignored when record is TRUE.

`record` Logical: If TRUE, emit the **record** form of the schema: the same properties, but every one required. A record states what a run used, so every field is present, and an unset value is written as an explicit `null`. Input and record schemas have identical membership.

`asserted` Logical: If TRUE, emit an **assertion** schema: the input form, but with every property required. For a results class: a document rtemis produces, with no record sibling. Mutually exclusive with `record`.

`provenance_url` Character or NULL: If set (and `record` is TRUE), adds a required `provenance` property referencing that schema. Only a top-level record carries it; a nested one inherits its parent's.

`session_url` Character or NULL: If set (and `record` is TRUE), adds a required, nullable `session` property referencing that schema – the reference to the execution graph written beside the record. Nullable because a run can produce no session: a pipeline result records none.

`fold_refs` Named character or NULL: If set (and `record` is TRUE), adds a required `folds` array whose entries reference these record schemas. Only a record of a run that fits models per resample carries one.

`metrics_refs` Named character or NULL: If set (and `record` is TRUE), adds the required `metrics` / `metrics_sd` headline blocks, and references these metrics schemas from each fold's own `metrics`.

`metrics_ref` Character or NULL: If set (and `record` is TRUE), adds a required, nullable `metrics` property referencing that one schema. For a run whose result is a single metrics object; mutually exclusive with `metrics_refs`.

`reference_urls` Optional named character: Publication URLs keyed by qualified class identity. If omitted, derived from the package catalog.

`closed` Logical: If TRUE (default) the schema sets `additionalProperties: false`. Pass FALSE for a family leaf, which its dispatcher composes into one object and closes with `unevaluatedProperties` instead (see S7_dispatcher_JSONSchema).

`instance_schema_url`
Character or NULL: If set, adds a `$schema` const property (instances self-identify, as in the config families).

Value

Named list: the JSON Schema. Serialize with write_JSONSchema.

Author(s)

EDG

Examples

```
## Not run:
schema <- S7_to_JSONSchema(
  LightRFHyperparameters,
  id = "https://schema.rtemis.org/hyperparameters/r/lightrf/v1/schema.json",
  base = Hyperparameters
```

```
)

## End(Not run)
```

`same_data` *same_data() (Deprecated)*

Description

`same_data()` is deprecated. Use `is_same_data()`, which takes the same arguments.

Usage

```
same_data(x, y)
```

Arguments

`x, y`	`DataFingerprint` objects.

Value

As `is_same_data()`.

Author(s)

EDG

Examples

```
suppressWarnings(same_data(data_fingerprint(iris), data_fingerprint(iris)))
```

`se` *Standard error of the fit*

Description

Computes standard errors of the fit for `newdata` from the fitted model, after applying the model's stored preprocessing.

Usage

```
se(x, newdata, ...)
```

Arguments

`x`	`Supervised` object.
`newdata`	tabular data: Features to compute standard errors for.
`...`	Additional arguments passed to methods.

Details

Available for GLM and GAM, which report the standard error of the linear predictor, for BART, which reports the posterior standard deviation of the mean function, and for LINADForest, which reports the infinitesimal jackknife estimate. A multiclass GAM and every other algorithm return NULL.

Value

Numeric vector of standard errors, one per case, or NULL when the algorithm reports none.

Author(s)

EDG

Examples

```
x <- data.frame(age = rnorm(100), bmi = rnorm(100))
x[["y"]] <- x[["age"]] * 2 + rnorm(100, sd = 0.3)
mod <- train(x, hyperparameters = setup_GLM(), verbosity = 0L)
se(mod, features(x)[1:5, ])
```

---

session_timeline *Session Execution Timeline Table*

Description

Flatten the execution graph captured in a `SupervisedSession` (from `train()`) into a timeline (Gantt) table: one row per recorded node, ordered as a depth-first walk of the execution tree, with start/end offsets in milliseconds from session start.

Usage

```
session_timeline(x)
```

Arguments

`x`	`Supervised` or `SupervisedRes` model, or the `SupervisedSession` it carries.

Details

This is the single source of the tree-walk, labeling, and status logic for timeline renderers. `rtemis.server` consumes it for the `job.result session` slice, which the rtemislive web UI draws; pair it with `session_kind_colors()` to color bars by node kind.

Nodes that never closed (status `"running"` or `"aborted"` with no end time) are drawn up to the session finish time when known, else to the latest recorded node start. A session that recorded no nodes, and a model that carries no session, have a timeline of zero rows.

Value

`data.table` with one row per node in depth-first order and columns:

- `label` Character: Unique display label, indented two spaces per tree depth (duplicates disambiguated with `" #<n>"` suffixes).
- `start` Numeric: Node start, milliseconds from session start.
- `end` Numeric: Node end, milliseconds from session start.
- `kind` Character: Node kind (e.g. `"tune"`, `"grid_cell"`, `"train_alg"`).
- `status` Character {"ok", "error", "aborted", "running"}: Node status.
- `failed` Logical: `TRUE` when `status` is `"error"` or `"aborted"`.
- `tip` Character: Preformatted tooltip text (`"name -- duration [status]"`).

Author(s)

EDG

Examples

```
mod <- train(iris, hyperparameters = setup_CART())
# One row per recorded node, depth-first: the run, then the steps under it.
session_timeline(mod)
```

---

setdiffsym *Symmetric Set Difference*

Description

Symmetric Set Difference

Usage

```
setdiffsym(x, y)
```

Arguments

x	vector
y	vector of same type as x

Value

Vector.

Author(s)

EDG

Examples

```
setdiff(1:10, 1:5)
setdiff(1:5, 1:10)
setdiffsym(1:10, 1:5)
setdiffsym(1:5, 1:10)
```

`setup_ARFFIngest` *Set up an* `ARFFIngestConfig`

Description

Creates the settings `ingest()` uses to read an ARFF file. The ARFF header declares the attribute types, and the format has no settings of its own.

Usage

```
setup_ARFFIngest(
  columns = NULL,
  character2factor = TRUE,
  clean_colnames = TRUE,
  remove_duplicates = FALSE
)
```

Arguments

`columns`	Optional Named vector: Declared column types, keyed by column name, in the `profile/r/v1` vocabulary – `c(Age = "integer")`. A column named here is converted to the declared type after reading, replacing the type the reader assigned.
`character2factor`	Logical: If TRUE, read character columns as factors.
`clean_colnames`	Logical: If TRUE, normalize column names.
`remove_duplicates`	Logical: If TRUE, drop rows repeating an earlier row.

Value

`ARFFIngestConfig` object.

Author(s)

EDG

Examples

```
setup_ARFFIngest()
```

setup_ASWCriterion *Setup ASWCriterionConfig*

Description

Setup the average silhouette width criterion for setup_PAMK: every candidate number of clusters is scored by the mean silhouette width over all cases, and the highest wins.

Usage

```
setup_ASWCriterion()
```

Value

`ASWCriterionConfig` object.

Author(s)

EDG

Examples

```
setup_ASWCriterion()
```

setup_Autoencoder *Setup Autoencoder config.*

Description

A fully connected autoencoder built and trained with `torch`: an encoder maps the features to `k` components through the hidden layers, and a decoder mirroring it maps the components back.

Usage

```
setup_Autoencoder(
  k = 2L,
  hidden_units = NULL,
  activation = "relu",
  norm = NULL,
  dropout = 0,
  input_dropout = 0,
  weight_decay = 0,
  input_noise = 0,
  loss = "mse",
  optimizer = "adamw",
  lr = 0.001,
```

```
  beta1 = NULL,
  beta2 = NULL,
  eps = NULL,
  momentum = NULL,
  lr_scheduler = NULL,
  batch_size = NULL,
  max_epochs = 100L,
  patience = 10L,
  max_grad_norm = NULL,
  validation_fraction = 0.1,
  features = NULL
)
```

Arguments

k	Integer [1, Inf): Number of components, the width of the latent layer.
hidden_units	Optional Integer [1, Inf) vector: Units in each hidden layer of the encoder, input side first.
activation	Character {"relu", "gelu", "silu", "elu", "selu", "leaky_relu", "tanh"}: Activation applied after every hidden layer.
norm	Optional Character {"batch_norm", "layer_norm"}: Normalization applied in every hidden layer. NULL applies none.
dropout	Numeric [0, 1): Dropout probability applied after every hidden layer.
input_dropout	Numeric [0, 1): Probability of zeroing each standardized input during training.
weight_decay	Numeric [0, Inf): L2 penalty, decoupled from the gradient under the adamw optimizer.
input_noise	Numeric [0, Inf): Standard deviation of the Gaussian noise added to the standardized inputs during training.
loss	Character {"mse", "l1", "smooth_l1"}: Reconstruction loss.
optimizer	Character {"adamw", "adam", "sgd", "rmsprop"}: Optimization algorithm.
lr	Numeric (0, Inf): Learning rate.
beta1	Optional Numeric [0, 1): Exponential decay rate of the first moment estimate. Applies to the adam and adamw optimizers.
beta2	Optional Numeric [0, 1): Exponential decay rate of the second moment estimate. Applies to the adam and adamw optimizers.
eps	Optional Numeric (0, Inf): Term added to the denominator for numerical stability. Applies to the adam, adamw and rmsprop optimizers.
momentum	Optional Numeric [0, Inf): Momentum factor. Applies to the sgd and rmsprop optimizers.
lr_scheduler	Optional Character {"step", "cosine_annealing", "one_cycle", "reduce_on_plateau"}: Learning-rate schedule. NULL holds the learning rate fixed.
batch_size	Optional Integer [1, Inf): Cases per optimization step.
max_epochs	Integer [1, Inf): Largest number of passes over the training cases.

`patience` Integer [1, Inf): Epochs without improvement of the held-out loss before stopping early.

`max_grad_norm` Optional Numeric (0, Inf): Clip the gradient norm to this value before each step. NULL does not clip.

`validation_fraction` Numeric [0, 1): Fraction of the cases held out for early stopping.

`features` Optional Character vector: Names of at least 2 distinct feature columns to decompose. `NULL` decomposes all numeric features.

Details

Architecture. `hidden_units` gives the encoder's hidden widths from the input side, e.g. `c(64L, 16L)`; the decoder uses them in reverse, and both the latent layer and the output layer are linear. NULL uses one hidden layer of width `round(sqrt(p * k))` for `p` features, the geometric mean of the input and latent widths, held between 32 and 512 and never below `k`. `batch_size = NULL` uses a tenth of the training cases, held between 16 and 256, so small data still gets several optimization steps per epoch. The fit reports and records both values it used.

Inputs are always centered and scaled with the training cases' statistics (a constant feature is only centered). reconstruct undoes this, so reconstructions and the reconstruction metrics are in the units of the data, comparable with any other decomposition's.

Denoising. `input_noise` adds Gaussian noise of that standard deviation to the standardized inputs, and `input_dropout` zeroes each input with that probability, during training only. The model learns to reconstruct the uncorrupted input, which regularizes the components; `apply_decomposition()` and reconstruct never corrupt.

Early stopping. `validation_fraction` of the cases, drawn at random, are held out; training stops after `patience` epochs without improvement of their reconstruction loss and keeps the weights of the best epoch. The components are then computed for every case, held-out ones included. `validation_fraction = 0` trains for `max_epochs`.

Reproducibility, threads and device come from the execution config passed to decomp: its `seed` seeds torch as well as R, its algorithm threads are libtorch's thread count, and its `device` is the compute device. Unset, the device is `cuda` where available and `cpu` otherwise. `"mps"`, the Apple silicon GPU, runs only when named. In rtemis benchmarks on an Apple M5 the CPU fitted faster than `"mps"` in every configuration measured, up to 50,000 cases, 5,000 features, 1,024 hidden units and batches of 2,048, by 1.1 to 3.7 times; `"mps"` also slows down across many fits in one R session. A seed does not reach dropout on `"mps"`, so a seeded fit there reproduces only while `dropout` and `input_dropout` are 0, and warns otherwise.

The optimizer, schedule and early-stopping settings mean what they do in setup_MLP.

Value

AutoencoderConfig object.

Author(s)

EDG

Examples

```
ae_config <- setup_Autoencoder(k = 2L, hidden_units = 8L, input_noise = 0.1)
ae_config
```

`setup_BART` *Setup BART Hyperparameters*

Description

Setup hyperparameters for Bayesian Additive Regression Trees training.

Usage

```
setup_BART(
  num_trees = 200L,
  alpha = 0.95,
  beta = 2,
  min_samples_leaf = 5L,
  max_depth = 10L,
  num_features_subsample = NULL,
  variance_forest_num_trees = 0L,
  num_gfr = 5L,
  num_burnin = 0L,
  num_mcmc = 100L,
  num_chains = 1L,
  keep_every = 1L,
  cutpoint_grid_size = 100L,
  standardize = TRUE,
  link = "probit",
  seed = NULL,
  ifw = FALSE
)
```

Arguments

`num_trees`	(Tunable) Integer [1, Inf): Number of trees in the mean forest.
`alpha`	(Tunable) Numeric (0, 1): Base of the tree split prior `alpha * (1 + depth)^-beta`.
`beta`	(Tunable) Numeric [0, Inf): Depth penalty exponent of the tree split prior `alpha * (1 + depth)^-beta`.
`min_samples_leaf`	(Tunable) Integer [1, Inf): Minimum number of training cases in a leaf.
`max_depth`	(Tunable) Optional Integer [1, Inf): Maximum depth of any tree. NULL imposes no limit.
`num_features_subsample`	(Tunable) Optional Integer [1, Inf): Number of features subsampled when growing each tree. NULL uses every feature.

`variance_forest_num_trees`
: (Tunable) Integer [0, Inf): Number of trees in the conditional variance forest. 0 fits a homoskedastic model, any larger value a heteroskedastic one.

`num_gfr`
: Integer [0, Inf): Number of grow-from-root warm-start iterations.

`num_burnin`
: Integer [0, Inf): Number of burn-in MCMC iterations.

`num_mcmc`
: Integer [1, Inf): Number of retained MCMC iterations per chain.

`num_chains`
: Integer [1, Inf): Number of independent MCMC chains. Cannot exceed `num_gfr` unless `num_gfr` is 0.

`keep_every`
: Integer [1, Inf): Thinning interval: retain one MCMC sample in every `keep_every`.

`cutpoint_grid_size`
: Integer [1, Inf): Maximum number of candidate cutpoints considered by the grow-from-root algorithm.

`standardize`
: Logical: If TRUE, center and scale the outcome before sampling.

`link`
: Character {"probit", "cloglog"}: Link function of the binary outcome model. Classification only.

`seed`
: Optional Integer: Random seed for the sampler. NULL leaves the sampler seeded by the system.

`ifw`
: (Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Both outcome types are fit with stochtree::bart, which samples a sum-of-trees model by MCMC, optionally warm-started by the grow-from-root algorithm. Regression uses a continuous outcome model, binary classification a discrete one with the link named by `link`. Multiclass classification is not supported.

Factors are expanded by the backend, so no encoding is needed beforehand. Case weights scale the residual variance and are honored under the default `link = "probit"`, but `stochtree` rejects them under `"cloglog"`, so that combination makes training abort.

Because the fit is a posterior rather than a point estimate, `se()` returns the standard deviation of the retained draws, and `get_varimp()` reports two measures rather than one:

- `importance`: the variable inclusion proportion, the share of splitting rules that use the feature, averaged across draws.
- `inclusion_sd`: its standard deviation across draws, separating a feature the sampler uses consistently from one whose importance rests on a few draws.

`plot_varimp(mod, measure = "inclusion_sd")` plots the second; the first is the default.

Inclusion proportions only discriminate when trees compete for splits. At the default `num_trees` each tree is a weak learner and uninformative features still get used, so the proportions flatten towards `1/n_features`; refit with a small ensemble (`num_trees = 10L` to `20L`) when the goal is variable selection rather than prediction.

Value

BARTHyperparameters object.

Author(s)

EDG

Examples

```
bart_hyperparams <- setup_BART(num_trees = 50L, num_mcmc = 200L)
bart_hyperparams
```

---

setup_Bootstrap *Setup Bootstrap Resampler*

Description

Setup Bootstrap Resampler

Usage

```
setup_Bootstrap(n_resamples = 10L, id_strat = NULL, seed = NULL)
```

Arguments

n_resamples	Integer [1, Inf): Number of resamples to make.
id_strat	Optional Character: Name of the column holding per-case grouping IDs, e.g. subject IDs when the dataset contains repeated measurements. Cases sharing an ID stay in the same resample, so a case can only be present in the training or the test set, not both. Requires that `resample()` be given the data frame the column lives in. train takes the column as a case identifier, so it groups the resamples by it and excludes it from the features.
seed	Optional Integer [0, Inf): Random seed.

Value

`BootstrapConfig` object.

Author(s)

EDG

Examples

```
bootstrap_resampler <- setup_Bootstrap(n_resamples = 10L, seed = 2026L)
bootstrap_resampler
```

setup_CART *Setup CART Hyperparameters*

Description

Setup hyperparameters for CART training.

Usage

```
setup_CART(
  cp = 0.01,
  maxdepth = 20L,
  minsplit = 2L,
  minbucket = 1L,
  prune_cp = NULL,
  method = "auto",
  model = TRUE,
  maxcompete = 4L,
  maxsurrogate = 5L,
  usesurrogate = 2L,
  surrogatestyle = 0L,
  xval = 0L,
  cost = NULL,
  ifw = FALSE
)
```

Arguments

cp	(Tunable) Numeric [0, Inf): Complexity parameter.
maxdepth	(Tunable) Integer [1, 30]: Maximum depth of tree.
minsplit	(Tunable) Integer [1, Inf): Minimum number of observations in a node to split.
minbucket	(Tunable) Integer [1, Inf): Minimum number of observations in a terminal node.
prune_cp	(Tunable) Optional Numeric [0, Inf): Complexity for cost-complexity pruning after tree is built. NULL prunes nothing after the tree is built.
method	Character {"auto", "anova", "class", "poisson", "exp"}: Splitting method.
model	Logical: If TRUE, keep a copy of the model frame.
maxcompete	Integer [0, Inf): Maximum number of competitive splits.
maxsurrogate	Integer [0, Inf): Maximum number of surrogate splits.
usesurrogate	Integer [0, 2]: Number of surrogate splits to use.
surrogatestyle	Integer [0, 1]: Type of surrogate splits.
xval	Integer [0, Inf): Number of cross-validation folds.
cost	Optional Numeric (0, Inf) vector: One for each feature. NULL uses 1 for every feature.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Get more information from rpart::rpart and rpart::rpart.control.

Value

CARTHyperparameters object.

Author(s)

EDG

Examples

```
cart_hyperparams <- setup_CART(cp = 0.01, maxdepth = 10L, ifw = TRUE)
cart_hyperparams
```

`setup_CHCriterion` *Setup CHCriterionConfig*

Description

Setup the Calinski-Harabasz criterion for setup_PAMK: every candidate number of clusters is scored by the ratio of between-cluster to within-cluster dispersion, and the highest wins.

Usage

```
setup_CHCriterion()
```

Value

`CHCriterionConfig` object.

Author(s)

EDG

Examples

```
setup_CHCriterion()
```

`setup_ClusterConfig` *Setup ClusterConfig*

Description

Setup a `ClusterConfig` object. `ClusterConfig` is a portable, data-agnostic recipe for a clustering pipeline: `dat_path` is optional, so the same config can be validated, shared, or described without data and have a path bound later (e.g. by the `rtemis` CLI) before cluster.

Usage

```
setup_ClusterConfig(
  dat_path = NULL,
  clustering_config = NULL,
  execution_config = setup_FutureExecution(),
  outdir = "results/",
  verbosity = 1L
)
```

Arguments

`dat_path`	Character or NULL: Path to input data file. NULL leaves the recipe unbound; set it (or supply data) before cluster.
`clustering_config`	`ClusteringConfig` object or NULL: Configuration for the clustering itself, naming the algorithm, its settings and the feature columns to cluster on. Setup with a clustering `setup_*` function, e.g. setup_KMeans. If NULL, cluster runs its default algorithm with default settings.
`execution_config`	`ExecutionConfig` object: Execution settings, e.g. setup_FutureExecution. Its seed seeds the fit.
`outdir`	Character: Output directory for results.
`verbosity`	Integer [0, Inf): Verbosity level.

Value

`ClusterConfig` object.

Author(s)

EDG

Examples

```
cc <- setup_ClusterConfig(
  dat_path = "data.csv",
  clustering_config = setup_KMeans(k = 3L),
  outdir = "results/"
)
```

`setup_CMeans` *Setup CMeansConfig*

Description

Setup CMeansConfig

Usage

```
setup_CMeans(
  k = 2L,
  max_iter = 100L,
  dist = "euclidean",
  method = "cmeans",
  m = 2,
  rate_par = NULL,
  weights = 1,
  control = list(),
  features = NULL
)
```

Arguments

`k`	Integer [1, Inf): Number of clusters.
`max_iter`	Integer [1, Inf): Maximum number of iterations.
`dist`	Character {"euclidean", "manhattan"}: Distance measure to use.
`method`	Character {"cmeans", "ufcl"}: "cmeans" - fuzzy c-means clustering; "ufcl": online update.
`m`	Numeric (1, Inf): Degree of fuzzification.
`rate_par`	Optional Numeric [0, 1]: Learning rate for the online variant.
`weights`	Numeric vector: Case weights. Either a scalar, applied to every case, or a vector with one value per case.
`control`	List: Control config for clustering algorithm.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Value

CMeansConfig object.

Author(s)

EDG

Examples

```
cmeans_config <- setup_CMeans(k = 4L, dist = "euclidean")
cmeans_config
```

---

setup_ConditionalSuperLearner

Setup ConditionalSuperLearner Hyperparameters

Description

Setup hyperparameters for the Conditional SuperLearner, which selects the best model from a library *conditional on the covariates* rather than combining them.

Usage

```
setup_ConditionalSuperLearner(
  base_learners = list(setup_GLM(), setup_GLMNET(), setup_Ranger()),
  meta_learner = setup_Ranger(),
  inner_resampling_config = setup_KFold(n_resamples = 10L),
  n_iterations = 4L,
  loss = NULL,
  init = "full",
  min_region_size = 10L,
  expand_search_spaces = TRUE,
  ifw = FALSE
)
```

Arguments

base_learners	List of Hyperparameters objects: The experts. Names label the regions and the reported region sizes; unnamed entries are named after their algorithm.
meta_learner	Hyperparameters object: The oracle, a classifier fitted on the extended dataset.
inner_resampling_config	ResamplerConfig object: Cross-validation scheme, which must hold out every case exactly once.
n_iterations	(Tunable) Integer [1, Inf): Alternations between fitting the oracle and refitting the experts.
loss	Optional Character {"squared_error", "log_loss"}: Per-case loss the oracle minimizes.
init	Character {"full", "random"}: Region each expert starts from.

`min_region_size`
: Integer [1, Inf): Fewest cases an expert's region may hold before the expert keeps its previous fit instead of being refitted.

`expand_search_spaces`
: Logical: If TRUE, expand a base learner's search space into one expert per combination.

`ifw`
: (Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

The model is `sum_k 1\{o(x) = k\} F_k(x)`: an "oracle" `o` assigns each case to one of the K "experts" `F_k`, and that expert predicts it. Oracle and experts are fitted by alternating, in the manner of k-means: with the experts fixed, the oracle minimizes the total loss, which reduces to a weighted K-class classification over an extended dataset holding each case once per expert; with the oracle fixed, each expert is refitted on the cases assigned to it. The losses are cross-validated throughout, so an expert is never scored on a case it was trained on.

The oracle is a classifier over the original covariates whatever the outcome is, so with more than two experts it must handle more than two classes. Its variable importance – which covariates decide *which model applies* – is what get_varimp returns for a fitted model, and a simple oracle such as setup_CART makes the partition itself readable.

Supports regression and binary classification.

Reference: Valdes, Interian, Gennatas & van der Laan, "Conditional Super Learner", IEEE Transactions on Pattern Analysis and Machine Intelligence (2022). doi:10.1109/TPAMI.2021.3131976

Value

ConditionalSuperLearnerHyperparameters object.

Author(s)

EDG

Examples

```
csl_hyperparams <- setup_ConditionalSuperLearner(
  base_learners = list(setup_GLM(), setup_CART())
)
csl_hyperparams
```

---

setup_CQR *Setup Conformalized Quantile Regression*

Description

Configure conformalized quantile regression for conformal.

Usage

```
setup_CQR(alpha = 0.1)
```

Arguments

alpha	Numeric (0, 1): Miscoverage rate. Also sets the quantile levels queried, alpha/2 and 1 - alpha/2.

Details

Split conformal adds and subtracts one number, so every case gets the same width whether the model is confident or not. CQR (Romano, Patterson and Candes, 2019) starts from the model's own alpha/2 and 1 - alpha/2 quantiles and conformalizes the pair with the score max(qlo - y, y - qhi), which widens or narrows both ends by one calibrated amount. Adaptive width, same finite-sample guarantee.

It needs a backend that answers a quantile query from the model already fitted. That is Ranger trained with setup_Ranger(quantreg = TRUE): a quantile regression forest keeps the training outcomes at its terminal nodes, so one fit answers every level. A gradient booster on the quantile objective targets the single level it was trained for, so a pair of bounds would be a pair of models – a training configuration, not something conformal() can do to a model it is handed.

It always needs calibration **data.** A Supervised stores its test predictions but not its test features, and CQR must query the model for quantiles at the calibration cases, so the stored-split default does not apply here.

Value

CQRConfig object.

Author(s)

EDG

Examples

```
setup_CQR()
setup_CQR(alpha = 0.05)
```

---

setup_CUDA *Setup an NVIDIA GPU as the compute device*

Description

Request an NVIDIA GPU through CUDA as the compute device of an execution config, naming which GPUs to use, e.g. setup_FutureExecution(device = setup_CUDA(ids = 1L)). Without ids, device = "cuda" is the same request. Algorithms that run on CUDA are the torch-backed ones (MLP, TabNet) and the LightGBM family built with CUDA support; every other algorithm runs on the CPU.

Usage

```
setup_CUDA(ids = NULL)
```

Arguments

ids	Optional Integer [0, Inf) vector: Zero-based indices of the GPUs to use. NULL uses the first visible GPU. A run uses the first index listed.

Value

CUDADeviceConfig object.

Author(s)

EDG

Examples

```
setup_CUDA()
setup_CUDA(ids = 1L)
```

---

setup_Custom *Setup Custom Resampler*

Description

Setup Custom Resampler

Usage

```
setup_Custom(resamples = NULL)
```

Arguments

resamples	Optional List: Resamples to use, one integer vector of training-case indices per resample.

Value

CustomConfig object.

Author(s)

EDG

Examples

```
custom_resampler <- setup_Custom(resamples = list(1:3, 2:4))
custom_resampler
```

`setup_CVPlus` *Setup CV+ Conformal Prediction*

Description

Configure CV+, jackknife+ and cross-conformal prediction for conformal over a model trained with outer resampling.

Usage

```
setup_CVPlus(alpha = 0.1, score = NULL, seed = NULL)
```

Arguments

`alpha`	Numeric (0, 1): Miscoverage rate. The region covers with probability at least `1 - 2 * alpha` in the worst case.
`score`	Optional Character {"absolute", "LAC", "APS"}: Nonconformity score. NULL resolves to "absolute" for a regression and "APS" for a classification. See setup_SplitConformal for the trade between the two set-valued scores, and for why APS draws.
`seed`	Optional Integer [0, Inf): Random seed for the APS draw. NULL draws one and records it.

Details

CV+ (Barber, Candes, Ramdas and Tibshirani, 2021) spends no data on calibration: every case is out-of-fold exactly once, so every case both trains and calibrates. A `SupervisedRes` stores the per-fold models and their out-of-fold predictions, so the computation is predicting `newdata` under each fold's model and taking two order statistics.

It delivers `1 - 2 * alpha` **in the worst case**, not `1 - alpha`. The bound is conservative and observed coverage usually sits near `1 - alpha`; the region states the guarantee it carries.

The folds must partition the cases, each case out-of-fold exactly once. `setup_KFold()` and `setup_LOOCV()` do; the subsampling and bootstrap types do not, and `conformal()` refuses them by checking the indices.

Which relative runs follows from the object: leave-one-out folds make it jackknife+, and a classification outcome makes it cross-conformal (Vovk, 2015), CV+ being defined for intervals. The result records which.

Value

`CVPlusConfig` object.

Author(s)

EDG

Examples

```
setup_CVPlus()
setup_CVPlus(alpha = 0.2)
```

`setup_DBSCAN` *Setup DBSCANConfig*

Description

Setup DBSCANConfig

Usage

```
setup_DBSCAN(
  eps = 0.5,
  min_points = 5L,
  weights = NULL,
  border_points = TRUE,
  search = "kdtree",
  bucket_size = 100L,
  split_rule = "SUGGEST",
  approx = 0,
  features = NULL
)
```

Arguments

`eps`	Numeric (0, Inf): Radius of neighborhood.
`min_points`	Integer [1, Inf): Minimum number of points in a neighborhood to form a cluster.
`weights`	Optional Numeric vector: Weights for data points.
`border_points`	Logical: If TRUE, assign border points to clusters.
`search`	Character {"kdtree", "linear", "dist"}: Nearest neighbor search strategy.
`bucket_size`	Integer [1, Inf): Size of buckets for k-d tree search.
`split_rule`	Character {"SUGGEST", "STD", "MIDPT", "FAIR", "SL_MIDPT", "SL_FAIR"}: Rule for splitting the k-d tree.
`approx`	Numeric [0, Inf): Approximation factor for the nearest neighbor search (`dbscan::frNN(approx =)`); 0 searches exactly. A positive value guarantees only the neighbors within `eps / (1 + approx)` and may omit others within `eps`, which can produce spurious clusters and noise points in exchange for speed.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Value

DBSCANConfig object.

Author(s)

EDG

Examples

```
dbscan_config <- setup_DBSCAN(eps = 0.5, min_points = 5L)
dbscan_config
```

---

setup_DecomposeConfig *Setup DecomposeConfig*

Description

Setup a DecomposeConfig object. DecomposeConfig is a portable, data-agnostic recipe for a decomposition pipeline: dat_path is optional, so the same config can be validated, shared, or described without data and have a path bound later (e.g. by the rtemis CLI) before decomp.

Usage

```
setup_DecomposeConfig(
  dat_path = NULL,
  decomposition_config = NULL,
  execution_config = setup_FutureExecution(),
  outdir = "results/",
  verbosity = 1L
)
```

Arguments

dat_path	Character or NULL: Path to input data file. NULL leaves the recipe unbound; set it (or supply data) before decomp.
decomposition_config	DecompositionConfig object or NULL: Configuration for the decomposition itself. Setup with a decomposition setup_* function, e.g. setup_PCA. If NULL, decomp runs its default algorithm with default settings.
execution_config	ExecutionConfig object: Execution settings, e.g. setup_FutureExecution. Its workers are threads for an algorithm that uses them, and its seed seeds the fit.
outdir	Character: Output directory for results.
verbosity	Integer [0, Inf): Verbosity level.

Value

DecomposeConfig object.

Author(s)

EDG

Examples

```
dc <- setup_DecomposeConfig(
  dat_path = "data.csv",
  decomposition_config = setup_PCA(k = 3L),
  outdir = "results/"
)
```

---

setup_DelimitedIngest *Set up a* DelimitedIngestConfig

Description

Creates the settings `ingest()` uses to read a delimited file.

Each format has its own constructor, which takes the settings of that format only; an argument of another format is an unused-argument error.

Usage

```
setup_DelimitedIngest(
  columns = NULL,
  character2factor = TRUE,
  clean_colnames = TRUE,
  remove_duplicates = FALSE,
  reader = "data.table",
  sep = NULL,
  quote = "\"",
  na_strings = ""
)
```

Arguments

`columns`	Optional Named vector: Declared column types, keyed by column name, in the `profile/r/v1` vocabulary – `c(Age = "integer")`. A column named here is converted to the declared type after reading, replacing the type the reader assigned.
`character2factor`	Logical: If TRUE, read character columns as factors.
`clean_colnames`	Logical: If TRUE, normalize column names.
`remove_duplicates`	Logical: If TRUE, drop rows repeating an earlier row.
`reader`	Character {"data.table", "vroom", "duckdb", "arrow"}: Library used to read the file.

sep	Optional Character: Field separator. If NULL, the reader detects it.
quote	Character: Quote character.
na_strings	Character vector: Strings to read as missing.

Value

DelimitedIngestConfig object.

Author(s)

EDG

Examples

```
setup_DelimitedIngest(sep = ";")
```

---

setup_DTAIngest *Set up a* DTAIngestConfig

Description

Creates the settings ingest() uses to read a Stata file. Stata files store their column types, and the format has no settings of its own.

Usage

```
setup_DTAIngest(
  columns = NULL,
  character2factor = TRUE,
  clean_colnames = TRUE,
  remove_duplicates = FALSE
)
```

Arguments

columns	Optional Named vector: Declared column types, keyed by column name, in the profile/r/v1 vocabulary – c(Age = "integer"). A column named here is converted to the declared type after reading, replacing the type the reader assigned.
character2factor	Logical: If TRUE, read character columns as factors.
clean_colnames	Logical: If TRUE, normalize column names.
remove_duplicates	Logical: If TRUE, drop rows repeating an earlier row.

Value

`DTAIngestConfig` object.

Author(s)

EDG

Examples

```
setup_DTAIngest()
```

`setup_FutureExecution` *Set up a* `FutureExecutionConfig`

Description

Parallel execution through **future**. A plan is required, and is filled in from `getOption("future.plan", "mirai_multisession")` when not given.

Usage

```
setup_FutureExecution(
  n_workers = NULL,
  n_workers_outer = NULL,
  n_workers_tuning = NULL,
  n_workers_algorithm = NULL,
  device = NULL,
  future_plan = getOption("future.plan", "mirai_multisession"),
  on_error = c("continue", "stop", "stop_outer"),
  seed = NULL,
  shared_memory = c("auto", "none", "always")
)
```

Arguments

`n_workers`	Integer [1, Inf): Number of parallel workers. Left `NULL`, it follows the named dispatch levels, or `default_n_workers()` if none is named.
`n_workers_outer`	Optional Integer [1, Inf): Workers for outer resampling, overriding the automatic assignment.
`n_workers_tuning`	Optional Integer [1, Inf): Workers for tuning, overriding the automatic assignment.
`n_workers_algorithm`	Optional Integer [1, Inf): Threads for a self-parallelizing algorithm, overriding the automatic assignment.

`device` Optional `DeviceConfig`: Compute device, as a type name – `"cpu"`, `"cuda"`, `"mps"` (the Apple silicon GPU) or `"opencl"` (a LightGBM GPU build) – or setup_CUDA to name which GPUs. `NULL` selects one per algorithm: `"cuda"` where the machine has it and the algorithm can use it, else the CPU. An algorithm that cannot use the requested device runs on the CPU, and the run's resources line says so; a device the machine lacks is an error when an algorithm tries to use it. `"mps"` is never selected automatically: in rtemis benchmarks on an Apple M5 the CPU was faster for every network narrower than about 1024 units or trained in batches smaller than about 2048 cases, often by 2 to 5 times, and `"mps"` was faster by up to about 25% only with both and thousands of features. It also slows down across many fits in one R session, and a seed does not reach dropout on it.

`future_plan` Character: Future plan to use. Defaults to `getOption("future.plan", "mirai_multisession")`.

`on_error` Character {"continue", "stop", "stop_outer"}: Failure policy. `"continue"` makes grid cells and unscorable hyperparameter combinations non-fatal (recorded, warned, and excluded), failing only when nothing is scorable or the final model fails; `"stop"` aborts on any error; `"stop_outer"` tolerates grid-cell failures but aborts on an outer-fold failure.

`seed` Optional Integer [0, Inf): Master seed for the run's computation RNG, from which one independent substream per parallel task is derived. Left `NULL`, a seed is drawn from the current RNG stream and recorded on the returned object, so every run is reproducible and the seed it used is auditable. Distinct from a `ResamplerConfig` seed, which governs how the data is split.

`shared_memory` Character {"none", "auto", "always"}: Whether to hand workers the training data through OS shared memory instead of serializing a copy to each. `"auto"` is the default: it shares whenever it can – workers parallel and on this machine, **mori** installed – and quietly does not when it cannot, falling back to the ordinary transport. `"none"` disables it. `"always"` is a demand: it shares even when the run is sequential, so a run can be compared against its own shared counterpart, and raises an error when the request cannot be met, for a caller relying on sharing to stay inside a memory budget.

Details

Worker levels

There are three levels that can absorb workers: a self-parallelizing algorithm, tuning, and outer resampling. Left alone, `n_workers` is assigned to exactly one of them, in that order of priority, so no two levels compete for the same cores.

Naming any of `n_workers_outer`, `n_workers_tuning` or `n_workers_algorithm` takes over that assignment: the automatic one is switched off entirely and any level not named gets 1, so a run never mixes a hand-picked level with an inferred one.

Outer resampling and tuning dispatch to worker *processes*, and an outer fold runs inside one of those processes, so only one of the two can be parallel – setting both above 1 is an error. `n_workers_algorithm` is threads within a worker, so it combines with either.

Reproducibility

Substreams are assigned by task index, so a run gives the same answer under every backend and worker count, and the parallel result matches the sequential one exactly.

Shared memory needs **mori**, whose minimum R version (4.3) is above **rtemis**'s own (4.1), so `"auto"` is the portable setting for a script that may run anywhere. It is local RAM: workers on another machine cannot map it.

Value

`FutureExecutionConfig` object.

Author(s)

EDG

Examples

```
setup_FutureExecution(n_workers = 4L, future_plan = "multisession")
```

---

setup_GAM *Setup GAM Hyperparameters*

Description

Setup hyperparameters for GAM training.

Usage

```
setup_GAM(k = 5L, ifw = FALSE)
```

Arguments

`k`	(Tunable) Integer [1, Inf): Number of knots.
`ifw`	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Get more information from mgcv::gam.

Value

GAMHyperparameters object.

Author(s)

EDG

Examples

```
gam_hyperparams <- setup_GAM(k = 5L, ifw = FALSE)
gam_hyperparams
```

setup_GLM *Setup GLM Hyperparameters*

Description

Setup hyperparameters for GLM training.

Usage

```
setup_GLM(ifw = FALSE)
```

Arguments

ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Value

GLMHyperparameters object.

Author(s)

EDG

Examples

```
glm_hyperparams <- setup_GLM(ifw = TRUE)
glm_hyperparams
```

setup_GLMNET *Setup GLMNET Hyperparameters*

Description

Setup hyperparameters for GLMNET training.

Usage

```
setup_GLMNET(
  alpha = 1,
  family = NULL,
  which_lambda_cv = "lambda.1se",
  nlambda = 100L,
  lambda = NULL,
  penalty_factor = NULL,
  standardize = TRUE,
  intercept = TRUE,
  ifw = TRUE
)
```

Arguments

alpha	(Tunable) Numeric [0, 1]: Elastic net mixing parameter.
family	Optional Character {"gaussian", "binomial", "poisson", "multinomial", "cox", "mgaussian"}: Family. NULL = set from outcome type.
which_lambda_cv	Character {"lambda.1se", "lambda.min"}: Which lambda to use for prediction.
nlambda	Integer [1, Inf): Number of lambda values.
lambda	Optional Numeric [0, Inf) vector: Lambda values. NULL = determined by cv.glmnet during tuning.
penalty_factor	Optional Numeric [0, Inf) vector: Penalty factor for each design-matrix column (factors expanded to indicator columns). NULL uses 1 for every column.
standardize	Logical: If TRUE, standardize features.
intercept	Logical: If TRUE, include intercept.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Get more information from glmnet::glmnet.

Value

GLMNETHyperparameters object.

Author(s)

EDG

Examples

```
glm_hyperparams <- setup_GLMNET(alpha = 1, ifw = TRUE)
glm_hyperparams
```

---

setup_GLMTree *Setup GLMTree Hyperparameters*

Description

Setup hyperparameters for GLMTree: model-based recursive partitioning, a tree carrying a generalized linear model in each leaf.

Usage

```
setup_GLMTree(
  alpha = 0.05,
  minsize = NULL,
  maxdepth = NULL,
  mtry = NULL,
  trim = 0.1,
  prune = NULL,
  numsplit = "left",
  catsplit = "binary",
  ordinal = "chisq",
  ifw = FALSE,
  regressors = NULL,
  partitioning_variables = NULL,
  bonferroni = TRUE,
  breakties = FALSE,
  restart = TRUE,
  dfsplit = TRUE,
  vcov = "opg",
  nrep = 10000L
)
```

Arguments

alpha	(Tunable) Numeric (0, 1]: Significance level a parameter-instability test must reach before a node is split.
minsize	(Tunable) Optional Integer [1, Inf): Fewest cases a node may hold. NULL uses ten times the number of parameters in a leaf's model.
maxdepth	(Tunable) Optional Integer [1, Inf): Deepest the tree may grow. NULL is unlimited.
mtry	(Tunable) Optional Integer [1, Inf): Partitioning variables sampled at each node. NULL tests every one.
trim	(Tunable) Numeric [0, Inf): Trimming for the split-point search; a fraction of cases below 1, a count at or above it.
prune	(Tunable) Optional Character {"AIC", "BIC"}: Post-prune the grown tree by an information criterion. NULL keeps every split.
numsplit	(Tunable) Character {"left", "center"}: Where a numeric split is placed.
catsplit	(Tunable) Character {"binary", "multiway"}: How a categorical variable is split.
ordinal	(Tunable) Character {"chisq", "max", "L2"}: Test statistic for an ordered partitioning variable.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.
regressors	Optional Character vector: Features entering the linear model in each leaf. NULL uses every feature.
partitioning_variables	Optional Character vector: Features the tree may split on. NULL uses every feature.

bonferroni	Logical: If TRUE, Bonferroni-adjust the test p-values for the number of partitioning variables.
breakties	Logical: If TRUE, break ties in numeric partitioning variables by adding noise.
restart	Logical: If TRUE, restart each leaf's model from scratch rather than from its parent's fit.
dfsplit	Logical: If TRUE, count each split against the degrees of freedom when pruning.
vcov	Character {"opg", "info", "sandwich"}: Covariance estimator the instability test is built on.
nrep	Integer [1, Inf): Simulation replicates for the p-value of an ordinal split statistic.

Details

Where a decision tree predicts a constant in each leaf, a GLMTree fits a regression there, so the model reads as "in this subgroup, these are the coefficients". `partykit::lmtree()` fits a regression, `partykit::glmtree()` a binary logistic one; rtemis selects between them from the outcome.

Value

GLMTreeHyperparameters object.

How it splits

A node is split when a parameter-instability test, a structural change test at significance level `alpha`, rejects constancy of its model's coefficients across a candidate partitioning variable. Consequences:

It has a hypothesis-testing foundation Splits carry p-values, and `alpha` is a significance level with a direct interpretation.

It detects a slope change with no shift in the mean A tree that splits on the response mean does not.

It needs statistical power With few cases, many partitioning variables, or both, the tests do not reach `alpha` and the tree does not split at all – returning a single global model.

When it will not split, check `minsize` before `alpha`

The default `minsize` is **ten times the number of parameters in a leaf's model**, and that scales with the number of regressors. A split has to leave `minsize` cases on each side, so with `p` regressors no split is possible below `n = 20 * (p + 1)`: 420 cases at 20 regressors, for any `alpha`. On a wide design, a single-leaf tree results from this bound rather than from `alpha`. Set `minsize` directly, or narrow `regressors`.

`regressors` and `partitioning_variables` are set independently: the model in each leaf may use features other than those the tree splits on. Both default to every feature.

Interpretability

Every leaf carries a full model over `regressors`, with no selection or shrinkage: one coefficient per regressor. Restrict `regressors` to reduce the number of coefficients.

What it does not provide

`partykit` provides variable importance for `cforest` and for constant-fit trees, not for a model-based tree. `get_varimp()` returns NULL; `get_varimp(explain(mod, newdata))` returns a per-feature measure.

Author(s)

EDG

References

Zeileis A, Hothorn T, Hornik K (2008). Model-Based Recursive Partitioning. *Journal of Computational and Graphical Statistics*, 17(2), 492-514. doi:10.1198/106186008X319331

Examples

```
glmtree_hyperparams <- setup_GLMTree(alpha = 0.01)
glmtree_hyperparams
```

`setup_GMM` *Setup GMMConfig*

Description

Setup a `GMMConfig` object for Gaussian Mixture Model clustering, via the 'mclust' package.

Usage

```
setup_GMM(k = NULL, model_names = NULL, features = NULL)
```

Arguments

`k`	Optional Integer [1, Inf): Number of mixture components. Unset selects it by BIC.
`model_names`	Optional Character {"EII", "VII", "EEI", "VEI", "EVI", "VVI", "EEE", "VEE", "EVE", "VVE", "EEV", "VEV", "EVV", "VVV"} vector: Covariance parameterizations to consider. Unset considers all of them.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Details

A GMM is the model-based counterpart of the prototype methods: each cluster is a Gaussian component with its own mean and covariance, so clusters may be elongated, differently oriented and differently sized, and every case gets a posterior probability for each component as well as a label. The result is therefore a soft clustering.

Both the number of components and the covariance parameterization are selected by BIC, which makes `k` optional: leave it unset to have the number of components chosen, or set it to fix the number and let BIC choose only the parameterization.

Argument names are rtemis' own: `k` is `mclust::Mclust()`'s `G` and `model_names` is its `modelNames`.

Value

`GMMConfig` object.

Author(s)

EDG

References

Scrucca L, Fop M, Murphy TB, Raftery AE (2016). mclust 5: Clustering, Classification and Density Estimation Using Gaussian Finite Mixture Models. *The R Journal*, 8(1), 289-317. doi:10.32614/RJ2016021

Examples

```
gmm_config <- setup_GMM(k = 3L)
gmm_config
```

`setup_GridSearch` *Setup Grid Search Config*

Description

Create a `GridSearchConfig` object that can be passed to train.

Usage

```
setup_GridSearch(
  resampler_config = setup_KFold(n_resamples = 5L),
  search_type = "exhaustive",
  randomize_p = NULL,
  metrics_aggregate_fn = "mean",
  metric = NULL,
  maximize = NULL
)
```

Arguments

resampler_config	Optional ResamplerConfig: Resampling configuration from a setup_* resampler function, e.g. setup_KFold.
search_type	Character {"exhaustive", "randomized"}: Type of grid search to use. Exhaustive search will try all combinations of config. Randomized will try a random sample of size randomize_p * N of total combinations
randomize_p	Optional Numeric (0, 1): Randomly test this proportion of combinations. Required when search_type is "randomized"; must be left NULL when it is "exhaustive".
metrics_aggregate_fn	Character: Name of function to use to aggregate error metrics.
metric	Optional Character: Metric to minimize or maximize. NULL sets it from the outcome type.
maximize	Optional Logical: If TRUE, maximize metric, otherwise minimize it. NULL maximizes accuracy, balanced_accuracy, sensitivity, specificity, ppv, npv, f1, auc, rsq and Concordance, and minimizes every other metric, including mae, mse, rmse and brier_score.

Value

A GridSearchConfig object.

Author(s)

EDG

Examples

```
gridsearch_config <- setup_GridSearch(
  resampler_config = setup_KFold(n_resamples = 5L),
  search_type = "exhaustive"
)
gridsearch_config
```

---

setup_GroupPartition *Set up a* GroupPartitionConfig

Description

A split that keeps every case sharing a group ID on the same side.

Usage

```
setup_GroupPartition(column = NULL, train_p = 0.75, seed = NULL)
```

Arguments

column	Optional Character: Name of the column holding per-case group IDs.
train_p	Numeric (0, 1): Approximate training set fraction, honored at the group level.
seed	Optional Integer [0, Inf): Random seed.

Value

GroupPartitionConfig object.

Author(s)

EDG

Examples

```
setup_GroupPartition(column = "subject_id", train_p = 0.8, seed = 2026L)
```

---

setup_HAL *Setup HAL Hyperparameters*

Description

Setup hyperparameters for Highly Adaptive Lasso training.

Usage

```
setup_HAL(
  max_degree = 2L,
  smoothness_orders = 1L,
  reduce_basis = NULL,
  num_knots = NULL,
  use_min = TRUE,
  nfolds = 10L,
  max_basis = 5000000L,
  seed = NULL,
  ifw = FALSE
)
```

Arguments

max_degree	(Tunable) Integer [1, Inf): Highest order of interaction among features that a basis function may represent.
smoothness_orders	(Tunable) Integer [0, 9]: Smoothness of the basis functions: 0 fits zero-order indicators, 1 piecewise linear splines, higher values higher-order splines.

reduce_basis	(Tunable) Optional Numeric (0, 1]: Minimum proportion of cases a basis function must be non-zero in to be kept. Applies only when smoothness_orders is 0; a search that also covers higher orders drops it from those grid cells.
num_knots	Optional Integer [1, Inf) vector: Number of knots per interaction degree, one value per degree, non-increasing. NULL generates them from max_degree and smoothness_orders.
use_min	Logical: If TRUE, select lambda.min from the internal cross-validation; if FALSE, the more heavily penalized lambda.1se.
nfolds	Integer [3, Inf): Number of folds of the internal cross-validation that selects lambda.
max_basis	Integer [1, Inf): Largest projected basis-function count that will be fit.
seed	Optional Integer: Random seed for the internal cross-validation's fold assignment. NULL leaves it drawn from the ambient RNG.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Both outcome types are fit with hal9001::fit_hal, which spans the outcome with a basis of indicator or spline terms over every interaction of the features up to max_degree, then fits a lasso over that basis. Regression uses the gaussian family, binary classification the binomial one; hal9001 has no multinomial family, so multiclass classification is not supported.

lambda is selected by cross-validation inside the fit and is not a search dimension. seed fixes that cross-validation's fold assignment; nfolds and use_min control it.

hal9001 takes a numeric matrix, so factors are one-hot encoded first and the encoder is re-applied at predict time.

Scaling: the basis has one function per knot per feature subset, so its size grows as C(n_features, max_degree) and, through the knots, with the number of cases, making the lasso that follows quadratic in the number of cases. Training projects the basis size before fitting, reports it at verbosity >= 1, warns above one million, and aborts above max_basis. max_degree has the largest effect on that count, followed by num_knots.

get_varimp() reports two measures, both aggregated over the basis functions that involve each feature:

- importance: the sum of the absolute values of their non-zero coefficients.
- max_coefficient: the largest single such absolute coefficient, separating a feature carried by one strong term from one carried by many weak ones.

plot_varimp(mod, measure = "max_coefficient") plots the second; the first is the default. Both read coefficients on the scale of the basis functions. At smoothness_orders = 0 the basis is made of indicators, so the coefficients are unit-free and directly comparable across features; at higher orders each basis function carries the units of its feature, so scale the features first if they are not already comparable.

Value

HALHyperparameters object.

Author(s)

EDG

Examples

```
hal_hyperparams <- setup_HAL(max_degree = 1L)
hal_hyperparams
```

`setup_HardCL` *Setup HardCLConfig*

Description

Setup HardCLConfig

Usage

```
setup_HardCL(k = 3L, dist = "euclidean", features = NULL)
```

Arguments

`k`	Integer [1, Inf): Number of clusters.
`dist`	Character {"euclidean", "manhattan"}: Distance measure to use.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Value

HardCLConfig object.

Author(s)

EDG

Examples

```
hardcl_config <- setup_HardCL(k = 4L, dist = "euclidean")
hardcl_config
```

`setup_HOPACH` *Setup HOPACHConfig*

Description

Setup a `HOPACHConfig` object for Hierarchical Ordered Partitioning and Collapsing Hybrid (HOPACH) clustering, via the Bioconductor package 'hopach'.

Usage

```
setup_HOPACH(
  dist = "cosangle",
  level_selection = "best",
  max_levels = 15L,
  max_children = 9L,
  max_children_mss = 9L,
  collapse = "seq",
  new_medoid = "medsil",
  mss = "med",
  min_improvement = 0,
  initial_order = "co",
  element_order = "own",
  features = NULL
)
```

Arguments

`dist`	Character {"cosangle", "abscosangle", "euclid", "cor", "abscor"}: Distance measure.
`level_selection`	Character {"best", "greedy"}: Which level of the tree becomes the returned partition. "best": the level minimizing MSS; "greedy": the first level below which MSS increases.
`max_levels`	Integer [1, 15]: Maximum number of levels in the tree.
`max_children`	Integer [2, 9]: Maximum number of children at each node.
`max_children_mss`	Integer [2, 9]: Maximum number of children at each node when computing MSS.
`collapse`	Character {"seq", "all"}: How collapsing is performed at each level. "seq": collapse pairs sequentially from the closest while MSS decreases; "all": collapse any pair that decreases MSS.
`new_medoid`	Character {"medsil", "center", "nn", "uwnn"}: How the medoid of a collapsed pair is chosen.
`mss`	Character {"med", "mean"}: Split silhouette criterion; median or mean.

`min_improvement`
: Numeric [0, 1]: Relative improvement in MSS required to accept a collapse step.

`initial_order`
: Character {"co", "clust"}: How clusters in the first level are ordered.

`element_order`
: Character {"own", "neighbor"}: How elements are ordered within a cluster.

`features`
: Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Details

HOPACH identifies the number of clusters itself, by building an ordered hierarchical tree and selecting the level with maximally homogeneous clusters under the Median (or Mean) Split Silhouette criterion. There is therefore no `k` to set: `max_levels` and `max_children` bound the tree, and the number of clusters found is reported by the resulting `Clustering`.

Argument names are rtemis' own. They map to `hopach::hopach()` as: `dist` -> `d`, `level_selection` -> `clusters`, `max_levels` -> `K`, `max_children` -> `kmax`, `max_children_mss` -> `khigh`, `collapse` -> `coll`, `new_medoid` -> `newmed`, `mss` -> `mss`, `min_improvement` -> `impr`, `initial_order` -> `initord`, `element_order` -> `ord`.

Two values `hopach` documents are not offered, because neither works: `d = "abseuclid"`, which `distancematrix()` has no branch for, and `ord = "co"`, which errors for every input.

On some datasets `hopach` collapses its tree all the way to a single cluster and cannot split it further, which ends the run with an error. Whether it happens depends on the data and on `dist`, so another `dist` clusters the same data.

Value

`HOPACHConfig` object.

Author(s)

EDG

References

van der Laan MJ, Pollard KS (2003). A new algorithm for hybrid hierarchical clustering with visualization and the bootstrap. *Journal of Statistical Planning and Inference*, 117(2), 275-303. doi:10.1016/S03783758(02)003889

Examples

```
hopach_config <- setup_HOPACH(dist = "euclid", max_levels = 3L)
hopach_config
```

setup_ICA *setup_ICA*

Description

Setup ICA config.

Usage

```
setup_ICA(
  k = 3L,
  type = "parallel",
  fun = "logcosh",
  alpha = 1,
  row_norm = TRUE,
  maxit = 100L,
  tol = 1e-04,
  features = NULL
)
```

Arguments

k	Integer [1, Inf): Number of components.
type	Character {"parallel", "deflation"}: Type of ICA.
fun	Character {"logcosh", "exp"}: ICA function.
alpha	Numeric [1, 2]: Used in approximation to neg-entropy with `fun = "logcosh"`.
row_norm	Logical: If TRUE, standardize each case across features before ICA, i.e. subtract each case's mean and divide by its standard deviation. Appropriate when the features share units and each case's overall level and scale are a nuisance rather than signal, as for spectra or assays whose total intensity varies by sample. It spends two degrees of freedom per case, so its cost is roughly `2 / ncol(x)`: negligible for hundreds of features, but substantial for a handful, where each case's level and scale are more likely to be signal. Set FALSE for narrow or mixed-unit data.
maxit	Integer [1, Inf): Maximum number of iterations.
tol	Numeric [0, Inf): Tolerance.
features	Optional Character vector: Names of at least 2 distinct feature columns to decompose. `NULL` decomposes all numeric features.

Value

ICAConfig object.

Author(s)

EDG

Examples

```
ica_config <- setup_ICA(k = 3L)
ica_config
```

`setup_Isomap` *Setup Isomap config.*

Description

Setup Isomap config.

Usage

```
setup_Isomap(
  k = 2L,
  dist_method = "euclidean",
  nsd = 0L,
  path = "shortest",
  features = NULL
)
```

Arguments

`k`	Integer [1, Inf): Number of components.
`dist_method`	Character {"euclidean", "manhattan"}: Distance method.
`nsd`	Integer [0, Inf): Number of shortest dissimilarities retained.
`path`	Character {"shortest", "extended"}: Path argument for `vegan::isomap`.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to decompose. `NULL` decomposes all numeric features.

Value

IsomapConfig object.

Author(s)

EDG

Examples

```
isomap_config <- setup_Isomap(k = 3L)
isomap_config
```

setup_Isotonic *Setup Isotonic Hyperparameters*

Description

Setup hyperparameters for Isotonic Regression.

Usage

```
setup_Isotonic(ifw = FALSE)
```

Arguments

ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Isotonic regression has no hyperparameters.

Value

IsotonicHyperparameters object.

Author(s)

EDG

Examples

```
isotonic_hyperparams <- setup_Isotonic(ifw = TRUE)
isotonic_hyperparams
```

setup_KFold *Setup KFold Resampler*

Description

Setup KFold Resampler

Usage

```
setup_KFold(
  n_resamples = 10L,
  stratify_var = NULL,
  strat_n_bins = 4L,
  id_strat = NULL,
  seed = NULL
)
```

Arguments

n_resamples	Integer [1, Inf): Number of resamples to make.
stratify_var	Optional Character: Variable to stratify by.
strat_n_bins	Integer [1, Inf): Number of bins to stratify by.
id_strat	Optional Character: Name of the column holding per-case grouping IDs, e.g. subject IDs when the dataset contains repeated measurements. Cases sharing an ID stay in the same resample, so a case can only be present in the training or the test set, not both. Requires that `resample()` be given the data frame the column lives in. train takes the column as a case identifier, so it groups the resamples by it and excludes it from the features.
seed	Optional Integer [0, Inf): Random seed.

Value

`KFoldConfig` object.

Author(s)

EDG

Examples

```
tenfold_resampler <- setup_KFold(n_resamples = 10L, seed = 2026L)
tenfold_resampler
```

---

setup_KMeans *Setup KMeansConfig*

Description

Setup KMeansConfig

Usage

```
setup_KMeans(k = 3L, dist = "euclidean", features = NULL)
```

Arguments

k	Integer [1, Inf): Number of clusters.
dist	Character {"euclidean", "manhattan"}: Distance measure to use.
features	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Value

KMeansConfig object.

Author(s)

EDG

Examples

```
kmeans_config <- setup_KMeans(k = 4L, dist = "euclidean")
kmeans_config
```

---

setup_KNN *Setup KNN Hyperparameters*

Description

Setup hyperparameters for k-Nearest Neighbors training.

Usage

```
setup_KNN(k = 7L, kernel = "optimal", distance = 2, scale = TRUE, ifw = FALSE)
```

Arguments

k	(Tunable) Integer [1, Inf): Number of neighbors. Must be less than the number of training cases.
kernel	(Tunable) Character {"rectangular", "triangular", "epanechnikov", "biweight", "triweight", "cos", "inv", "gaussian", "rank", "optimal"}: Kernel used to weight neighbors by distance.
distance	(Tunable) Numeric (0, Inf): Parameter of the Minkowski distance.
scale	Logical: If TRUE, scale features to unit variance before computing distances.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Both outcome types are fit with kknn::train.kknn, which selects between regression and classification from the outcome. Factors are one-hot encoded by rtemis first, as an algorithm-internal preprocessor that is re-applied at predict time: the backend's own factor handling resolves a contrast function off the search path, which is not reachable when kknn is loaded but not attached, as a suggested package is.

kknn provides no case weights, so ifw cannot be honored: enabling it makes training abort.

Value

KNNHyperparameters object.

Author(s)

EDG

Examples

```
knn_hyperparams <- setup_KNN(k = 11L, kernel = "rectangular")
knn_hyperparams
```

`setup_LightCART` *Setup LightCART Hyperparameters*

Description

Setup hyperparameters for LightCART training.

Usage

```
setup_LightCART(
  num_leaves = 32L,
  max_depth = -1L,
  lambda_l1 = 0,
  lambda_l2 = 0,
  max_cat_threshold = 32L,
  min_data_per_group = 100L,
  linear_tree = FALSE,
  objective = NULL,
  force_col_wise = TRUE,
  feature_fraction = 1,
  feature_fraction_bynode = 1,
  feature_fraction_seed = 2L,
  deterministic = FALSE,
  objective_seed = 5L,
  force_row_wise = FALSE,
  histogram_pool_size = -1,
  cegb_tradeoff = 1,
  cegb_penalty_split = 0,
  cegb_penalty_feature_lazy = NULL,
  cegb_penalty_feature_coupled = NULL,
  use_quantized_grad = FALSE,
  num_grad_quant_bins = NULL,
  quant_train_renew_leaf = NULL,
  stochastic_rounding = NULL,
  monotone_constraints = NULL,
  monotone_constraints_method = NULL,
  monotone_penalty = NULL,
  interaction_constraints = NULL,
  feature_contri = NULL,
  min_data_in_leaf = 20L,
  min_sum_hessian_in_leaf = 0.001,
  min_gain_to_split = 0,
  max_delta_step = 0,
```

```
  path_smooth = 0,
  extra_trees = FALSE,
  extra_seed = 6L,
  linear_lambda = NULL,
  max_bin = 255L,
  min_data_in_bin = 3L,
  use_missing = TRUE,
  zero_as_missing = FALSE,
  cat_l2 = 10,
  cat_smooth = 10,
  max_cat_to_onehot = 4L,
  alpha = NULL,
  tweedie_variance_power = NULL,
  fair_c = NULL,
  poisson_max_delta_step = NULL,
  sigmoid = NULL,
  boost_from_average = NULL,
  reg_sqrt = NULL,
  ifw = FALSE
)
```

Arguments

num_leaves	(Tunable) Integer [1, Inf): Maximum number of leaves in one tree.
max_depth	(Tunable) Integer: Maximum depth of trees. -1 = no limit.
lambda_l1	(Tunable) Numeric [0, Inf): L1 regularization.
lambda_l2	(Tunable) Numeric [0, Inf): L2 regularization.
max_cat_threshold	(Tunable) Integer [1, Inf): Maximum number of categories for categorical features.
min_data_per_group	(Tunable) Integer [1, Inf): Minimum number of observations per categorical group.
linear_tree	(Tunable) Logical: If TRUE, use linear trees.
objective	Optional Character: Objective function. NULL = set from outcome type.
force_col_wise	Logical: Force column-wise histogram building (CPU only).
feature_fraction	(Tunable) Numeric (0, 1]: Fraction of features sampled per tree.
feature_fraction_bynode	(Tunable) Numeric (0, 1]: Fraction of features sampled at each node, on top of the per-tree fraction.
feature_fraction_seed	Integer: Random seed for feature sampling.
deterministic	Logical: Force reproducible results at some cost in speed.
objective_seed	Integer: Random seed for the objective, where it draws.

`force_row_wise` Logical: Force row-wise histogram building. Cheaper in memory than column-wise, slower per iteration.

`histogram_pool_size`
Numeric: Memory cap in MB for the histogram cache; -1 leaves it uncapped.

`cegb_tradeoff` (Tunable) Numeric [0, Inf): Overall multiplier on every cost-efficiency penalty. Its scale is not comparable between boosting modes.

`cegb_penalty_split`
(Tunable) Numeric [0, Inf): Cost charged per split, in units of loss reduction.

`cegb_penalty_feature_lazy`
Optional List: Per-feature cost charged the first time a case uses it, in [0, Inf). One vector applied to every tree, or one vector per tree. NULL charges no cost.

`cegb_penalty_feature_coupled`
Optional List: Per-feature cost charged the first time any case uses it, in [0, Inf). One vector applied to every tree, or one vector per tree. NULL charges no cost.

`use_quantized_grad`
Logical: Discretize gradients before finding splits. Faster, and changes the fit.

`num_grad_quant_bins`
(Tunable) Optional Integer [2, Inf): Bins the gradients are discretized into. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of 4.

`quant_train_renew_leaf`
Optional Logical: Recompute leaf values from the exact gradients after each iteration. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of FALSE.

`stochastic_rounding`
Optional Logical: Round gradients stochastically rather than to nearest. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of TRUE.

`monotone_constraints`
Optional Integer [-1, 1] vector: Per-feature monotonicity: 1 increasing, -1 decreasing, 0 unconstrained. One value per feature. NULL constrains no feature.

`monotone_constraints_method`
Optional Character {"basic", "intermediate", "advanced"}: How monotonicity is enforced. Requires `monotone_constraints`. NULL uses LightGBM's default, "basic".

`monotone_penalty`
(Tunable) Optional Numeric [0, Inf): Depth penalty on splits that would break monotonicity. Requires `monotone_constraints`. NULL uses LightGBM's default of 0.

`interaction_constraints`
Optional Character: Feature groups permitted to interact, in LightGBM's own bracketed form, e.g. "[0,1],[2,3]". NULL permits every interaction.

`feature_contri` Optional Numeric vector: Per-feature multiplier on split gain, one value per feature. NULL uses 1 for every feature.

`min_data_in_leaf`
(Tunable) Integer [0, Inf): Minimum number of cases in a leaf.

`min_sum_hessian_in_leaf`
(Tunable) Numeric [0, Inf): Minimum sum of hessians in a leaf: the smooth counterpart of a case count.

`min_gain_to_split`
: (Tunable) Numeric [0, Inf): Minimum loss reduction a split must produce for the split to be made.

`max_delta_step`
: (Tunable) Numeric: Cap on each leaf's output; 0 or less leaves it uncapped.

`path_smooth`
: (Tunable) Numeric [0, Inf): Smooth each leaf towards its parent, more strongly the fewer cases it holds.

`extra_trees`
: (Tunable) Logical: Choose one split threshold per feature at random rather than the best.

`extra_seed`
: Integer: Random seed for the thresholds `extra_trees` draws.

`linear_lambda`
: (Tunable) Optional Numeric [0, Inf): L2 penalty on the linear models fitted at the leaves. Requires `linear_tree = TRUE`. NULL uses LightGBM's default of 0.

`max_bin`
: (Tunable) Integer [2, Inf): Most bins a feature is discretized into. Fewer is faster and more regularized.

`min_data_in_bin`
: (Tunable) Integer [1, Inf): Minimum number of cases per bin.

`use_missing`
: Logical: Handle missing values. FALSE ignores them.

`zero_as_missing`
: Logical: Treat zeros as missing, rather than only NA.

`cat_l2`
: (Tunable) Numeric [0, Inf): L2 regularization on a categorical split.

`cat_smooth`
: (Tunable) Numeric [0, Inf): Smoothing over categorical levels, which reduces the effect of rare ones.

`max_cat_to_onehot`
: (Tunable) Integer [1, Inf): Below this many levels, split one-versus-rest rather than by partition.

`alpha`
: (Tunable) Optional Numeric (0, Inf): Huber delta or quantile level. Requires `objective` to be "huber" or "quantile". NULL uses LightGBM's default of 0.9.

`tweedie_variance_power`
: (Tunable) Optional Numeric [1, 2): Tweedie power: 1 is Poisson-like, near 2 is gamma-like. Requires `objective = "tweedie"`. NULL uses LightGBM's default of 1.5.

`fair_c`
: (Tunable) Optional Numeric (0, Inf): Fair loss scale. Requires `objective = "fair"`. NULL uses LightGBM's default of 1.

`poisson_max_delta_step`
: (Tunable) Optional Numeric (0, Inf): Step cap safeguarding Poisson optimization. Requires `objective = "poisson"`. NULL uses LightGBM's default of 0.7.

`sigmoid`
: (Tunable) Optional Numeric (0, Inf): Sigmoid slope, for a binary or one-vs-all objective. NULL uses LightGBM's default of 1.

`boost_from_average`
: Optional Logical: Start boosting from the constant score that best fits the outcome under the objective, such as its mean for regression. NULL uses LightGBM's default of TRUE.

reg_sqrt	Optional Logical: Fit the square root of the outcome and square the prediction back, for the "regression", "regression_l1", "fair", "quantile" or "mape" objective. NULL uses LightGBM's default of FALSE.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Get more information from lightgbm::lgb.train.

Value

LightCARTHyperparameters object.

Author(s)

EDG

Examples

```
lightcart_hyperparams <- setup_LightCART(num_leaves = 32L, ifw = FALSE)
lightcart_hyperparams
```

---

setup_LightGBM *Setup LightGBM Hyperparameters*

Description

Setup hyperparameters for LightGBM training.

Usage

```
setup_LightGBM(
  max_nrounds = 1000L,
  force_nrounds = NULL,
  early_stopping_rounds = 10L,
  num_leaves = 8L,
  max_depth = -1L,
  learning_rate = 0.01,
  feature_fraction = 1,
  bagging_fraction = 1,
  bagging_freq = 1L,
  lambda_l1 = 0,
  lambda_l2 = 0,
  max_cat_threshold = 32L,
  min_data_per_group = 32L,
  linear_tree = FALSE,
  ifw = FALSE,
  objective = NULL,
```

```
  early_stopping_min_delta = 0,
  deterministic = FALSE,
  objective_seed = 5L,
  force_row_wise = FALSE,
  histogram_pool_size = -1,
  top_k = NULL,
  pos_bagging_fraction = 1,
  neg_bagging_fraction = 1,
  bagging_seed = 3L,
  feature_fraction_bynode = 1,
  feature_fraction_seed = 2L,
  cegb_tradeoff = 1,
  cegb_penalty_split = 0,
  cegb_penalty_feature_lazy = NULL,
  cegb_penalty_feature_coupled = NULL,
  use_quantized_grad = FALSE,
  num_grad_quant_bins = NULL,
  quant_train_renew_leaf = NULL,
  stochastic_rounding = NULL,
  monotone_constraints = NULL,
  monotone_constraints_method = NULL,
  monotone_penalty = NULL,
  interaction_constraints = NULL,
  feature_contri = NULL,
  min_data_in_leaf = 20L,
  min_sum_hessian_in_leaf = 0.001,
  min_gain_to_split = 0,
  max_delta_step = 0,
  path_smooth = 0,
  extra_trees = FALSE,
  extra_seed = 6L,
  linear_lambda = NULL,
  max_bin = 255L,
  min_data_in_bin = 3L,
  use_missing = TRUE,
  zero_as_missing = FALSE,
  cat_l2 = 10,
  cat_smooth = 10,
  max_cat_to_onehot = 4L,
  boosting = "gbdt",
  data_sample_strategy = "bagging",
  drop_rate = NULL,
  max_drop = NULL,
  skip_drop = NULL,
  uniform_drop = NULL,
  xgboost_dart_mode = NULL,
  drop_seed = NULL,
  top_rate = NULL,
```

```
  other_rate = NULL,
  alpha = NULL,
  tweedie_variance_power = NULL,
  fair_c = NULL,
  poisson_max_delta_step = NULL,
  sigmoid = NULL,
  boost_from_average = NULL,
  reg_sqrt = NULL,
  device_type = NULL,
  tree_learner = "serial",
  force_col_wise = TRUE
)
```

Arguments

max_nrounds	Integer [1, Inf): Maximum number of boosting rounds.
force_nrounds	Optional Integer [1, Inf): Use this many boosting rounds. Disables search for nrounds. NULL selects nrounds during tuning from the best iteration each tuning fit reports, up to max_nrounds: the fits stop early on their validation cases except under "dart" boosting, and a fit that reports no best iteration counts as 100 rounds.
early_stopping_rounds	Integer [1, Inf): Number of rounds without improvement to stop training.
num_leaves	(Tunable) Integer [1, Inf): Maximum number of leaves in one tree.
max_depth	(Tunable) Integer: Maximum depth of trees. -1 = no limit.
learning_rate	(Tunable) Numeric (0, 1]: Learning rate.
feature_fraction	(Tunable) Numeric (0, 1]: Fraction of features to use.
bagging_fraction	(Tunable) Numeric (0, 1]: Fraction of cases sampled per tree.
bagging_freq	(Tunable) Integer [1, Inf): Resample every this many iterations.
lambda_l1	(Tunable) Numeric [0, Inf): L1 regularization.
lambda_l2	(Tunable) Numeric [0, Inf): L2 regularization.
max_cat_threshold	(Tunable) Integer [1, Inf): Maximum number of categories for categorical features.
min_data_per_group	(Tunable) Integer [1, Inf): Minimum number of observations per categorical group.
linear_tree	(Tunable) Logical: If TRUE, use linear trees.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.
objective	Optional Character: Objective function. NULL = set from outcome type.
early_stopping_min_delta	(Tunable) Numeric [0, Inf): Smallest improvement that resets the early-stopping counter.

`deterministic` Logical: Force reproducible results at some cost in speed.

`objective_seed` Integer: Random seed for the objective, where it draws.

`force_row_wise` Logical: Force row-wise histogram building. Cheaper in memory than column-wise, slower per iteration.

`histogram_pool_size`
Numeric: Memory cap in MB for the histogram cache; -1 leaves it uncapped.

`top_k` (Tunable) Optional Integer [1, Inf): Features each machine votes for in the voting tree learner. Requires `tree_learner = "voting"`. NULL uses LightGBM's default of 20.

`pos_bagging_fraction`
(Tunable) Numeric (0, 1]: Fraction of positive cases sampled per tree, for a binary outcome.

`neg_bagging_fraction`
(Tunable) Numeric (0, 1]: Fraction of negative cases sampled per tree, for a binary outcome.

`bagging_seed` Integer: Random seed for bagging.

`feature_fraction_bynode`
(Tunable) Numeric (0, 1]: Fraction of features sampled at each node, on top of the per-tree fraction.

`feature_fraction_seed`
Integer: Random seed for feature sampling.

`cegb_tradeoff` (Tunable) Numeric [0, Inf): Overall multiplier on every cost-efficiency penalty. Its scale is not comparable between boosting modes.

`cegb_penalty_split`
(Tunable) Numeric [0, Inf): Cost charged per split, in units of loss reduction.

`cegb_penalty_feature_lazy`
Optional List: Per-feature cost charged the first time a case uses it, in [0, Inf). One vector applied to every tree, or one vector per tree. NULL charges no cost.

`cegb_penalty_feature_coupled`
Optional List: Per-feature cost charged the first time any case uses it, in [0, Inf). One vector applied to every tree, or one vector per tree. NULL charges no cost.

`use_quantized_grad`
Logical: Discretize gradients before finding splits. Faster, and changes the fit.

`num_grad_quant_bins`
(Tunable) Optional Integer [2, Inf): Bins the gradients are discretized into. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of 4.

`quant_train_renew_leaf`
Optional Logical: Recompute leaf values from the exact gradients after each iteration. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of FALSE.

`stochastic_rounding`
Optional Logical: Round gradients stochastically rather than to nearest. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of TRUE.

`monotone_constraints`
Optional Integer [-1, 1] vector: Per-feature monotonicity: 1 increasing, -1 decreasing, 0 unconstrained. One value per feature. NULL constrains no feature.

`monotone_constraints_method`
: Optional Character {"basic", "intermediate", "advanced"}: How monotonicity is enforced. Requires `monotone_constraints`. NULL uses LightGBM's default, "basic".

`monotone_penalty`
: (Tunable) Optional Numeric [0, Inf): Depth penalty on splits that would break monotonicity. Requires `monotone_constraints`. NULL uses LightGBM's default of 0.

`interaction_constraints`
: Optional Character: Feature groups permitted to interact, in LightGBM's own bracketed form, e.g. "[0,1],[2,3]". NULL permits every interaction.

`feature_contri`
: Optional Numeric vector: Per-feature multiplier on split gain, one value per feature. NULL uses 1 for every feature.

`min_data_in_leaf`
: (Tunable) Integer [0, Inf): Minimum number of cases in a leaf.

`min_sum_hessian_in_leaf`
: (Tunable) Numeric [0, Inf): Minimum sum of hessians in a leaf: the smooth counterpart of a case count.

`min_gain_to_split`
: (Tunable) Numeric [0, Inf): Minimum loss reduction a split must produce for the split to be made.

`max_delta_step`
: (Tunable) Numeric: Cap on each leaf's output; 0 or less leaves it uncapped.

`path_smooth`
: (Tunable) Numeric [0, Inf): Smooth each leaf towards its parent, more strongly the fewer cases it holds.

`extra_trees`
: (Tunable) Logical: Choose one split threshold per feature at random rather than the best.

`extra_seed`
: Integer: Random seed for the thresholds `extra_trees` draws.

`linear_lambda`
: (Tunable) Optional Numeric [0, Inf): L2 penalty on the linear models fitted at the leaves. Requires `linear_tree = TRUE`. NULL uses LightGBM's default of 0.

`max_bin`
: (Tunable) Integer [2, Inf): Most bins a feature is discretized into. Fewer is faster and more regularized.

`min_data_in_bin`
: (Tunable) Integer [1, Inf): Minimum number of cases per bin.

`use_missing`
: Logical: Handle missing values. FALSE ignores them.

`zero_as_missing`
: Logical: Treat zeros as missing, rather than only NA.

`cat_l2`
: (Tunable) Numeric [0, Inf): L2 regularization on a categorical split.

`cat_smooth`
: (Tunable) Numeric [0, Inf): Smoothing over categorical levels, which reduces the effect of rare ones.

`max_cat_to_onehot`
: (Tunable) Integer [1, Inf): Below this many levels, split one-versus-rest rather than by partition.

`boosting`
: Character {"gbdt", "rf", "dart"}: Boosting algorithm: gradient boosting, random forest, or DART (dropout).

`data_sample_strategy`
: Character {"bagging", "goss"}: How cases are subsampled: at random, or by gradient magnitude (GOSS). GOSS cannot be combined with bagging, so it requires `bagging_fraction = 1`.

`drop_rate`
: (Tunable) Optional Numeric [0, 1]: Fraction of existing trees to drop per iteration. Requires `boosting = "dart"`. NULL uses LightGBM's default of 0.1.

`max_drop`
: (Tunable) Optional Integer: Most trees dropped in one iteration; 0 or less removes the cap. Requires `boosting = "dart"`. NULL uses LightGBM's default of 50.

`skip_drop`
: (Tunable) Optional Numeric [0, 1]: Probability of skipping dropout for an iteration. Requires `boosting = "dart"`. NULL uses LightGBM's default of 0.5.

`uniform_drop`
: (Tunable) Optional Logical: Drop trees uniformly rather than weighted. Requires `boosting = "dart"`. NULL uses LightGBM's default of FALSE.

`xgboost_dart_mode`
: (Tunable) Optional Logical: Use the XGBoost DART normalization. Requires `boosting = "dart"`. NULL uses LightGBM's default of FALSE.

`drop_seed`
: Optional Integer: Random seed for choosing which trees to drop. Requires `boosting = "dart"`. NULL uses LightGBM's default of 4.

`top_rate`
: (Tunable) Optional Numeric [0, 1]: Share of large-gradient cases GOSS retains. Requires `data_sample_strategy = "goss"`. NULL uses LightGBM's default of 0.2.

`other_rate`
: (Tunable) Optional Numeric [0, 1]: Share of the remaining cases GOSS retains; with `top_rate` it must not exceed 1. Requires `data_sample_strategy = "goss"`. NULL uses LightGBM's default of 0.1.

`alpha`
: (Tunable) Optional Numeric (0, Inf): Huber delta or quantile level. Requires `objective` to be "huber" or "quantile". NULL uses LightGBM's default of 0.9.

`tweedie_variance_power`
: (Tunable) Optional Numeric [1, 2): Tweedie power: 1 is Poisson-like, near 2 is gamma-like. Requires `objective = "tweedie"`. NULL uses LightGBM's default of 1.5.

`fair_c`
: (Tunable) Optional Numeric (0, Inf): Fair loss scale. Requires `objective = "fair"`. NULL uses LightGBM's default of 1.

`poisson_max_delta_step`
: (Tunable) Optional Numeric (0, Inf): Step cap safeguarding Poisson optimization. Requires `objective = "poisson"`. NULL uses LightGBM's default of 0.7.

`sigmoid`
: (Tunable) Optional Numeric (0, Inf): Sigmoid slope, for a binary or one-vs-all objective. NULL uses LightGBM's default of 1.

`boost_from_average`
: Optional Logical: Start boosting from the constant score that best fits the outcome under the objective, such as its mean for regression. NULL uses LightGBM's default of TRUE.

`reg_sqrt`
: Optional Logical: Fit the square root of the outcome and square the prediction back, for the "regression", "regression_l1", "fair", "quantile" or "mape" objective. NULL uses LightGBM's default of FALSE.

device_type	Optional Character: Deprecated and ignored. Set the compute device in the execution_config passed to train, e.g. setup_FutureExecution(device = "cuda"); "gpu" is device = "opencl" there.
tree_learner	Character {"serial", "feature", "data", "voting"}: Tree learner type.
force_col_wise	Logical: Use only with CPU - If TRUE, force col-wise histogram building.

Details

Get more information from lightgbm::lgb.train. nrounds is auto-tuned using early stopping (up to max_nrounds) unless force_nrounds is set.

Value

LightGBMHyperparameters object.

Author(s)

EDG

Examples

```
lightgbm_hyperparams <- setup_LightGBM(
  max_nrounds = 500L,
  learning_rate = tune_over(0.001, 0.01, 0.05), ifw = TRUE
)
lightgbm_hyperparams
```

---

setup_LightRF *Setup LightRF Hyperparameters*

Description

Setup hyperparameters for LightRF training.

Usage

```
setup_LightRF(
  nrounds = 500L,
  num_leaves = 4096L,
  max_depth = -1L,
  feature_fraction = NULL,
  bagging_fraction = 0.623,
  lambda_l1 = 0,
  lambda_l2 = 0,
  max_cat_threshold = 32L,
  min_data_per_group = 32L,
  linear_tree = FALSE,
  ifw = FALSE,
```

```
  objective = NULL,
  early_stopping_min_delta = 0,
  deterministic = FALSE,
  objective_seed = 5L,
  force_row_wise = FALSE,
  histogram_pool_size = -1,
  top_k = NULL,
  pos_bagging_fraction = 1,
  neg_bagging_fraction = 1,
  bagging_seed = 3L,
  feature_fraction_bynode = 1,
  feature_fraction_seed = 2L,
  cegb_tradeoff = 1,
  cegb_penalty_split = 0,
  cegb_penalty_feature_lazy = NULL,
  cegb_penalty_feature_coupled = NULL,
  use_quantized_grad = FALSE,
  num_grad_quant_bins = NULL,
  quant_train_renew_leaf = NULL,
  stochastic_rounding = NULL,
  monotone_constraints = NULL,
  monotone_constraints_method = NULL,
  monotone_penalty = NULL,
  interaction_constraints = NULL,
  feature_contri = NULL,
  min_data_in_leaf = 20L,
  min_sum_hessian_in_leaf = 0.001,
  min_gain_to_split = 0,
  max_delta_step = 0,
  path_smooth = 0,
  extra_trees = FALSE,
  extra_seed = 6L,
  linear_lambda = NULL,
  max_bin = 255L,
  min_data_in_bin = 3L,
  use_missing = TRUE,
  zero_as_missing = FALSE,
  cat_l2 = 10,
  cat_smooth = 10,
  max_cat_to_onehot = 4L,
  alpha = NULL,
  tweedie_variance_power = NULL,
  fair_c = NULL,
  poisson_max_delta_step = NULL,
  sigmoid = NULL,
  boost_from_average = NULL,
  reg_sqrt = NULL,
  device_type = NULL,
```

```
  tree_learner = "serial",
  force_col_wise = TRUE
)
```

Arguments

`nrounds`	(Tunable) Integer [1, Inf): Number of boosting rounds.
`num_leaves`	(Tunable) Integer [1, Inf): Maximum number of leaves in one tree.
`max_depth`	(Tunable) Integer: Maximum depth of trees. -1 = no limit.
`feature_fraction`	(Tunable) Optional Numeric (0, 1]: Fraction of features to use. NULL derives it from the data: sqrt(n_features)/n_features for classification, 0.33 for regression.
`bagging_fraction`	(Tunable) Numeric (0, 1]: Fraction of cases sampled per tree.
`lambda_l1`	(Tunable) Numeric [0, Inf): L1 regularization.
`lambda_l2`	(Tunable) Numeric [0, Inf): L2 regularization.
`max_cat_threshold`	(Tunable) Integer [1, Inf): Maximum number of categories for categorical features.
`min_data_per_group`	(Tunable) Integer [1, Inf): Minimum number of observations per categorical group.
`linear_tree`	(Tunable) Logical: If TRUE, use linear trees.
`ifw`	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.
`objective`	Optional Character: Objective function. NULL = set from outcome type.
`early_stopping_min_delta`	(Tunable) Numeric [0, Inf): Smallest improvement that resets the early-stopping counter.
`deterministic`	Logical: Force reproducible results at some cost in speed.
`objective_seed`	Integer: Random seed for the objective, where it draws.
`force_row_wise`	Logical: Force row-wise histogram building. Cheaper in memory than column-wise, slower per iteration.
`histogram_pool_size`	Numeric: Memory cap in MB for the histogram cache; -1 leaves it uncapped.
`top_k`	(Tunable) Optional Integer [1, Inf): Features each machine votes for in the voting tree learner. Requires `tree_learner = "voting"`. NULL uses LightGBM's default of 20.
`pos_bagging_fraction`	(Tunable) Numeric (0, 1]: Fraction of positive cases sampled per tree, for a binary outcome.
`neg_bagging_fraction`	(Tunable) Numeric (0, 1]: Fraction of negative cases sampled per tree, for a binary outcome.
`bagging_seed`	Integer: Random seed for bagging.

`feature_fraction_bynode`
: (Tunable) Numeric (0, 1]: Fraction of features sampled at each node, on top of the per-tree fraction.

`feature_fraction_seed`
: Integer: Random seed for feature sampling.

`cegb_tradeoff`
: (Tunable) Numeric [0, Inf): Overall multiplier on every cost-efficiency penalty. Its scale is not comparable between boosting modes.

`cegb_penalty_split`
: (Tunable) Numeric [0, Inf): Cost charged per split, in units of loss reduction.

`cegb_penalty_feature_lazy`
: Optional List: Per-feature cost charged the first time a case uses it, in [0, Inf). One vector applied to every tree, or one vector per tree. NULL charges no cost.

`cegb_penalty_feature_coupled`
: Optional List: Per-feature cost charged the first time any case uses it, in [0, Inf). One vector applied to every tree, or one vector per tree. NULL charges no cost.

`use_quantized_grad`
: Logical: Discretize gradients before finding splits. Faster, and changes the fit.

`num_grad_quant_bins`
: (Tunable) Optional Integer [2, Inf): Bins the gradients are discretized into. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of 4.

`quant_train_renew_leaf`
: Optional Logical: Recompute leaf values from the exact gradients after each iteration. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of FALSE.

`stochastic_rounding`
: Optional Logical: Round gradients stochastically rather than to nearest. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of TRUE.

`monotone_constraints`
: Optional Integer [-1, 1] vector: Per-feature monotonicity: 1 increasing, -1 decreasing, 0 unconstrained. One value per feature. NULL constrains no feature.

`monotone_constraints_method`
: Optional Character {"basic", "intermediate", "advanced"}: How monotonicity is enforced. Requires `monotone_constraints`. NULL uses LightGBM's default, "basic".

`monotone_penalty`
: (Tunable) Optional Numeric [0, Inf): Depth penalty on splits that would break monotonicity. Requires `monotone_constraints`. NULL uses LightGBM's default of 0.

`interaction_constraints`
: Optional Character: Feature groups permitted to interact, in LightGBM's own bracketed form, e.g. "[0,1],[2,3]". NULL permits every interaction.

`feature_contri`
: Optional Numeric vector: Per-feature multiplier on split gain, one value per feature. NULL uses 1 for every feature.

`min_data_in_leaf`
: (Tunable) Integer [0, Inf): Minimum number of cases in a leaf.

`min_sum_hessian_in_leaf`
: (Tunable) Numeric [0, Inf): Minimum sum of hessians in a leaf: the smooth counterpart of a case count.

`min_gain_to_split`
: (Tunable) Numeric [0, Inf): Minimum loss reduction a split must produce for the split to be made.

`max_delta_step`
: (Tunable) Numeric: Cap on each leaf's output; 0 or less leaves it uncapped.

`path_smooth`
: (Tunable) Numeric [0, Inf): Smooth each leaf towards its parent, more strongly the fewer cases it holds.

`extra_trees`
: (Tunable) Logical: Choose one split threshold per feature at random rather than the best.

`extra_seed`
: Integer: Random seed for the thresholds `extra_trees` draws.

`linear_lambda`
: (Tunable) Optional Numeric [0, Inf): L2 penalty on the linear models fitted at the leaves. Requires `linear_tree = TRUE`. NULL uses LightGBM's default of 0.

`max_bin`
: (Tunable) Integer [2, Inf): Most bins a feature is discretized into. Fewer is faster and more regularized.

`min_data_in_bin`
: (Tunable) Integer [1, Inf): Minimum number of cases per bin.

`use_missing`
: Logical: Handle missing values. FALSE ignores them.

`zero_as_missing`
: Logical: Treat zeros as missing, rather than only NA.

`cat_l2`
: (Tunable) Numeric [0, Inf): L2 regularization on a categorical split.

`cat_smooth`
: (Tunable) Numeric [0, Inf): Smoothing over categorical levels, which reduces the effect of rare ones.

`max_cat_to_onehot`
: (Tunable) Integer [1, Inf): Below this many levels, split one-versus-rest rather than by partition.

`alpha`
: (Tunable) Optional Numeric (0, Inf): Huber delta or quantile level. Requires `objective` to be "huber" or "quantile". NULL uses LightGBM's default of 0.9.

`tweedie_variance_power`
: (Tunable) Optional Numeric [1, 2): Tweedie power: 1 is Poisson-like, near 2 is gamma-like. Requires `objective = "tweedie"`. NULL uses LightGBM's default of 1.5.

`fair_c`
: (Tunable) Optional Numeric (0, Inf): Fair loss scale. Requires `objective = "fair"`. NULL uses LightGBM's default of 1.

`poisson_max_delta_step`
: (Tunable) Optional Numeric (0, Inf): Step cap safeguarding Poisson optimization. Requires `objective = "poisson"`. NULL uses LightGBM's default of 0.7.

`sigmoid`
: (Tunable) Optional Numeric (0, Inf): Sigmoid slope, for a binary or one-vs-all objective. NULL uses LightGBM's default of 1.

boost_from_average
: Optional Logical: Start boosting from the constant score that best fits the outcome under the objective, such as its mean for regression. NULL uses LightGBM's default of TRUE.

reg_sqrt
: Optional Logical: Fit the square root of the outcome and square the prediction back, for the "regression", "regression_l1", "fair", "quantile" or "mape" objective. NULL uses LightGBM's default of FALSE.

device_type
: Optional Character: Deprecated and ignored. Set the compute device in the `execution_config` passed to train, e.g. `setup_FutureExecution(device = "cuda")`; `"gpu"` is `device = "opencl"` there.

tree_learner
: Character {"serial", "feature", "data", "voting"}: Tree learner type.

force_col_wise
: Logical: Use only with CPU - If TRUE, force col-wise histogram building.

Details

Get more information from lightgbm::lgb.train. `boosting`, `learning_rate`, `bagging_freq`, and `early_stopping_rounds` are constants here: their values make `lightgbm` train a random forest. All of them are settable in setup_LightGBM.

Value

LightRFHyperparameters object.

Author(s)

EDG

Examples

```
lightrf_hyperparams <- setup_LightRF(nrounds = 1000L, ifw = FALSE)
lightrf_hyperparams
```

---

setup_LightRuleFit *Setup LightRuleFit Hyperparameters*

Description

Setup hyperparameters for LightRuleFit training.

Usage

```
setup_LightRuleFit(
  nrounds = 200L,
  num_leaves = 32L,
  max_depth = 4L,
  learning_rate = 0.1,
  bagging_fraction = 0.666,
```

```
  bagging_freq = 1L,
  lambda_l1 = 0,
  lambda_l2 = 0,
  objective = NULL,
  force_col_wise = TRUE,
  tree_learner = "serial",
  top_k = NULL,
  feature_fraction = 1,
  feature_fraction_bynode = 1,
  feature_fraction_seed = 2L,
  linear_tree = FALSE,
  linear_lambda = NULL,
  max_cat_threshold = 32L,
  min_data_per_group = 32L,
  alpha = NULL,
  deterministic = FALSE,
  objective_seed = 5L,
  force_row_wise = FALSE,
  histogram_pool_size = -1,
  pos_bagging_fraction = 1,
  neg_bagging_fraction = 1,
  bagging_seed = 3L,
  cegb_tradeoff = 1,
  cegb_penalty_split = 0,
  cegb_penalty_feature_lazy = NULL,
  cegb_penalty_feature_coupled = NULL,
  use_quantized_grad = FALSE,
  num_grad_quant_bins = NULL,
  quant_train_renew_leaf = NULL,
  stochastic_rounding = NULL,
  monotone_constraints = NULL,
  monotone_constraints_method = NULL,
  monotone_penalty = NULL,
  interaction_constraints = NULL,
  feature_contri = NULL,
  min_data_in_leaf = 20L,
  min_sum_hessian_in_leaf = 0.001,
  min_gain_to_split = 0,
  max_delta_step = 0,
  path_smooth = 0,
  extra_trees = FALSE,
  extra_seed = 6L,
  max_bin = 255L,
  min_data_in_bin = 3L,
  use_missing = TRUE,
  zero_as_missing = FALSE,
  cat_l2 = 10,
  cat_smooth = 10,
```

```
  max_cat_to_onehot = 4L,
  boosting = "gbdt",
  data_sample_strategy = "bagging",
  drop_rate = NULL,
  max_drop = NULL,
  skip_drop = NULL,
  uniform_drop = NULL,
  xgboost_dart_mode = NULL,
  drop_seed = NULL,
  top_rate = NULL,
  other_rate = NULL,
  tweedie_variance_power = NULL,
  fair_c = NULL,
  poisson_max_delta_step = NULL,
  sigmoid = NULL,
  boost_from_average = NULL,
  reg_sqrt = NULL,
  ifw_lightgbm = FALSE,
  alpha_glmnet = 1,
  lambda_glmnet = NULL,
  ifw_glmnet = FALSE,
  ifw = FALSE
)
```

Arguments

nrounds	(Tunable) Integer [1, Inf): Number of boosting rounds.
num_leaves	(Tunable) Integer [1, Inf): Maximum number of leaves in one tree.
max_depth	(Tunable) Integer: Maximum depth of trees.
learning_rate	(Tunable) Numeric (0, 1]: Learning rate.
bagging_fraction	(Tunable) Numeric (0, 1]: Fraction of cases sampled per tree.
bagging_freq	(Tunable) Integer [1, Inf): Resample every this many iterations.
lambda_l1	(Tunable) Numeric [0, Inf): L1 regularization.
lambda_l2	(Tunable) Numeric [0, Inf): L2 regularization.
objective	Optional Character: Objective function. NULL = set from outcome type.
force_col_wise	Logical: Force column-wise histogram building (CPU only).
tree_learner	Character {"serial", "feature", "data", "voting"}: Tree learner type.
top_k	(Tunable) Optional Integer [1, Inf): Features each machine votes for in the voting tree learner. Requires `tree_learner = "voting"`. NULL uses LightGBM's default of 20.
feature_fraction	(Tunable) Numeric (0, 1]: Fraction of features sampled per tree.
feature_fraction_bynode	(Tunable) Numeric (0, 1]: Fraction of features sampled at each node, on top of the per-tree fraction.

`feature_fraction_seed`
: Integer: Random seed for feature sampling.

`linear_tree`
: (Tunable) Logical: Fit linear models at leaves.

`linear_lambda`
: (Tunable) Optional Numeric [0, Inf): L2 penalty on the linear models fitted at the leaves. Requires `linear_tree = TRUE`. NULL uses LightGBM's default of 0.

`max_cat_threshold`
: (Tunable) Integer [1, Inf): Maximum number of split points for categorical features.

`min_data_per_group`
: (Tunable) Integer [1, Inf): Minimum number of cases per categorical group.

`alpha`
: (Tunable) Optional Numeric (0, Inf): Huber delta or quantile level. Requires `objective` to be "huber" or "quantile". NULL uses LightGBM's default of 0.9.

`deterministic`
: Logical: Force reproducible results at some cost in speed.

`objective_seed`
: Integer: Random seed for the objective, where it draws.

`force_row_wise`
: Logical: Force row-wise histogram building. Cheaper in memory than column-wise, slower per iteration.

`histogram_pool_size`
: Numeric: Memory cap in MB for the histogram cache; -1 leaves it uncapped.

`pos_bagging_fraction`
: (Tunable) Numeric (0, 1]: Fraction of positive cases sampled per tree, for a binary outcome.

`neg_bagging_fraction`
: (Tunable) Numeric (0, 1]: Fraction of negative cases sampled per tree, for a binary outcome.

`bagging_seed`
: Integer: Random seed for bagging.

`cegb_tradeoff`
: (Tunable) Numeric [0, Inf): Overall multiplier on every cost-efficiency penalty. Its scale is not comparable between boosting modes.

`cegb_penalty_split`
: (Tunable) Numeric [0, Inf): Cost charged per split, in units of loss reduction.

`cegb_penalty_feature_lazy`
: Optional List: Per-feature cost charged the first time a case uses it, in [0, Inf). One vector applied to every tree, or one vector per tree. NULL charges no cost.

`cegb_penalty_feature_coupled`
: Optional List: Per-feature cost charged the first time any case uses it, in [0, Inf). One vector applied to every tree, or one vector per tree. NULL charges no cost.

`use_quantized_grad`
: Logical: Discretize gradients before finding splits. Faster, and changes the fit.

`num_grad_quant_bins`
: (Tunable) Optional Integer [2, Inf): Bins the gradients are discretized into. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of 4.

`quant_train_renew_leaf`
: Optional Logical: Recompute leaf values from the exact gradients after each iteration. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of FALSE.

`stochastic_rounding`
: Optional Logical: Round gradients stochastically rather than to nearest. Requires `use_quantized_grad = TRUE`. NULL uses LightGBM's default of TRUE.

`monotone_constraints`
: Optional Integer [-1, 1] vector: Per-feature monotonicity: 1 increasing, -1 decreasing, 0 unconstrained. One value per feature. NULL constrains no feature.

`monotone_constraints_method`
: Optional Character {"basic", "intermediate", "advanced"}: How monotonicity is enforced. Requires `monotone_constraints`. NULL uses LightGBM's default, "basic".

`monotone_penalty`
: (Tunable) Optional Numeric [0, Inf): Depth penalty on splits that would break monotonicity. Requires `monotone_constraints`. NULL uses LightGBM's default of 0.

`interaction_constraints`
: Optional Character: Feature groups permitted to interact, in LightGBM's own bracketed form, e.g. "[0,1],[2,3]". NULL permits every interaction.

`feature_contri` Optional Numeric vector: Per-feature multiplier on split gain, one value per feature. NULL uses 1 for every feature.

`min_data_in_leaf`
: (Tunable) Integer [0, Inf): Minimum number of cases in a leaf.

`min_sum_hessian_in_leaf`
: (Tunable) Numeric [0, Inf): Minimum sum of hessians in a leaf: the smooth counterpart of a case count.

`min_gain_to_split`
: (Tunable) Numeric [0, Inf): Minimum loss reduction a split must produce for the split to be made.

`max_delta_step` (Tunable) Numeric: Cap on each leaf's output; 0 or less leaves it uncapped.

`path_smooth` (Tunable) Numeric [0, Inf): Smooth each leaf towards its parent, more strongly the fewer cases it holds.

`extra_trees` (Tunable) Logical: Choose one split threshold per feature at random rather than the best.

`extra_seed` Integer: Random seed for the thresholds `extra_trees` draws.

`max_bin` (Tunable) Integer [2, Inf): Most bins a feature is discretized into. Fewer is faster and more regularized.

`min_data_in_bin`
: (Tunable) Integer [1, Inf): Minimum number of cases per bin.

`use_missing` Logical: Handle missing values. FALSE ignores them.

`zero_as_missing`
: Logical: Treat zeros as missing, rather than only NA.

`cat_l2` (Tunable) Numeric [0, Inf): L2 regularization on a categorical split.

`cat_smooth` (Tunable) Numeric [0, Inf): Smoothing over categorical levels, which reduces the effect of rare ones.

`max_cat_to_onehot`
: (Tunable) Integer [1, Inf): Below this many levels, split one-versus-rest rather than by partition.

`boosting`
: Character {"gbdt", "rf", "dart"}: Boosting algorithm: gradient boosting, random forest, or DART (dropout).

`data_sample_strategy`
: Character {"bagging", "goss"}: How cases are subsampled: at random, or by gradient magnitude (GOSS). GOSS cannot be combined with bagging, so it requires `bagging_fraction = 1`.

`drop_rate`
: (Tunable) Optional Numeric [0, 1]: Fraction of existing trees to drop per iteration. Requires `boosting = "dart"`. NULL uses LightGBM's default of 0.1.

`max_drop`
: (Tunable) Optional Integer: Most trees dropped in one iteration; 0 or less removes the cap. Requires `boosting = "dart"`. NULL uses LightGBM's default of 50.

`skip_drop`
: (Tunable) Optional Numeric [0, 1]: Probability of skipping dropout for an iteration. Requires `boosting = "dart"`. NULL uses LightGBM's default of 0.5.

`uniform_drop`
: (Tunable) Optional Logical: Drop trees uniformly rather than weighted. Requires `boosting = "dart"`. NULL uses LightGBM's default of FALSE.

`xgboost_dart_mode`
: (Tunable) Optional Logical: Use the XGBoost DART normalization. Requires `boosting = "dart"`. NULL uses LightGBM's default of FALSE.

`drop_seed`
: Optional Integer: Random seed for choosing which trees to drop. Requires `boosting = "dart"`. NULL uses LightGBM's default of 4.

`top_rate`
: (Tunable) Optional Numeric [0, 1]: Share of large-gradient cases GOSS retains. Requires `data_sample_strategy = "goss"`. NULL uses LightGBM's default of 0.2.

`other_rate`
: (Tunable) Optional Numeric [0, 1]: Share of the remaining cases GOSS retains; with `top_rate` it must not exceed 1. Requires `data_sample_strategy = "goss"`. NULL uses LightGBM's default of 0.1.

`tweedie_variance_power`
: (Tunable) Optional Numeric [1, 2): Tweedie power: 1 is Poisson-like, near 2 is gamma-like. Requires `objective = "tweedie"`. NULL uses LightGBM's default of 1.5.

`fair_c`
: (Tunable) Optional Numeric (0, Inf): Fair loss scale. Requires `objective = "fair"`. NULL uses LightGBM's default of 1.

`poisson_max_delta_step`
: (Tunable) Optional Numeric (0, Inf): Step cap safeguarding Poisson optimization. Requires `objective = "poisson"`. NULL uses LightGBM's default of 0.7.

`sigmoid`
: (Tunable) Optional Numeric (0, Inf): Sigmoid slope, for a binary or one-vs-all objective. NULL uses LightGBM's default of 1.

`boost_from_average`
: Optional Logical: Start boosting from the constant score that best fits the outcome under the objective, such as its mean for regression. NULL uses LightGBM's default of TRUE.

`reg_sqrt`	Optional Logical: Fit the square root of the outcome and square the prediction back, for the "regression", "regression_l1", "fair", "quantile" or "mape" objective. NULL uses LightGBM's default of FALSE.
`ifw_lightgbm`	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in the LightGBM step.
`alpha_glmnet`	(Tunable) Numeric [0, 1]: Alpha for GLMNET.
`lambda_glmnet`	Optional Numeric [0, Inf) vector: Lambda for GLMNET. NULL = determined by cv.glmnet.
`ifw_glmnet`	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in the GLMNET step.
`ifw`	Logical: If TRUE, use Inverse Frequency Weighting in classification. This applies IFW to both LightGBM and GLMNET; cannot be combined with `ifw_lightgbm` or `ifw_glmnet`.

Details

Get more information from lightgbm::lgb.train.

Value

LightRuleFitHyperparameters object.

Author(s)

EDG

Examples

```
lightrulefit_hyperparams <- setup_LightRuleFit(nrounds = 300L, max_depth = 3L)
lightrulefit_hyperparams
```

---

setup_LINAD *Setup LINAD Hyperparameters*

Description

Setup hyperparameters for LINAD, the Linear Additive Tree. LINAD generalizes the decision tree and the regularized linear model: CART, the Additive Tree, and a penalized linear fit are each a special case of it, recovered exactly by the hyperparameter combinations given under Details.

Usage

```
setup_LINAD(
  max_leaves = 20L,
  min_cases_split = 2L,
  min_cases_leaf = 1L,
  min_cases_node_model = NULL,
  nvmax = NULL,
  forward_stop = NULL,
  lambda = NULL,
  alpha = NULL,
  learning_rate = 0.1,
  root_nvmax = NULL,
  root_lambda = NULL,
  root_alpha = NULL,
  root_learning_rate = 1,
  n_cuts = NULL,
  gamma = 0.1,
  split_binning = NULL,
  split_bin_type = "frequency",
  node_test = NULL,
  split_criterion = NULL,
  line_search = "expansion",
  constant_rule = "closed_form",
  ifw = FALSE,
  patience = NULL,
  force_max_leaves = FALSE,
  smooth_validation_curve = NULL,
  node_model = "forward",
  root_model = NULL,
  split_search = "stump",
  split_features = NULL,
  linear_features = NULL,
  global_features = NULL,
  line_search_max = 1000,
  root_forward_stop = NULL,
  root_node_test = NULL
)
```

Arguments

`max_leaves`	(Tunable) Integer [1, Inf): Largest number of terminal nodes to grow.
`min_cases_split`	(Tunable) Integer [2, Inf): Fewest cases a node may hold and still be considered for splitting.
`min_cases_leaf`	(Tunable) Integer [1, Inf): Fewest cases a split must leave on each side.
`min_cases_node_model`	(Tunable) Optional Integer [1, Inf): Fewest cases needed to fit a linear model at a node. Applies only when `node_model` is not "constant". NULL uses 10.

`nvmax` (Tunable) Optional Integer [1, Inf): Most terms forward selection may add beside the intercept. Applies only when `node_model` is "forward". NULL uses 3.

`forward_stop` (Tunable) Optional Character {"bic", "aic", "none"}: Criterion a term must improve to be added by forward selection. Applies only when `node_model` is "forward". NULL uses "bic".

`lambda` (Tunable) Optional Numeric [0, Inf): L2 penalty on the leaf models. Applies only when `node_model` is "forward", "ridge" or "elasticnet". NULL uses 0.05.

`alpha` (Tunable) Optional Numeric [0, 1]: Elastic-net mixing, 0 ridge to 1 lasso. Applies only when `node_model` is "elasticnet". NULL uses 1.

`learning_rate` (Tunable) Numeric (0, 1]: Shrinkage applied to every functional update.

`root_nvmax` (Tunable) Optional Integer [1, Inf): `nvmax` for the root model. NULL uses `nvmax`.

`root_lambda` (Tunable) Optional Numeric [0, Inf): `lambda` for the root model. NULL uses `lambda`.

`root_alpha` (Tunable) Optional Numeric [0, 1]: `alpha` for the root model. NULL uses `alpha`.

`root_learning_rate` (Tunable) Numeric [0, 1]: Shrinkage applied to the root model's slopes.

`n_cuts` (Tunable) Optional Integer [2, Inf): Cut points tried per feature; `split_bin_type` decides their spacing. Applies only when `split_search` is "exhaustive". NULL uses 20.

`gamma` (Tunable) Numeric [0, 1]: Weight a case retains in the branch it does not belong to. 0 is a hard partition.

`split_binning` (Tunable) Optional Integer [2, Inf): Discretize each numeric feature into this many bins and consider only bin boundaries as splits. Applies to both split searches.

`split_bin_type` (Tunable) Character {"frequency", "width"}: How bin edges are placed.

`node_test` (Tunable) Optional Character {"none", "aic", "bic"}: Information criterion a node's slopes must improve over the constant alone. Applies only when `node_model` is "ridge" or "elasticnet"; forward selection uses `forward_stop`. NULL uses "none".

`split_criterion` (Tunable) Optional Character {"mean", "linear"}: What the stump search scores a candidate side by. Applies only when `split_search` is "stump". NULL uses "mean".

`line_search` (Tunable) Character {"expansion", "child", "none"}: Scope of the Newton step for each update.

`constant_rule` (Tunable) Character {"closed_form", "least_squares"}: How a node's constant is computed. The two coincide for regression and differ only for classification.

`ifw` (Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

`patience` (Tunable) Optional Integer [1, Inf): Expansions without validation improvement before growth stops. Requires a validation set. NULL disables patience-based early stopping.

`force_max_leaves`
: Logical: If TRUE, keep every leaf grown instead of selecting a size on the validation set.

`smooth_validation_curve`
: Optional Logical: If TRUE, smooth the validation curve before reading its minimum. Applies only when `force_max_leaves` is FALSE. NULL does not smooth.

`node_model`
: Character {"forward", "ridge", "elasticnet", "constant"}: Model fitted at each node. "constant" is the intercept-only model every decision tree node carries; the others add a regularized linear model on top of it.

`root_model`
: Optional Character {"forward", "ridge", "elasticnet", "constant"}: Model fitted at the root. NULL uses `node_model`.

`split_search`
: Character {"stump", "exhaustive"}: How a split is chosen.

`split_features`
: Optional Character vector: Features that may define a split. NULL imposes no constraint, so every feature may.

`linear_features`
: Optional Character vector: Features that get a slope in the node models. NULL imposes no constraint, so every feature does.

`global_features`
: Optional Character vector: Linear features whose slope is shared by every leaf. Must be a subset of `linear_features` where that is set. NULL imposes no constraint, so no coefficient is pinned.

`line_search_max`
: Numeric (0, Inf): Largest absolute step the line search may take.

`root_forward_stop`
: (Tunable) Optional Character {"bic", "aic", "none"}: `forward_stop` for the root model. NULL uses `forward_stop`, or "bic" where that is NULL. Applies only when the root model is "forward".

`root_node_test`
: (Tunable) Optional Character {"none", "aic", "bic"}: `node_test` for the root model. NULL uses `node_test`, or "none" where that is NULL. Applies only when the root model is "ridge" or "elasticnet".

Details

Every node carries a constant, as any decision tree's nodes do. `node_model` selects the model fitted at each node: `"constant"` for that constant alone, or a regularized linear model on top of it. With `max_leaves` and `gamma`, these settings span a hyperparameter space that contains the following algorithms and the models between them.

Value

LINADHyperparameters object.

What LINAD generalizes

Each of the following is a special case of LINAD, recovered exactly:

A pure linear model `max_leaves = 1`. Nothing is split, so the model is the root's alone – ridge, elastic net or forward stepwise selection, whichever `node_model` names. Verified against a direct penalized solve.

CART `node_model = "constant"`, `gamma = 0`, `learning_rate = 1`, `line_search = "none"`. Constants only, a hard partition and no shrinkage. Fitted values match `rpart`'s to machine precision.

The Additive Tree The same, with `gamma > 0`. Soft weighting is precisely what separates the Additive Tree from CART (Luna and colleagues, 2019; see References).

LINAD Any `node_model` other than `"constant"`, with `max_leaves > 1` – a tree whose nodes each carry a linear model.

Every setting named above is tunable, so the data can select where a dataset belongs in that space: a linear generating process should select a full root fit and no splitting, while terminal nodes needing opposite-signed coefficients should select a split first. `root_learning_rate` is the dial for that first step – at 0 the root is the outcome's mean and the tree splits first, at 1 a full linear model is fitted before any split.

How it grows

LINAD fits a regularized linear model on the whole sample, splits on the gradient of the loss, fits a model on each side of the split, and repeats. Every update is linear, so the coefficients along a root-to-leaf path sum: the finished model is a tree with one linear model per leaf, optimized stagewise rather than fitted leaf by leaf. Prediction routes a case to its leaf and evaluates that leaf's coefficients.

Splits are found on the features as given, so a factor splits on a set of its levels; the node models use a reference-coded design matrix built internally. LINAD therefore needs no encoding preprocessor and handles factors directly.

`max_leaves` plays the role that the number of trees plays in gradient boosting, and like it is chosen on held-out data: the tree is grown to `max_leaves` and the size with the lowest validation loss is kept, unless `force_max_leaves` is TRUE. Pass `dat_validation` to train to control the set that choice is made on.

`gamma` lets every case carry weight into both branches, decaying as `gamma^depth`, so each node's model is pulled toward the fit to the rest of the data, which reduces variance.

`split_search = "exhaustive"` scores each candidate split by the loss after fitting both child models, rather than taking the best split of the gradient. It costs more per split and detects an interaction that changes another feature's slope without changing either side's mean, which leaves no trace in any one feature's marginal fit.

A node's model is fitted to what its parent left, so its slopes are the *change* to the parent's coefficients and `lambda` and `alpha` penalize that change. At `alpha = 1` the change is a lasso, so a feature whose coefficient does not need to move gets exactly zero and inherits its parent's value, so each feature's effect adapts or stays fixed per node. Ridge shrinks every change but zeroes none, so every coefficient moves at every node.

`node_test` selects, node by node, between a linear model and a constant: the slopes are kept only where they improve the node's information criterion, and the node carries a constant otherwise. The resulting tree carries coefficients only in regions where the response is locally linear. The test can reduce accuracy; the default is `"none"`.

It applies to the shrinking models only. Forward selection applies the same criterion per term through `forward_stop`, and a set of terms that each passes it also passes it jointly.

`split_criterion` sets what the stump scores a side by. `"mean"` scores the level its fit explains, the CART criterion, which does not detect curvature: a parabola's two halves have the same mean, so the split at the vertex scores near last. `"linear"` adds the slope in the split variable, at one further pass per feature per node, and on such a problem recovers the exhaustive search's fit. It does not detect the cross-variable case above, and where the change belongs to another feature it can select a split on the wrong one. The default is `"mean"`.

`split_binning` discretizes numeric features before either search, so a candidate split falls on a bin boundary rather than between any two distinct values; `n_cuts` then thins what is left, for the exhaustive search only. `split_bin_type` governs both: `"frequency"` spaces cuts evenly through the cases, `"width"` evenly through the feature's range.

Feature roles

A feature's behavior has two independent axes: whether it may **define a partition**, and whether its modeled **effect** is absent, shared by every subgroup, or free to change along a path. Three name-based selectors set them, and on each of the three NULL imposes no constraint, which means "all" for the first two and "nothing pinned" for the third:

`split_features` Features the tree may split on.

`linear_features` Features that receive a slope in the node models.

`global_features` Linear features whose slope is shared by every leaf: the root estimates it and no node below may change it. Must be a subset of `linear_features` where that is set.

The axes are independent, so a feature may split without entering any node model, enter every node model without ever splitting, do both, or neither. Roles are assigned to **source features**: naming a factor moves all of its encoded columns together, never one of them.

Declaring effect scope turns LINAD into a varying-coefficient model with a hierarchical form: a global baseline plus the node deviations accumulated down the path. `global` is a hard constraint on the coefficient and holds exactly at any `gamma`. `gamma` is a soft constraint, pulling every node's fit toward the fit to the rest of the data until, at `gamma = 1`, every coefficient is effectively global. The two are independent: `gamma` sets how far a node's adaptive coefficients may move, and `global_features` sets which coefficients may move.

Because a global coefficient is estimated at the root, `root_learning_rate` shrinks it: at 0 the root fits no slopes and a global feature carries no effect anywhere.

The constrained family has these reductions. No splits (`max_leaves = 1`) is a penalized linear model; `node_model = "constant"` is a tree of constants; making every linear feature global gives a tree with shared coefficients and node-specific constants; and leaving all three NULL reproduces unconstrained LINAD exactly.

The split search cost is linear in the number of candidate features, so restricting the split set reduces training time on wide data.

Learning rate and the root model

`learning_rate` shrinks the update made at each node. It does not shrink the root, which `root_learning_rate` governs and which defaults to 1: the root model is the initialization `F_0`, not a boosting step.

Consequently, for any `learning_rate`, the fit includes the full global linear model at the root. On data whose signal is largely linear, shrinking the tree's correction changes the metrics little, and different small learning rates give similar fits. On a signal the root cannot represent, such as a step with no linear component, the same sweep moves test R-squared from 0.58 to 0.97.

To shrink everything, including the root, set `root_learning_rate` to the same value. A small rate then under-fits: at `1e-6` the fit reduces to the constant that minimizes the loss (the weighted mean of the outcome, or half its log odds for a classification), which scores an R-squared of 0. `root_learning_rate` interpolates between that constant and the full linear model, never towards zero.

The `line_search` step does not counteract this. It is scale-free (for squared error it is the exact minimizer along the update direction, near 1 for any residual size), so the applied step stays proportional to `learning_rate`. Measured node updates scale linearly with it, by a factor of 50 across a 50-fold change in the rate.

`line_search` and `constant_rule` each select among alternative update rules – the scope of the Newton step, which frontier node is split next, and how a node's constant is computed. Each is tunable.

Author(s)

EDG

References

Luna JM, Gennatas ED, Ungar LH, Eaton E, Diffenderfer ES, Jensen ST, Simone CB 2nd, Friedman JH, Solberg TD, Valdes G (2019). Building more accurate decision trees with the additive tree. *Proceedings of the National Academy of Sciences*, 116(40), 19887-19893. doi:10.1073/pnas.1816748116

Examples

```
linad_hyperparams <- setup_LINAD(max_leaves = 10L, learning_rate = 0.1)
linad_hyperparams
# The Additive Tree is LINAD without the linear models
addtree_hyperparams <- setup_LINAD(node_model = "constant", gamma = 0.8)
addtree_hyperparams
```

---

setup_LINADForest *Setup LINADForest Hyperparameters*

Description

Setup hyperparameters for LINADForest: a bagged ensemble of Linear Additive Trees. Each tree is fitted on a bootstrap sample of the cases, restricted to `mtry_tree` of the features, searching `mtry_split` of them at each split, and chooses its own number of leaves on the cases its bootstrap left out.

Usage

```
setup_LINADForest(
  n_trees = 50L,
  mtry_split = NULL,
  mtry_tree = NULL,
  max_leaves = 20L,
  min_cases_split = 2L,
  min_cases_leaf = 1L,
  min_cases_node_model = NULL,
  nvmax = NULL,
  forward_stop = NULL,
  lambda = NULL,
  alpha = NULL,
  learning_rate = 1,
  root_nvmax = NULL,
  root_lambda = NULL,
  root_alpha = NULL,
  root_learning_rate = 1,
  n_cuts = NULL,
  gamma = 0.1,
  split_binning = NULL,
  split_bin_type = "frequency",
  node_test = NULL,
  split_criterion = NULL,
  line_search = "expansion",
  constant_rule = "closed_form",
  ifw = FALSE,
  patience = NULL,
  force_max_leaves = FALSE,
  smooth_validation_curve = NULL,
  node_model = "forward",
  root_model = NULL,
  split_search = "stump",
  split_features = NULL,
  linear_features = NULL,
  global_features = NULL,
  line_search_max = 1000,
  root_forward_stop = NULL,
  root_node_test = NULL
)
```

Arguments

n_trees	(Tunable) Integer [1, Inf): Number of trees in the forest.
mtry_split	(Tunable) Optional Integer [1, Inf): Features sampled at each split search. NULL scans every feature the tree holds.
mtry_tree	(Tunable) Optional Integer [1, Inf): Features available to a whole tree. NULL gives every tree every feature.

`max_leaves` (Tunable) Integer [1, Inf): Largest number of terminal nodes to grow in each tree.

`min_cases_split` (Tunable) Integer [2, Inf): Fewest cases a node may hold and still be considered for splitting.

`min_cases_leaf` (Tunable) Integer [1, Inf): Fewest cases a split must leave on each side.

`min_cases_node_model` (Tunable) Optional Integer [1, Inf): Fewest cases needed to fit a linear model at a node. Applies only when `node_model` fits one. NULL uses 10.

`nvmax` (Tunable) Optional Integer [1, Inf): Most terms forward selection may add beside the intercept. Applies only when `node_model` is "forward". NULL uses 3.

`forward_stop` (Tunable) Optional Character {"bic", "aic", "none"}: Criterion a term must improve to be added by forward selection. Applies only when `node_model` is "forward". NULL uses "bic".

`lambda` (Tunable) Optional Numeric [0, Inf): L2 penalty on the node models. Applies only when `node_model` is "forward", "ridge" or "elasticnet". NULL uses 0.05.

`alpha` (Tunable) Optional Numeric [0, 1]: Elastic-net mixing, 0 ridge to 1 lasso. Applies only when `node_model` is "elasticnet". NULL uses 1.

`learning_rate` (Tunable) Numeric (0, 1]: Shrinkage applied to every functional update.

`root_nvmax` (Tunable) Optional Integer [1, Inf): `nvmax` for the root model. NULL uses `nvmax`.

`root_lambda` (Tunable) Optional Numeric [0, Inf): `lambda` for the root model. NULL uses `lambda`.

`root_alpha` (Tunable) Optional Numeric [0, 1]: `alpha` for the root model. NULL uses `alpha`.

`root_learning_rate` (Tunable) Numeric [0, 1]: Shrinkage applied to the root model's slopes.

`n_cuts` (Tunable) Optional Integer [2, Inf): Cut points tried per feature. Applies only when `split_search` is "exhaustive". NULL uses 20.

`gamma` (Tunable) Numeric [0, 1]: Weight a case retains in the branch it does not belong to. 0 is a hard partition.

`split_binning` (Tunable) Optional Integer [2, Inf): Discretize each numeric feature into this many bins and consider only bin boundaries as splits.

`split_bin_type` (Tunable) Character {"frequency", "width"}: How bin edges are placed.

`node_test` (Tunable) Optional Character {"none", "aic", "bic"}: Information criterion a node's slopes must improve over the constant alone. Applies only when `node_model` is "ridge" or "elasticnet"; forward selection uses `forward_stop`. NULL uses "none".

`split_criterion` (Tunable) Optional Character {"mean", "linear"}: What the stump search scores a candidate side by. Applies only when `split_search` is "stump". NULL uses "mean".

`line_search` (Tunable) Character {"expansion", "child", "none"}: Scope of the Newton step for each update.

`constant_rule` (Tunable) Character {"closed_form", "least_squares"}: How a node's constant is computed.

`ifw` (Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

`patience` (Tunable) Optional Integer [1, Inf): Expansions without validation improvement before growth stops. Requires a validation set. NULL disables patience-based early stopping.

`force_max_leaves` Logical: If TRUE, grow every tree to `max_leaves` instead of selecting a size on its out-of-bag cases.

`smooth_validation_curve` Optional Logical: If TRUE, smooth the out-of-bag curve before reading its minimum. Applies only when `force_max_leaves` is FALSE. NULL does not smooth.

`node_model` Character {"forward", "ridge", "elasticnet", "constant"}: Model fitted at each node.

`root_model` Optional Character {"forward", "ridge", "elasticnet", "constant"}: Model fitted at each tree's root. NULL uses `node_model`.

`split_search` Character {"stump", "exhaustive"}: How a split is chosen.

`split_features` Optional Character vector: Features that may define a split. NULL imposes no constraint, so every feature may. `mtry_split` samples within this set.

`linear_features` Optional Character vector: Features that get a slope in the node models. NULL imposes no constraint, so every feature does.

`global_features` Optional Character vector: Linear features whose slope is shared by every leaf of a tree. Must be a subset of `linear_features` where that is set. NULL imposes no constraint.

`line_search_max` Numeric (0, Inf): Largest absolute step the line search may take.

`root_forward_stop` (Tunable) Optional Character {"bic", "aic", "none"}: `forward_stop` for the root model. NULL uses `forward_stop`, or "bic" where that is NULL. Applies only when the root model is "forward".

`root_node_test` (Tunable) Optional Character {"none", "aic", "bic"}: `node_test` for the root model. NULL uses `node_test`, or "none" where that is NULL. Applies only when the root model is "ridge" or "elasticnet".

Details

Every tree-level hyperparameter of setup_LINAD is available here and tunable. The size of each tree is selected per tree from its out-of-bag cases, so the defaults are usable without tuning.

Value

LINADForestHyperparameters object.

How the trees differ

Three sources of variation, each acting at a different level:

The bootstrap Each tree is fitted on `n` cases drawn with replacement, so roughly 37% of cases are out-of-bag for any given tree.

`mtry_tree` Each tree is restricted to a random subset of the features, for its splits and its node models alike. A tree's coefficients then cover only the features it held.

`mtry_split` Each split search sees a fresh sample of the features the tree holds, which decorrelates the trees further. If that sample finds no split that reduces the loss, the node is searched again over every feature, so `mtry_split` changes which split is made and never whether a node can split.

Both `mtry_*` default to NULL, meaning every feature.

Choosing each tree's size

A LINAD tree picks its number of leaves on held-out data, as gradient boosting picks its number of trees. Each tree selects its size on its own out-of-bag cases, so no validation set is required. `force_max_leaves = TRUE` turns the selection off and grows every tree to `max_leaves`.

The same out-of-bag predictions give the fitted forest an error estimate with no resampling, reported by `print()`.

Defaults that differ from a single tree

`learning_rate` defaults to 1, where a single tree uses 0.1. A single LINAD tree shrinks each node update to control its variance; in a forest, averaging controls variance, so each tree is a strong learner, as a random forest's trees are unpruned.

Workers

LINADForest parallelizes across trees, by process rather than by thread. `n_workers_algorithm` therefore has an effect only when the run is not already dispatching: inside a parallel outer fold the forest runs its trees sequentially, since a fold may not dispatch again. For a threaded backend such as Ranger the same setting means threads within each fold.

Author(s)

EDG

References

Wager S, Hastie T, Efron B (2014). Confidence intervals for random forests: the jackknife and the infinitesimal jackknife. *Journal of Machine Learning Research*, 15, 1625-1651.

Examples

```
linadforest_hyperparams <- setup_LINADForest(n_trees = 25L)
linadforest_hyperparams
```

setup_LinearSVM *Setup LinearSVM Hyperparameters*

Description

Setup hyperparameters for LinearSVM training.

Usage

```
setup_LinearSVM(cost = 1, ifw = FALSE)
```

Arguments

cost	(Tunable) Numeric (0, Inf): Cost of constraints violation.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Get more information from e1071::svm.

Value

LinearSVMHyperparameters object.

Author(s)

EDG

Examples

```
linear_svm_hyperparams <- setup_LinearSVM(cost = 0.5, ifw = TRUE)
linear_svm_hyperparams
```

setup_LOOCV *Setup LOOCV Resampler*

Description

Leave-one-out cross-validation: one resample per case. `n_resamples` is not a parameter here because only the data can say how many cases there are – `resample()` fills it in.

Usage

```
setup_LOOCV()
```

Value

LOOCVConfig object.

Author(s)

EDG

Examples

```
loocv_resampler <- setup_LOOCV()
loocv_resampler
```

---

setup_MARS *Setup MARS Hyperparameters*

Description

Setup hyperparameters for Multivariate Adaptive Regression Splines training.

Usage

```
setup_MARS(
  degree = 1L,
  penalty = NULL,
  nk = NULL,
  nprune = NULL,
  thresh = 0.001,
  minspan = 0L,
  endspan = 0L,
  newvar_penalty = 0,
  fast_k = 20L,
  pmethod = "backward",
  nfold = 0L,
  ncross = 1L,
  stratify = TRUE,
  fast_beta = 1,
  ifw = FALSE
)
```

Arguments

degree	(Tunable) Integer [1, Inf): Maximum degree of interaction. 1 builds an additive model with no interaction terms.
penalty	(Tunable) Optional Numeric [-1, Inf): Generalized Cross Validation penalty per knot. NULL uses 3 when degree is greater than 1 and 2 otherwise.

nk	(Tunable) Optional Integer [1, Inf): Maximum number of terms, including the intercept, created by the forward pass. NULL lets earth derive it from the number of features.
nprune	(Tunable) Optional Integer [1, Inf): Maximum number of terms, including the intercept, retained after pruning. NULL places no cap on the terms retained, which pruning still selects by pmethod.
thresh	(Tunable) Numeric [0, 1): Forward pass stopping threshold: stop once adding a term changes R-squared by less than this.
minspan	(Tunable) Integer (-Inf, Inf): Minimum number of observations between knots. 0 derives the value internally, and a negative value instead sets the maximum number of equally spaced knots per feature.
endspan	(Tunable) Integer [0, Inf): Minimum number of observations before the first and after the final knot. 0 derives the value internally.
newvar_penalty	(Tunable) Numeric [0, Inf): Penalty for adding a feature not already in the model during the forward pass.
fast_k	(Tunable) Integer [0, Inf): Maximum number of parent terms considered at each step of the forward pass. 0 disables Fast MARS.
pmethod	Character {"backward", "none", "exhaustive", "forward", "seqrep", "cv"}: Pruning method. "cv" requires nfold. Multiclass classification allows only "backward" and "none".
nfold	Integer [0, Inf): Number of cross-validation folds used to estimate out-of-fold R-squared. 0 disables cross-validation.
ncross	Integer [1, Inf): Number of times the nfold cross-validation is repeated.
stratify	Logical: If TRUE, stratify the cross-validation folds on the outcome.
fast_beta	Numeric [0, 1]: Fast MARS aging coefficient.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Get more information from earth::earth.

get_varimp() returns earth's three importance criteria, in this order: importance (the GCV criterion), rss (the RSS criterion), and subset_proportion (the fraction of pruning subsets that retain the feature). See varimp_super in train_MARS.R for how each is derived.

Value

MARSHyperparameters object.

Author(s)

EDG

Examples

```
mars_hyperparams <- setup_MARS(degree = 2L, nprune = 10L)
mars_hyperparams
```

setup_MDS *Setup MDS config.*

Description

Stress-minimizing multidimensional scaling with `vegan::monoMDS()`: finds `k` coordinates per case whose distances match the dissimilarities between cases as closely as possible, measured by Kruskal's stress-1. For classical (Torgerson) scaling, which solves an eigenproblem instead, see setup_PCoA.

Usage

```
setup_MDS(
  k = 2L,
  dist_method = "euclidean",
  model = "global",
  nstart = 20L,
  max_iter = 200L,
  features = NULL
)
```

Arguments

`k`	Integer [1, Inf): Number of components. (passed to `monoMDS` `k`)
`dist_method`	Character {"euclidean", "manhattan", "chord", "bray", "canberra", "jaccard", "kulczynski", "hellinger"}: Dissimilarity between cases, computed with `vegan::vegdist()`. "bray", "canberra", "jaccard", "kulczynski" and "hellinger" require non-negative data.
`model`	Character {"global", "local", "linear"}: "global" is nonmetric MDS (one monotone regression); "local" fits one monotone regression per case; "linear" is metric MDS. (passed to `monoMDS` `model`)
`nstart`	Integer [1, Inf): Number of starting configurations. 1 starts from the classical scaling solution only.
`max_iter`	Integer [1, Inf): Maximum number of iterations per start. (passed to `monoMDS` `maxit`)
`features`	Optional Character vector: Names of at least 2 distinct feature columns to decompose. `NULL` decomposes all numeric features.

Details

Each start runs `vegan::monoMDS()` from its own initial configuration, and the one with the lowest stress is kept. The first start is the classical scaling solution, `stats::cmdscale()`; if that has fewer than `k` dimensions with positive eigenvalues, the missing ones are random. Every other start is uniform on [-1, 1]. The random values come from R's random number generator, so the execution config's seed reproduces the fit. The final configuration is centered, scaled and rotated to its principal axes, so its components are uncorrelated, but the first `j` components of a `k`-dimensional solution are not the `j`-dimensional solution.

MDS cannot be applied to new data, so it cannot be used as the decomposition step in train, and reconstruct does not apply.

Value

MDSConfig object.

Author(s)

EDG

Examples

```
mds_config <- setup_MDS(k = 2L, dist_method = "manhattan")
mds_config
```

---

setup_MiraiExecution *Set up a* MiraiExecutionConfig

Description

Parallel execution through **mirai**. Takes no plan: **mirai** has no equivalent of a `future_plan`, which is why it is not an argument here.

Usage

```
setup_MiraiExecution(
  n_workers = NULL,
  n_workers_outer = NULL,
  n_workers_tuning = NULL,
  n_workers_algorithm = NULL,
  device = NULL,
  on_error = c("continue", "stop", "stop_outer"),
  seed = NULL,
  shared_memory = c("auto", "none", "always"),
  warm_workers = TRUE
)
```

Arguments

`n_workers`	Integer [1, Inf): Number of parallel workers. Left `NULL`, it follows the named dispatch levels, or `default_n_workers()` if none is named.
`n_workers_outer`	Optional Integer [1, Inf): Workers for outer resampling, overriding the automatic assignment.
`n_workers_tuning`	Optional Integer [1, Inf): Workers for tuning, overriding the automatic assignment.

`n_workers_algorithm`
: Optional Integer [1, Inf): Threads for a self-parallelizing algorithm, overriding the automatic assignment.

`device`
: Optional `DeviceConfig`: Compute device, as a type name – `"cpu"`, `"cuda"`, `"mps"` (the Apple silicon GPU) or `"opencl"` (a LightGBM GPU build) – or setup_CUDA to name which GPUs. `NULL` selects one per algorithm: `"cuda"` where the machine has it and the algorithm can use it, else the CPU. An algorithm that cannot use the requested device runs on the CPU, and the run's resources line says so; a device the machine lacks is an error when an algorithm tries to use it. `"mps"` is never selected automatically: in rtemis benchmarks on an Apple M5 the CPU was faster for every network narrower than about 1024 units or trained in batches smaller than about 2048 cases, often by 2 to 5 times, and `"mps"` was faster by up to about 25% only with both and thousands of features. It also slows down across many fits in one R session, and a seed does not reach dropout on it.

`on_error`
: Character {"continue", "stop", "stop_outer"}: Failure policy. `"continue"` makes grid cells and unscorable hyperparameter combinations non-fatal (recorded, warned, and excluded), failing only when nothing is scorable or the final model fails; `"stop"` aborts on any error; `"stop_outer"` tolerates grid-cell failures but aborts on an outer-fold failure.

`seed`
: Optional Integer [0, Inf): Master seed for the run's computation RNG, from which one independent substream per parallel task is derived. Left `NULL`, a seed is drawn from the current RNG stream and recorded on the returned object, so every run is reproducible and the seed it used is auditable. Distinct from a `ResamplerConfig` seed, which governs how the data is split.

`shared_memory`
: Character {"none", "auto", "always"}: Whether to hand workers the training data through OS shared memory instead of serializing a copy to each. `"auto"` is the default: it shares whenever it can – workers parallel and on this machine, **mori** installed – and quietly does not when it cannot, falling back to the ordinary transport. `"none"` disables it. `"always"` is a demand: it shares even when the run is sequential, so a run can be compared against its own shared counterpart, and raises an error when the request cannot be met, for a caller relying on sharing to stay inside a memory budget.

`warm_workers`
: Logical: Load **rtemis** in every worker as the pool is built, rather than leaving each worker to load it on its first task. Either way the result is the same; TRUE records the loading time in the run's worker-pool setup step.

Details

See setup_FutureExecution for how the three worker levels are assigned and how the run's seed is resolved; both are identical under either backend.

Value

`MiraiExecutionConfig` object.

Author(s)

EDG

Examples

```
setup_MiraiExecution(n_workers = 2L)
```

`setup_MLP` *Setup MLP Hyperparameters*

Description

Setup hyperparameters for MLP (Multilayer Perceptron) training.

Usage

```
setup_MLP(
  hidden_units = NULL,
  shape = NULL,
  shape_layers = NULL,
  shape_max_units = NULL,
  activation = "relu",
  norm = NULL,
  residual = FALSE,
  dropout = 0,
  input_dropout = 0,
  weight_decay = 0,
  l1_penalty = 0,
  embedding_dim = NULL,
  embedding_dropout = 0,
  optimizer = "adamw",
  lr = 0.001,
  batch_size = 256L,
  max_epochs = 100L,
  max_grad_norm = NULL,
  ifw = FALSE,
  norm_first = FALSE,
  bias = TRUE,
  embeddings = TRUE,
  beta1 = NULL,
  beta2 = NULL,
  eps = NULL,
  momentum = NULL,
  lr_scheduler = NULL,
  patience = 10L,
  loss = NULL,
  seed = NULL,
  num_workers = 0L,
  drop_last = FALSE
)
```

Arguments

hidden_units	(Tunable) Optional Integer [1, Inf) vector: Units in each hidden layer, one value per layer. NULL generates the widths from the shape settings.
shape	(Tunable) Optional Character {"funnel", "constant", "triangle", "long_funnel", "diamond", "hexagon", "stairs"}: Profile of the generated hidden layer widths. Ignored when hidden_units is set. NULL uses "funnel" where hidden_units is NULL.
shape_layers	(Tunable) Optional Integer [1, Inf): Number of hidden layers to generate. Ignored when hidden_units is set. NULL uses 3 where hidden_units is NULL.
shape_max_units	(Tunable) Optional Integer [1, Inf): Widest generated hidden layer. Ignored when hidden_units is set. NULL derives it from the encoded input width where hidden_units is NULL.
activation	(Tunable) Character {"relu", "gelu", "silu", "elu", "selu", "leaky_relu", "tanh"}: Activation applied after every hidden layer.
norm	(Tunable) Optional Character {"batch_norm", "layer_norm"}: Normalization applied in every hidden layer. NULL applies none.
residual	(Tunable) Logical: If TRUE, add a residual connection around every hidden layer, projected when the layer changes width.
dropout	(Tunable) Numeric [0, 1): Dropout probability applied after every hidden layer.
input_dropout	(Tunable) Numeric [0, 1): Dropout probability applied to the encoded input.
weight_decay	(Tunable) Numeric [0, Inf): L2 penalty, decoupled from the gradient under the adamw optimizer.
l1_penalty	(Tunable) Numeric [0, Inf): L1 penalty on the linear weights, added to the loss.
embedding_dim	(Tunable) Optional Integer [1, Inf): Width of every embedding. NULL sizes each from its feature's cardinality.
embedding_dropout	(Tunable) Numeric [0, 1): Dropout probability applied to the concatenated embeddings.
optimizer	(Tunable) Character {"adamw", "adam", "sgd", "rmsprop"}: Optimization algorithm.
lr	(Tunable) Numeric (0, Inf): Learning rate.
batch_size	(Tunable) Integer [1, Inf): Cases per optimization step.
max_epochs	(Tunable) Integer [1, Inf): Largest number of passes over the training set.
max_grad_norm	(Tunable) Optional Numeric (0, Inf): Clip the gradient norm to this value before each step. NULL does not clip.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.
norm_first	Logical: If TRUE, normalize before the activation rather than after it.
bias	Logical: If TRUE, give every hidden layer and the output layer a bias term.
embeddings	Logical: If TRUE, represent each categorical feature by a learned embedding; if FALSE, one-hot encode them.

beta1	Optional Numeric [0, 1): Exponential decay rate of the first moment estimate. Applies to the adam and adamw optimizers.
beta2	Optional Numeric [0, 1): Exponential decay rate of the second moment estimate. Applies to the adam and adamw optimizers.
eps	Optional Numeric (0, Inf): Term added to the denominator for numerical stability. Applies to the adam, adamw and rmsprop optimizers.
momentum	Optional Numeric [0, Inf): Momentum factor. Applies to the sgd and rmsprop optimizers.
lr_scheduler	Optional Character {"step", "cosine_annealing", "one_cycle", "reduce_on_plateau"}: Learning-rate schedule. NULL holds the learning rate fixed.
patience	Integer [1, Inf): Epochs without validation improvement before stopping early.
loss	Optional Character {"mse", "l1", "smooth_l1", "cross_entropy"}: Training objective. NULL sets it from the outcome type.
seed	Optional Integer: Random seed for weight initialization, dropout and batch shuffling.
num_workers	Integer [0, Inf): Subprocesses used to load batches.
drop_last	Logical: If TRUE, drop the last incomplete batch of each training epoch.

Details

A fully connected feedforward network built and trained with `torch`, for regression, binary and multiclass classification.

Architecture. Give the hidden layers directly with `hidden_units` – `c(256L, 128L, 64L)` is three layers of those widths – or leave it NULL and let `shape`, `shape_layers` and `shape_max_units` generate them. Setting both is an error. The generated profiles are `funnel` (a linear taper from the widest layer down to a third of it), `constant`, `triangle` (a linear rise from the input width), `long_funnel`, `diamond`, `hexagon` and `stairs`; the vocabulary is Talos's, by way of AutoPyTorch. `shape_max_units` defaults to four times the encoded input width, clamped to [64, 512] and never below that width.

The published tabular deep-learning benchmarks tune constant-width MLPs; `shape = "constant"` reproduces that architecture. The default is `funnel`.

Tuning. `hidden_units` is tunable like any other hyperparameter, with one architecture per candidate: `setup_MLP(hidden_units = tune_over(c(64L, 32L), c(128L, 64L, 32L)))`. A single bare vector there is one architecture, not a set of candidates, and is rejected as such.

Categorical features are represented by learned embeddings, each sized `min(600, round(1.6 * cardinality^0.56))` unless `embedding_dim` fixes them all. `embeddings = FALSE` one-hot encodes them. Numeric features are always centered and scaled, and the fitted encoder is re-applied at predict time.

Device and reproducibility. The device is the execution config's: see the `device` argument of setup_FutureExecution. Unset picks `cuda` where available and `cpu` otherwise, and train names the one it resolved. `"mps"`, the Apple silicon GPU, runs only when named. In rtemis benchmarks on an Apple M5 the CPU was faster for every network narrower than about 1024 units or trained in batches smaller than about 2048 cases, often by 2 to 5 times; with both, and thousands of features, `"mps"` was faster by up to about 25%. It also slows down across many fits in one R session. Try

it for a wide network on wide data. A seed governs weight initialization and batch shuffling on mps but **not dropout**, so a seeded mps fit reproduces exactly until a dropout rate is non-zero. That combination warns.

Early stopping needs a validation set: pass dat_validation to train, and the fit keeps the weights of the best validation epoch rather than the last. Without one it runs the full max_epochs and patience has no effect.

l1_penalty **is not** weight_decay**.** weight_decay is the L2 term the torch optimizer applies, decoupled from the gradient under adamw; l1_penalty has no torch equivalent and is accumulated over the linear weights and added to the loss.

Batch normalization and dropout can degrade training when combined; both are off by default.

The scheduler takes no settings of its own – each configures itself from the run's budget: step decays by 0.1 every max_epochs / 3 epochs, cosine_annealing anneals over max_epochs, one_cycle peaks at lr over the run's real step count, and reduce_on_plateau decays by 0.1 after half the early-stopping patience.

get_varimp() returns NULL: a torch MLP has no native importance measure.

Value

MLPHyperparameters object.

Author(s)

EDG

Examples

```
mlp_hyperparams <- setup_MLP(hidden_units = c(64L, 32L), max_epochs = 20L)
mlp_hyperparams
```

---

setup_ModalityStacking

Setup ModalityStacking Hyperparameters

Description

Setup hyperparameters for modality stacking: a SuperLearner whose base learners each see one group of features.

Usage

```
setup_ModalityStacking(
  feature_groups = NULL,
  base_learners = list(setup_GLM(), setup_GLMNET(), setup_Ranger()),
  meta_learner = setup_NNLS(),
  inner_resampling_config = setup_KFold(n_resamples = 10L),
  discrete = FALSE,
```

```
  expand_search_spaces = TRUE,
  ifw = FALSE
)
```

Arguments

feature_groups	Optional List: Features each base learner sees, as a named list of character vectors keyed by base learner name. Required by train.
base_learners	List of Hyperparameters objects, or one to use for every group: The library, named to match feature_groups.
meta_learner	Hyperparameters object: Learner fitted on the base learners' cross-validated predictions.
inner_resampling_config	ResamplerConfig object: Cross-validation scheme used to build the level-one predictions.
discrete	(Tunable) Logical: If TRUE, keep the single lowest-risk library entry rather than the weighted combination.
expand_search_spaces	Logical: If TRUE, expand a base learner's search space into one library entry per combination.
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

With a wide x formed by concatenating modalities – imaging, genomics, labs – and few cases, one model per modality can outperform one model over all features, and each modality can take the algorithm that suits it: SPLS for correlated blocks, LASSO for sparse ones, gradient boosting for interactions. Training is otherwise identical to setup_SuperLearner: cross-validated predictions from each base learner, combined by the meta learner.

feature_groups maps each base learner's name to the features it sees. Pass a single Hyperparameters object as base_learners to use the same algorithm on every group.

Two properties affect whether this outperforms one model over the concatenated modalities. Each base learner sees only its own group, so what every *other* modality contributes to the outcome is irreducible noise to it, which can hide a real signal in a weak modality behind a strong one. And the default meta learner returns a convex combination, which averages the modalities rather than adding them: where they are complementary rather than competing, meta_learner = setup_NNLS(normalize = FALSE) lets their contributions sum.

feature_groups is checked against the training data at train time, not at setup: it names columns, and there is no data to name them in yet.

Supports regression and binary classification.

Value

ModalityStackingHyperparameters object.

Author(s)

EDG

Examples

```
modality_hyperparams <- setup_ModalityStacking(
  feature_groups = list(a = c("Sepal.Length", "Sepal.Width"), b = "Petal.Length"),
  base_learners = list(a = setup_GLM(), b = setup_CART())
)
modality_hyperparams
```

---

setup_MonotonicHAL *Setup MonotonicHAL Hyperparameters*

Description

Setup hyperparameters for monotonic Highly Adaptive Lasso training.

Usage

```
setup_MonotonicHAL(
  smoothness_orders = 1L,
  reduce_basis = NULL,
  num_knots = NULL,
  penalized = TRUE,
  use_min = TRUE,
  nfolds = 10L,
  seed = NULL,
  ifw = FALSE
)
```

Arguments

smoothness_orders	(Tunable) Integer [0, 1]: Smoothness of the basis functions: 0 fits zero-order indicators and yields a step function, 1 fits piecewise linear splines and yields a continuous one.
reduce_basis	(Tunable) Optional Numeric (0, 1]: Minimum proportion of cases a basis function must be non-zero in to be kept. Applies only when `smoothness_orders` is 0; a search that also covers higher orders drops it from those grid cells.
num_knots	Optional Integer [1, Inf): Number of knots spanning each feature. NULL generates them from smoothness_orders.
penalized	Logical: If TRUE, apply the lasso penalty to the basis functions; if FALSE, remove it.
use_min	Logical: If TRUE, select `lambda.min` from the internal cross-validation; if FALSE, the more heavily penalized `lambda.1se`.

`nfolds`	Integer [3, Inf): Largest number of folds of the internal cross-validation that selects lambda.
`seed`	Optional Integer: Random seed for the internal cross-validation's fold assignment. NULL leaves it drawn from the ambient RNG.
`ifw`	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

A Highly Adaptive Lasso restricted to additive, monotonic non-decreasing fits. hal9001::fit_hal is given a formula that constrains every basis function's coefficient to be non-negative, and the interaction degree is fixed at 1. Regression uses the gaussian family, binary classification the binomial one; `hal9001` has no multinomial family, so multiclass classification is not supported.

The intended use is probability calibration, where the single feature is a classifier's score. A non-decreasing calibration map preserves the ranking of scores, and so AUC. Pass it to calibrate to use it instead of the default calibrator, setup_Isotonic.

Relative to `Isotonic`, which is the other monotonic calibrator, this fits on the logit scale, so it does not saturate at 0 and 1 as the boundary bins of isotonic regression do; it can round to an endpoint in double precision when the input scores are extreme. At `smoothness_orders = 1` it is strictly increasing rather than a step function, so it introduces no ties and leaves AUC unchanged.

`smoothness_orders` 0 fits indicator basis functions and yields a step function; 1 fits piecewise linear splines and yields a continuous map.

`penalized = FALSE` removes the lasso penalty, giving the non-parametric maximum likelihood estimate over the monotonic class. Combined with `smoothness_orders = 0` that is isotonic regression, up to the logit-scale parameterization.

`lambda` is selected by cross-validation inside the fit and is not a search dimension. `seed` fixes that cross-validation's fold assignment; `nfolds` and `use_min` control it. `nfolds` is an upper bound: training uses at most as many folds as the data supports at three cases each, and reports the number at `verbosity >= 1`.

`get_varimp()` reports the same two measures as `setup_HAL`, `importance` and `max_coefficient`.

Value

MonotonicHALHyperparameters object.

Author(s)

EDG

Examples

```
monotonichal_hyperparams <- setup_MonotonicHAL(smoothness_orders = 0L)
monotonichal_hyperparams
```

setup_MultiASWCriterion *Setup MultiASWCriterionConfig*

Description

Setup the subsampled average silhouette width criterion for setup_PAMK: the silhouette width is computed on random subsets of the cases and averaged, which makes the score affordable on many cases.

Usage

```
setup_MultiASWCriterion(n_subsets = NULL)
```

Arguments

n_subsets	Optional Integer [1, Inf): Number of random subsets the average silhouette width is computed on.

Details

n_subsets maps to fpc::pamk()'s ns.

Value

MultiASWCriterionConfig object.

Author(s)

EDG

Examples

```
setup_MultiASWCriterion(n_subsets = 20L)
```

setup_NeuralGas *Setup NeuralGasConfig*

Description

Setup NeuralGasConfig

Usage

```
setup_NeuralGas(k = 3L, dist = "euclidean", features = NULL)
```

Arguments

k	Integer [1, Inf): Number of clusters.
dist	Character {"euclidean", "manhattan"}: Distance measure to use.
features	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. NULL clusters on all numeric features.

Value

NeuralGasConfig object.

Author(s)

EDG

Examples

```
neuralgas_config <- setup_NeuralGas(k = 4L, dist = "euclidean")
neuralgas_config
```

---

setup_NMF *Setup NMF config.*

Description

Setup NMF config.

Usage

```
setup_NMF(k = 2L, method = "brunet", nrun = 1L, features = NULL)
```

Arguments

k	Integer [1, Inf): Number of components.
method	Character: NMF method. See NMF::nmf.
nrun	Integer [1, Inf): Number of runs to perform.
features	Optional Character vector: Names of at least 2 distinct feature columns to decompose. NULL decomposes all numeric features.

Value

NMFConfig object.

Author(s)

EDG

Examples

```
nmf_config <- setup_NMF(k = 3L)
nmf_config
```

`setup_NNLS` *Setup NNLS Hyperparameters*

Description

Setup hyperparameters for non-negative least squares.

Usage

```
setup_NNLS(normalize = TRUE, ifw = FALSE)
```

Arguments

`normalize`	(Tunable) Logical: If TRUE, scale the coefficients to sum to 1.
`ifw`	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

NNLS fits `y ~ Xb` subject to `b >= 0`, with no intercept. It is the default meta learner of the stacked meta learners (setup_SuperLearner, setup_ModalityStacking), where the predictors are the base learners' cross-validated predictions and a non-negative, sum-to-one coefficient vector is the ensemble weighting. With no intercept and a sign constraint, it fits only outcomes in the cone the predictors span.

With `normalize = TRUE` the coefficients are scaled to sum to 1, so the fit is a convex combination of the predictors. For classification the outcome is coded 0/1 on the second factor level and the fitted values are read as probabilities of that level; unnormalized coefficients do not guarantee a value in [0, 1], so predictions are clamped.

Get more information from `nnls::nnls`.

Value

NNLSHyperparameters object.

Author(s)

EDG

Examples

```
nnls_hyperparams <- setup_NNLS(normalize = FALSE)
nnls_hyperparams
```

setup_Nystrom *Setup NystromConfig*

Description

Setup the Nystrom approximation for setup_SpectralRBF or setup_SpectralLaplace. Passing one turns the approximation on; its `sample` says how many cases it samples.

Usage

```
setup_Nystrom(sample = NULL)
```

Arguments

`sample`	Optional Integer [1, Inf): Number of cases the approximation samples. Must be fewer than the number of cases. Unset samples one sixth of them.

Details

`sample` maps to `kernlab::specc()`'s `nystrom.sample`; passing the config at all sets `nystrom.red = TRUE`.

Value

`NystromConfig` object.

Author(s)

EDG

Examples

```
setup_Nystrom(sample = 200L)
```

setup_PAM *Setup PAMConfig*

Description

Setup a `PAMConfig` object for Partitioning Around Medoids (PAM) clustering, via the 'cluster' package.

Usage

```
setup_PAM(
  k = 3L,
  dist = "euclidean",
  stand = FALSE,
  do_swap = TRUE,
  variant = "original",
  nstart = NULL,
  features = NULL
)
```

Arguments

k	Integer [1, Inf): Number of clusters. Must be fewer than the number of cases.
dist	Character {"euclidean", "manhattan"}: Distance measure to use.
stand	Logical: If TRUE, standardize each variable before computing dissimilarities.
do_swap	Logical: If TRUE, perform the swap phase.
variant	Character {"original", "o_1", "o_2", "f_3", "f_4", "f_5", "faster"}: Algorithm variant; "original" is the published algorithm, the others trade exactness for speed.
nstart	Optional Integer [1, Inf): Number of random starts. NULL uses the deterministic build phase instead of drawing the initial medoids at random.
features	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. NULL clusters on all numeric features.

Details

PAM is the medoid-based counterpart of k-means: each cluster is represented by one of the cases, which makes it usable with non-Euclidean dissimilarities and less sensitive to outliers.

Argument names are rtemis' own. They map to `cluster::pam()` as: `dist` -> `metric`, `do_swap` -> `do.swap`; the rest keep their names.

Value

`PAMConfig` object.

Author(s)

EDG

References

Kaufman L, Rousseeuw PJ (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley. doi:10.1002/9780470316801

Examples

```
pam_config <- setup_PAM(k = 3L)
pam_config
```

`setup_PAMK` *Setup PAMKConfig*

Description

Setup a `PAMKConfig` object for Partitioning Around Medoids with estimation of the number of clusters (PAMK), via the 'fpc' package.

Usage

```
setup_PAMK(
  krange = 2:10,
  criterion = setup_ASWCriterion(),
  use_pam = TRUE,
  scaling = FALSE,
  alpha = 0.001,
  features = NULL
)
```

Arguments

`krange`	Integer [1, Inf) vector: Candidate numbers of clusters to compare. Must include at least one value greater than 1.
`criterion`	`PAMKCriterionConfig`: Criterion used to choose the number of clusters, from setup_ASWCriterion, setup_CHCriterion or setup_MultiASWCriterion.
`use_pam`	Logical: If TRUE, fit each candidate with PAM; if FALSE, with CLARA.
`scaling`	Logical: If TRUE, scale each variable by its root mean square after centering.
`alpha`	Numeric [0, 1]: Significance level of the Duda-Hart test, used only when 1 is among the candidates.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Details

PAMK fits PAM (or CLARA) for every candidate in `krange` and keeps the one scoring best under `criterion`. There is therefore no `k` to set, and the number of clusters chosen is reported by the resulting `Clustering`.

Argument names are rtemis' own. They map to `fpc::pamk()` as: `use_pam` -> `usepam`; the criterion's `type` -> `criterion` and its `n_subsets` -> `ns`; the rest keep their names.

`fpc::pamk()` also accepts a per-variable numeric vector for `scaling`; only the logical form is exposed here. Scale by column with preprocess instead, where the decision is recorded and replayed with the data.

Value

PAMKConfig object.

Author(s)

EDG

References

Kaufman L, Rousseeuw PJ (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley. doi:10.1002/9780470316801

Examples

```
pamk_config <- setup_PAMK(krange = 2:5)
pamk_config
setup_PAMK(krange = 2:5, criterion = setup_MultiASWCriterion(n_subsets = 20L))
```

---

setup_ParquetIngest *Set up a* ParquetIngestConfig

Description

Creates the settings ingest() uses to read a Parquet file. Parquet stores its column types, and the only format-specific setting is the reader library.

Usage

```
setup_ParquetIngest(
  columns = NULL,
  character2factor = TRUE,
  clean_colnames = TRUE,
  remove_duplicates = FALSE,
  reader = "arrow"
)
```

Arguments

columns	Optional Named vector: Declared column types, keyed by column name, in the profile/r/v1 vocabulary – c(Age = "integer"). A column named here is converted to the declared type after reading, replacing the type the reader assigned.
character2factor	Logical: If TRUE, read character columns as factors.
clean_colnames	Logical: If TRUE, normalize column names.
remove_duplicates	Logical: If TRUE, drop rows repeating an earlier row.
reader	Character {"arrow", "nanoparquet"}: Library used to read the file.

Value

ParquetIngestConfig object.

Author(s)

EDG

Examples

```
setup_ParquetIngest()
```

`setup_PCA` *Setup PCA config.*

Description

Setup PCA config.

Usage

```
setup_PCA(k = 3L, center = TRUE, scale = TRUE, tol = NULL, features = NULL)
```

Arguments

k	Integer [1, Inf): Number of components. (passed to prcomp rank.)
center	Logical: If TRUE, center the data.
scale	Logical: If TRUE, scale the data.
tol	Optional Numeric [0, Inf): Tolerance.
features	Optional Character vector: Names of at least 2 distinct feature columns to decompose. NULL decomposes all numeric features.

Value

PCAConfig object.

Author(s)

EDG

Examples

```
pca_config <- setup_PCA(k = 3L)
pca_config
```

`setup_PCoA` *Setup PCoA config.*

Description

Principal coordinates analysis, also called classical or Torgerson multidimensional scaling: an eigendecomposition of the doubly centered squared dissimilarities between cases, with `stats::cmdscale()`. With `dist_method = "euclidean"` the components equal PCA scores of the centered, unscaled data, up to sign; other dissimilarities are what PCoA is for.

Usage

```
setup_PCoA(k = 2L, dist_method = "euclidean", features = NULL)
```

Arguments

`k`	Integer [1, Inf): Number of components. (passed to `cmdscale` `k`)
`dist_method`	Character {"euclidean", "manhattan", "chord", "bray", "canberra", "jaccard", "kulczynski", "hellinger"}: Dissimilarity between cases, computed with `vegan::vegdist()`. "bray", "canberra", "jaccard", "kulczynski" and "hellinger" require non-negative data.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to decompose. `NULL` decomposes all numeric features.

Details

PCoA applies to new data with Gower's (1968) out-of-sample formula, from the dissimilarities between each new case and the training cases, so it can be used as the decomposition step in train. Applying a fit to its own training data reproduces the fitted components. Because those dissimilarities are needed, the fit keeps the training feature matrix, and its size grows with the number of training cases.

Dissimilarities other than "euclidean", "chord" and "hellinger" generally give some negative eigenvalues; only dimensions with positive eigenvalues are returned, and asking for more is an error that states how many there are. reconstruct does not apply.

Value

PCoAConfig object.

Author(s)

EDG

Examples

```
pcoa_config <- setup_PCoA(k = 2L, dist_method = "bray")
pcoa_config
```

setup_PredefinedPartition *Set up a* PredefinedPartitionConfig

Description

A split already recorded in the data itself.

Usage

```
setup_PredefinedPartition(
  column = NULL,
  training_value = "train",
  test_value = "test"
)
```

Arguments

column	Optional Character: Name of the column holding each case's partition label.
training_value	Character: Value in column identifying training cases.
test_value	Character: Value in column identifying held-out test cases.

Value

PredefinedPartitionConfig object.

Author(s)

EDG

Examples

```
setup_PredefinedPartition(column = "split")
```

setup_Preprocessor *Setup Preprocessor*

Description

Creates a PreprocessorConfig object, which can be used in preprocess.

Usage

```
setup_Preprocessor(
  complete_cases = FALSE,
  remove_features_thres = NULL,
  remove_cases_thres = NULL,
  missingness = FALSE,
  impute = FALSE,
  impute_type = c("missRanger", "micePMM", "meanMode"),
  impute_missRanger_params = list(pmm.k = 3L, maxiter = 10L, num.trees = 500L),
  impute_numeric = c("mean", "median"),
  impute_categorical = "mode",
  impute_numeric_values = NULL,
  impute_level_values = NULL,
  integer2factor = FALSE,
  integer2numeric = FALSE,
  logical2factor = FALSE,
  logical2numeric = FALSE,
  numeric2factor = FALSE,
  numeric2factor_levels = NULL,
  numeric_cut_n = 0L,
  numeric_cut_labels = FALSE,
  numeric_quant_n = 0L,
  numeric_quant_NAonly = FALSE,
  unique_len2factor = 0L,
  character2factor = FALSE,
  factorNA2missing = FALSE,
  factorNA2missing_level = "missing",
  factor2integer = FALSE,
  factor2integer_startat0 = TRUE,
  factor2integer_levels = NULL,
  scale = FALSE,
  center = scale,
  scale_centers = NULL,
  scale_coefficients = NULL,
  remove_constants = FALSE,
  remove_constants_skip_missing = TRUE,
  remove_features = NULL,
  remove_duplicates = FALSE,
  one_hot = FALSE,
  one_hot_levels = NULL,
  add_date_features = FALSE,
  date_features = c("weekday", "month", "year"),
  add_holidays = FALSE,
  holidays = c("USLaborDay", "NewYearsDay", "ChristmasDay"),
  exclude = NULL
)
```

Arguments

`complete_cases` Logical: If TRUE, only retain complete cases (no missing data). Not available inside train: a fitted preprocessor is replayed at predict time, where dropping rows would return fewer predictions than rows.

`remove_features_thres`
Optional Numeric (0, 1]: Remove features with missing values in >= to this fraction of cases. 1 removes only fully-missing features. Not available inside train: each resample would learn its own set, so the folds would train on different features. Name the columns with `remove_features` instead, or drop them beforehand with preprocess.

`remove_cases_thres`
Optional Numeric (0, 1]: Remove cases with >= to this fraction of missing features. 1 removes only fully-missing cases. Not available inside train: a fitted preprocessor is replayed at predict time, where dropping rows would return fewer predictions than rows.

`missingness` Logical: If TRUE, generate new boolean columns for each feature with missing values, indicating which cases were missing data.

`impute` Logical: If TRUE, impute missing cases. See `impute_type`.

`impute_type` Character {"missRanger", "micePMM", "meanMode"}: Imputation method. "missRanger" fits one random forest per feature with missing values, kept in the `Preprocessor` and applied to new data by apply_preprocessor. "meanMode" imputes the values that `impute_numeric` and `impute_categorical` compute on the training data. "micePMM" imputes by predictive mean matching with 'mice', which imputes only the data it is run on: a `Preprocessor` using it cannot be applied to new data. See the Imputation section.

`impute_missRanger_params`
List: Parameters passed to `missRanger::missRanger`, e.g. "pmm.k", "maxiter", "num.trees". `pmm.k` greater than 0 results in predictive mean matching. Reduce `num.trees` for faster imputation especially in large datasets. Set `pmm.k = 0` to disable predictive mean matching.

`impute_numeric` Character {"mean", "median"}: Statistic imputed into numeric and logical features for `impute_type = "meanMode"`, and into such features with no missing training values for `impute_type = "missRanger"`. Integer and `bit64::integer64` features take the statistic rounded half to even, so they remain integer. Logical features take the statistic of their 0/1 values rounded half to even: the more frequent value, FALSE on a tie. Integer features that code categories belong in factors: see `integer2factor`.

`impute_categorical`
Character {"mode"}: Statistic imputed into factor and character features for `impute_type = "meanMode"`, and into such features with no missing training values for `impute_type = "missRanger"`. "mode" is the most frequent value. A tie goes to the first tied level of a factor and to the first tied value in Unicode code point order of a character feature.

`impute_numeric_values`
Optional Named vector: Value imputed into numeric or logical features, keyed by feature name; a logical feature takes 1 as TRUE and 0 as FALSE. Covers

every such feature for impute_type = "meanMode" and features with no missing training values for impute_type = "missRanger". Used in place of the value impute_numeric computes. A value for an integer feature must be a whole number, and a value for a logical feature 0 or 1. preprocess stores the values it uses in the Preprocessor, and apply_preprocessor imputes new data with them.

impute_level_values
: Optional Named vector: Value imputed into factor or character features, keyed by feature name. Covers every such feature for impute_type = "meanMode" and features with no missing training values for impute_type = "missRanger". Used in place of the value impute_categorical computes. preprocess stores the values it uses in the Preprocessor, and apply_preprocessor imputes new data with them.

integer2factor
: Logical: If TRUE, convert all integers to factors. This includes bit64::integer64 columns.

integer2numeric
: Logical: If TRUE, convert all integers to numeric (will only work if integer2factor = FALSE). This includes bit64::integer64 columns.

logical2factor
: Logical: If TRUE, convert all logical variables to factors.

logical2numeric
: Logical: If TRUE, convert all logical variables to numeric.

numeric2factor
: Logical: If TRUE, convert all numeric variables to factors.

numeric2factor_levels
: Optional Character vector: Will be passed to levels arg of factor() if numeric2factor = TRUE. For advanced/ specific use cases; need to know unique values of numeric vector(s) and given all numeric vars have same unique values.

numeric_cut_n
: Integer [0, Inf): If > 0, convert all numeric variables to factors by binning using base::cut with breaks equal to this number.

numeric_cut_labels
: Logical: The labels argument of base::cut.

numeric_quant_n
: Integer [0, Inf): If > 0, convert all numeric variables to factors by binning using base::cut with breaks equal to this number of quantiles, produced using stats::quantile. 0 disables; 1 is a single break, which bounds no bin, and is rejected.

numeric_quant_NAonly
: Logical: If TRUE, only bin numeric variables with missing values.

unique_len2factor
: Integer [0, Inf): Convert all variables with less than or equal to this number of unique values to factors. 0 and 1 both disable it: a feature with one unique value is a constant, which remove_constants covers. For example, unique_len2factor = 2 converts binary variables encoded as 1, 2 to factors.

character2factor
: Logical: If TRUE, convert all character variables to factors.

factorNA2missing
: Logical: If TRUE, make NA values in factors be of level factorNA2missing_level. This step runs before imputation, so one preprocess call can code missing values in categorical variables as a level and impute numeric variables.

`factorNA2missing_level`
: Character: Name of level if `factorNA2missing = TRUE`.

`factor2integer`
: Logical: If TRUE, convert all factors to integers.

`factor2integer_startat0`
: Logical: If TRUE, start integer coding at 0.

`factor2integer_levels`
: Optional List: Named list of the form "feature_name" = "levels". Used to code factors identically to the training data when applying `factor2integer` to validation or test data using `Preprocessor`. Values whose level is absent take the single index above the known levels.

`scale`
: Logical: If TRUE, scale columns of `x`.

`center`
: Logical: If TRUE, center columns of `x`. If unset, follows `scale`.

`scale_centers`
: Optional Named vector: Centering values for each feature.

`scale_coefficients`
: Optional Named vector: Scaling values for each feature.

`remove_constants`
: Logical: If TRUE, remove constant columns.

`remove_constants_skip_missing`
: Logical: If TRUE, skip missing values, before checking if feature is constant.

`remove_features`
: Optional Character vector: Features to remove.

`remove_duplicates`
: Logical: If TRUE, remove duplicate cases. Not available inside train: a fitted preprocessor is replayed at predict time, where dropping rows would return fewer predictions than rows.

`one_hot`
: Logical: If TRUE, convert all factors using one-hot encoding.

`one_hot_levels`
: Optional List: Named list of the form "feature_name" = "levels". Used when applying one-hot encoding to validation or test data using `Preprocessor`.

`add_date_features`
: Logical: If TRUE, extract date features from date columns.

`date_features`
: Character {"weekday", "month", "year"} vector: Features to extract from dates.

`add_holidays`
: Logical: If TRUE, extract holidays from date columns.

`holidays`
: Character vector: Holidays to flag when `add_holidays` is TRUE, named as timeDate holiday functions, e.g. "USLaborDay", "ChristmasDay". See `timeDate::listHolidays()` for the valid names. Note that unprefixed names are not US holidays: "LaborDay" is May 1.

`exclude`
: Optional Integer vector: Exclude these columns from preprocessing.

Value

`PreprocessorConfig` object.

Imputation

"meanMode" stores one value per feature, computed on the data `preprocess()` is given, and apply_preprocessor imputes new data with those values. Each value has its feature's type: an integer feature receives an integer, a logical feature a logical, and a factor one of its levels.

"missRanger" fits one random forest per feature with missing values and keeps the forests, and a copy of the data `predict()` draws starting values from, in the `Preprocessor`, which therefore grows with the data. The forests are an R object: a config written to JSON carries the method and its parameters, not the forests. A feature with no missing values in the data has no forest and takes its `impute_numeric` or `impute_categorical` value when it is missing in new data. `predict()` stops on a factor level of an imputed feature that is absent from the data the forests were fitted on.

"micePMM" imputes the data it is run on and fits no model for new data; apply_preprocessor refuses a `Preprocessor` that uses it.

For preprocessing inside train, see the Imputation section of setup_SupervisedPreprocessor.

Order of Operations

Steps run in this order, and the order is part of what each one does: `remove_constants` runs before the quantile cut because a constant has no quantiles, `factorNA2missing` before `impute` so that missing values in a categorical feature become a level, and `one_hot` last among the encoders so that it sees every factor the earlier steps created.

Feature *creation* precedes feature *transformation*: `add_date_features` and `add_holidays` run ahead of the conversions, so the weekday, month and holiday factors they derive are encoded and the year is scaled exactly as any column of the input would be. They run after the case and feature filters, which judge the data as it was given.

- keep complete cases only
- set aside excluded columns
- remove named features
- remove constants
- remove duplicates
- remove cases by missingness threshold
- remove features by missingness threshold
- add date features
- add holidays
- integer to factor
- logical to factor
- numeric to factor
- character to factor
- features with <= N unique values to factor
- integer to numeric
- logical to numeric
- cut numeric to n bins

- cut numeric to n quantiles
- factor NA to named level
- factor to integer
- add missingness column
- impute
- scale and/or center
- one-hot encoding
- re-add excluded columns, after the preprocessed ones

Author(s)

EDG

Examples

```
preproc_config <- setup_Preprocessor(factorNA2missing = TRUE)
preproc_config
```

`setup_RadialSVM` *Setup RadialSVM Hyperparameters*

Description

Setup hyperparameters for RadialSVM training.

Usage

```
setup_RadialSVM(cost = 1, gamma = 0.01, ifw = FALSE)
```

Arguments

`cost`	(Tunable) Numeric (0, Inf): Cost of constraints violation.
`gamma`	(Tunable) Numeric (0, Inf): Kernel coefficient.
`ifw`	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Get more information from e1071::svm.

Value

RadialSVMHyperparameters object.

Author(s)

EDG

Examples

```
radial_svm_hyperparams <- setup_RadialSVM(cost = 10, gamma = 0.1, ifw = TRUE)
radial_svm_hyperparams
```

---

setup_RandomPartition *Set up a* RandomPartitionConfig

Description

A uniformly random split. Row order and grouping play no part in the assignment.

Usage

```
setup_RandomPartition(train_p = 0.75, seed = NULL)
```

Arguments

train_p	Numeric (0, 1): Training set fraction.
seed	Optional Integer [0, Inf): Random seed.

Value

RandomPartitionConfig object.

Author(s)

EDG

Examples

```
setup_RandomPartition(train_p = 0.8, seed = 2026L)
```

---

setup_Ranger *Setup Ranger Hyperparameters*

Description

Setup hyperparameters for Ranger Random Forest training.

Usage

```
setup_Ranger(
  num_trees = 500L,
  mtry = NULL,
  importance = "impurity",
  write_forest = TRUE,
  probability = FALSE,
  min_node_size = NULL,
  min_bucket = NULL,
  max_depth = NULL,
  replace = TRUE,
  sample_fraction = ifelse(replace, 1, 0.632),
  class_weights = NULL,
  splitrule = NULL,
  num_random_splits = 1L,
  alpha = 0.5,
  minprop = 0.1,
  poisson_tau = 1,
  split_select_weights = NULL,
  always_split_variables = NULL,
  respect_unordered_factors = NULL,
  scale_permutation_importance = FALSE,
  local_importance = FALSE,
  regularization_factor = 1,
  regularization_usedepth = FALSE,
  keep_inbag = FALSE,
  inbag = NULL,
  holdout = FALSE,
  quantreg = FALSE,
  time_interest = NULL,
  oob_error = TRUE,
  save_memory = FALSE,
  verbose = TRUE,
  node_stats = FALSE,
  seed = NULL,
  na_action = "na.learn",
  ifw = FALSE
)
```

Arguments

num_trees	(Tunable) Integer [1, Inf): Number of trees.
mtry	(Tunable) Optional Integer [1, Inf): Number of features to consider at each split.
importance	Character {"none", "impurity", "impurity_corrected", "permutation"}: Variable importance mode. "impurity" is the Gini index for classification, the response variance for regression.
write_forest	Logical: If TRUE, save the forest object (required for prediction). Set to FALSE to reduce memory if no prediction is intended.

`probability` Logical: If TRUE, grow a probability forest. Classification only.

`min_node_size` (Tunable) Optional Integer [1, Inf): Minimal node size. If NULL, ranger uses 1 for classification, 5 for regression, 3 for survival, and 10 for probability.

`min_bucket` Optional Integer [1, Inf): Fewest cases a terminal node may hold; no split creates a smaller node. Unset uses ranger's default of 1.

`max_depth` (Tunable) Optional Integer [0, Inf): Maximal tree depth. NULL or 0 means unlimited depth, 1 means tree stumps.

`replace` (Tunable) Logical: If TRUE, sample with replacement.

`sample_fraction`
(Tunable) Numeric (0, 1]: Fraction of observations to sample. Default is 1 with replacement and 0.632 without.

`class_weights` Optional Numeric [0, Inf) vector: Per-class weights for classification. Length equal to the number of classes, named by class label. NULL weights every class equally.

`splitrule` (Tunable) Optional Character: Splitting rule. Classification: "gini", "extratrees", "hellinger"; regression: "variance", "extratrees", "maxstat", "beta"; survival: "logrank", "extratrees", "C", "maxstat".

`num_random_splits`
(Tunable) Integer [1, Inf): Number of random splits per candidate variable, for the "extratrees" splitrule.

`alpha` (Tunable) Numeric [0, 1]: Significance threshold to allow splitting, for the "maxstat" splitrule.

`minprop` (Tunable) Numeric [0, 1]: Lower quantile of the covariate distribution considered for splitting, for the "maxstat" splitrule.

`poisson_tau` Numeric (0, Inf): Tau parameter, for the "poisson" regression splitrule.

`split_select_weights`
Optional List: Per-feature probabilities of being selected for splitting, in [0, 1]. One vector applied to every tree, or a list of length `num_trees` with one vector per tree. NULL gives every feature the same probability.

`always_split_variables`
Optional Character vector: Names of variables to always include as split candidates, in addition to the `mtry` variables. NULL adds none.

`respect_unordered_factors`
Optional Character {"partition", "ignore", "order"}: Handling of unordered factors. "partition" considers all 2-partitions, "ignore" orders levels by first occurrence, "order" orders levels by mean response.

`scale_permutation_importance`
Logical: If TRUE, scale permutation importance by its standard error. Permutation importance only.

`local_importance`
Logical: If TRUE, compute local (per-observation) permutation importance.

`regularization_factor`
(Tunable) Numeric [0, Inf): Regularization factor penalizing variables with many split points. Requires `splitrule = "variance"`.

`regularization_usedepth`
: Logical: If TRUE, apply the regularization factor with node depth. Requires `regularization_factor`.

`keep_inbag`
: Logical: If TRUE, record how often each observation is in-bag per tree.

`inbag`
: Optional List: Manually set in-bag counts; a list of length `num_trees`, each a per-case count vector. Can be used for stratified sampling. NULL draws each tree's sample by `replace` and `sample_fraction`, with the training case weights as sampling weights where train supplies them.

`holdout`
: Logical: If TRUE, use hold-out mode: hold out samples with case weight 0 and use them for variable importance and prediction error.

`quantreg`
: Logical: If TRUE, prepare quantile prediction (quantile regression forests). Regression only; set `keep_inbag = TRUE` for out-of-bag quantile prediction.

`time_interest`
: Optional Numeric vector: Time points of interest for survival prediction. Survival only. NULL uses the observed event times.

`oob_error`
: Logical: If TRUE, compute the OOB prediction error. Set to FALSE to save time if only the forest is needed.

`save_memory`
: Logical: If TRUE, use the memory-saving (slower) splitting mode.

`verbose`
: Logical: If TRUE, show computation status and estimated runtime.

`node_stats`
: Logical: If TRUE, save additional node statistics (terminal nodes only).

`seed`
: Optional Integer: Random seed. If NULL, the seed is generated from R. Set to 0 to ignore the R seed.

`na_action`
: Character {"na.learn", "na.omit", "na.fail"}: How to handle missing values. "na.learn" uses observations with missing values in splitting, treating missing as a separate category.

`ifw`
: (Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Get more information from ranger::ranger.

Value

RangerHyperparameters object.

Author(s)

EDG

Examples

```
ranger_hyperparams <- setup_Ranger(num_trees = 1000L, ifw = FALSE)
ranger_hyperparams
```

setup_RDSIngest *Set up an* RDSIngestConfig

Description

Creates the settings ingest() uses to read an RDS file. RDS stores R types, and the format has no settings of its own.

Usage

```
setup_RDSIngest(
  columns = NULL,
  character2factor = TRUE,
  clean_colnames = TRUE,
  remove_duplicates = FALSE
)
```

Arguments

columns	Optional Named vector: Declared column types, keyed by column name, in the profile/r/v1 vocabulary – c(Age = "integer"). A column named here is converted to the declared type after reading, replacing the type the reader assigned.
character2factor	Logical: If TRUE, read character columns as factors.
clean_colnames	Logical: If TRUE, normalize column names.
remove_duplicates	Logical: If TRUE, drop rows repeating an earlier row.

Value

RDSIngestConfig object.

Author(s)

EDG

Examples

```
setup_RDSIngest()
```

setup_Resampler *Setup a Resampler (Deprecated)*

Description

setup_Resampler() is deprecated. Use setup_KFold(), setup_StratSub(), setup_StratBoot(), setup_Bootstrap(), setup_LOOCV(), or setup_Custom(). Only arguments relevant to the selected type are forwarded. Omitted arguments retain the selected setup function's default resolution.

Usage

```
setup_Resampler(
  n_resamples = 10L,
  type = c("KFold", "StratSub", "StratBoot", "Bootstrap", "LOOCV", "Custom"),
  stratify_var = NULL,
  train_p = 0.75,
  strat_n_bins = 4L,
  target_length = NULL,
  id_strat = NULL,
  seed = NULL,
  verbosity = 1L,
  resamples = NULL
)
```

Arguments

n_resamples	Integer [1, Inf): Number of resamples to make.
type	Character: Resampling method; matched without regard to case.
stratify_var	Optional Character: Variable to stratify by.
train_p	Numeric (0, 1): Training set percentage.
strat_n_bins	Integer [1, Inf): Number of bins to stratify by.
target_length	Optional Integer [1, Inf): Target length for stratified bootstraps.
id_strat	Optional Character: Name of the column holding per-case grouping IDs, e.g. subject IDs when the dataset contains repeated measurements. Cases sharing an ID stay in the same resample, so a case can only be present in the training or the test set, not both. Requires that resample() be given the data frame the column lives in. train takes the column as a case identifier, so it groups the resamples by it and excludes it from the features.
seed	Optional Integer [0, Inf): Random seed.
verbosity	Integer: Console verbosity. The deprecation warning is always signaled and can be handled with suppressWarnings().
resamples	Optional List: Resamples to use, one integer vector of training-case indices per resample.

Details

`id_strat` names a column in the input data; per-case ID vectors are not accepted. Arguments that do not apply to the selected type are ignored.

Value

A resampler configuration of the selected type.

Examples

```
config <- suppressWarnings(setup_Resampler(5L, "KFold", seed = 2026L))
config
```

---

setup_SerialExecution *Set up a* SerialExecutionConfig

Description

Execution in the calling process: nothing is dispatched to a worker. Takes no `n_workers` and no `future_plan`, neither of which a serial run can act on.

Usage

```
setup_SerialExecution(
  n_workers_algorithm = NULL,
  device = NULL,
  on_error = c("continue", "stop", "stop_outer"),
  seed = NULL,
  shared_memory = c("auto", "none", "always")
)
```

Arguments

`n_workers_algorithm`
: Optional Integer [1, Inf): Threads for a self-parallelizing algorithm, overriding the automatic assignment.

`device`
: Optional `DeviceConfig`: Compute device, as a type name – `"cpu"`, `"cuda"`, `"mps"` (the Apple silicon GPU) or `"opencl"` (a LightGBM GPU build) – or setup_CUDA to name which GPUs. `NULL` selects one per algorithm: `"cuda"` where the machine has it and the algorithm can use it, else the CPU. An algorithm that cannot use the requested device runs on the CPU, and the run's resources line says so; a device the machine lacks is an error when an algorithm tries to use it. `"mps"` is never selected automatically: in rtemis benchmarks on an Apple M5 the CPU was faster for every network narrower than about 1024 units or trained in batches smaller than about 2048 cases, often by 2 to 5 times, and `"mps"` was faster by up to about 25% only with both and thousands of features. It also slows down across many fits in one R session, and a seed does not reach dropout on it.

on_error	Character {"continue", "stop", "stop_outer"}: Failure policy. "continue" makes grid cells and unscorable hyperparameter combinations non-fatal (recorded, warned, and excluded), failing only when nothing is scorable or the final model fails; "stop" aborts on any error; "stop_outer" tolerates grid-cell failures but aborts on an outer-fold failure.
seed	Optional Integer [0, Inf): Master seed for the run's computation RNG, from which one independent substream per parallel task is derived. Left NULL, a seed is drawn from the current RNG stream and recorded on the returned object, so every run is reproducible and the seed it used is auditable. Distinct from a ResamplerConfig seed, which governs how the data is split.
shared_memory	Character {"none", "auto", "always"}: Whether to hand workers the training data through OS shared memory instead of serializing a copy to each. "auto" is the default: it shares whenever it can – workers parallel and on this machine, **mori** installed – and quietly does not when it cannot, falling back to the ordinary transport. "none" disables it. "always" is a demand: it shares even when the run is sequential, so a run can be compared against its own shared counterpart, and raises an error when the request cannot be met, for a caller relying on sharing to stay inside a memory budget.

Details

n_workers_algorithm still applies: it is threads *within the calling process*, not worker processes, so a self-parallelizing algorithm runs multi-threaded under a serial config.

See setup_FutureExecution for how the run's seed is resolved.

Value

SerialExecutionConfig object.

Author(s)

EDG

Examples

```
setup_SerialExecution()
```

---

setup_SHAP *Setup SHAP Explanation*

Description

Configure per-case Shapley additive explanations for explain().

Usage

```
setup_SHAP(
  estimator = "auto",
  perturbation = NULL,
  scale = NULL,
  background_n = NULL,
  n_coalitions = NULL,
  approach = NULL,
  seed = NULL
)
```

Arguments

estimator	Character {"auto", "exact", "kernel"}: Which estimator to use.
perturbation	Optional Character {"interventional", "conditional"}: Value function. NULL uses the resolved estimator's documented default.
scale	Optional Character {"margin", "probability"}: Scale the contributions are additive on. NULL sets it from the outcome type.
background_n	Optional Integer [1, Inf): Cases to subsample from the background data. NULL uses all of it.
n_coalitions	Optional Integer [2, Inf): Most feature subsets the kernel estimator evaluates. NULL lets the backend decide.
approach	Optional Character {"empirical", "gaussian", "copula", "ctree", "vaeac", "categorical", "timeseries"}: Conditional-distribution estimator used by the kernel estimator.
seed	Optional Integer [0, Inf): Random seed.

Details

Three choices here define what is computed, and each is recorded on the returned explanation, so explanations computed differently are told apart.

`perturbation` **is the question being asked.** `"interventional"` breaks the dependence between features, attributing to what the model *uses*: a feature the model ignores gets zero even if it predicts the outcome perfectly. `"conditional"` respects the joint distribution, attributing to what the case's features carry information about, so a correlated proxy for a used feature receives credit. The first describes the model's computation; the second describes the case.

`scale` **decides whether the contributions add up.** They are additive on the scale they are computed on. For regression that is the outcome and there is nothing to choose. For classification the model's additive scale is the margin, and probability is a nonlinear transform of it, so contributions on the probability scale do **not** sum to the predicted probability.

`estimator` **defaults to** `"auto"`**,** which uses the exact estimator wherever the algorithm has one and the model-agnostic kernel estimator otherwise; `explanation_traits()` reports which applies where.

`approach` and `n_coalitions` apply only to the kernel estimator, and `approach` only when `perturbation` is `"conditional"` – the interventional answer is not one of several conditional estimators, it is the absence of conditioning.

Setting either therefore requires `estimator = "kernel"` rather than `"auto"`, even for an algorithm `"auto"` would send to the kernel estimator anyway. Tuning an estimator is a decision to use it, and the alternative – accepting settings that a resolved exact estimator would then ignore – would drop them silently.

Value

`SHAPConfig` object.

Author(s)

EDG

Examples

```
setup_SHAP()
setup_SHAP(perturbation = "conditional")
setup_SHAP(estimator = "kernel", perturbation = "conditional", approach = "ctree")
```

---

setup_SpectralLaplace *Setup SpectralLaplaceConfig*

Description

Setup a `SpectralLaplaceConfig` object for spectral clustering with an exponential (Laplacian) kernel, via the 'kernlab' package. The exponential kernel decays more slowly than the Gaussian of setup_SpectralRBF and so keeps more weight on distant pairs.

Usage

```
setup_SpectralLaplace(
  k = 3L,
  sigma = NULL,
  iterations = 200L,
  nystrom = NULL,
  features = NULL
)
```

Arguments

`k`	Integer [2, Inf): Number of clusters. Must be no more than the number of cases.
`sigma`	Optional Numeric (0, Inf): Inverse kernel width. Unset uses 1.
`iterations`	Integer [1, Inf): Maximum number of k-means iterations run on the spectral embedding.
`nystrom`	Optional `NystromConfig`: Approximate the affinity matrix from a sample of the cases, from setup_Nystrom. `NULL` decomposes it in full.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Details

Argument names are rtemis' own. They map to `kernlab::specc()` as: `k` -> `centers`, `sigma` -> `kpar = list(sigma)` with `kernel = "laplacedot"`, `nystrom` -> `nystrom.red` and `nystrom.sample`; `iterations` keeps its name.

Value

`SpectralLaplaceConfig` object.

Author(s)

EDG

References

Ng AY, Jordan MI, Weiss Y (2001). On Spectral Clustering: Analysis and an Algorithm. *Advances in Neural Information Processing Systems*, 14, 849-856.

Examples

```
setup_SpectralLaplace(k = 3L, sigma = 0.5)
```

---

setup_SpectralLocal *Setup SpectralLocalConfig*

Description

Setup a `SpectralLocalConfig` object for spectral clustering with a locally scaled Gaussian kernel, via the 'kernlab' package. Each case's kernel width is taken from the distance to its seventh nearest neighbor, which lets one clustering hold groups of differing density. The per-case widths need every pairwise distance, so this variant has no Nystrom approximation.

Usage

```
setup_SpectralLocal(k = 3L, iterations = 200L, features = NULL)
```

Arguments

`k`	Integer [2, Inf): Number of clusters. Must be no more than the number of cases.
`iterations`	Integer [1, Inf): Maximum number of k-means iterations run on the spectral embedding.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Details

Argument names are rtemis' own. They map to `kernlab::specc()` as: `k` -> `centers`, with `kpar = "local"`; `iterations` keeps its name.

Value

`SpectralLocalConfig` object.

Author(s)

EDG

References

Zelnik-Manor L, Perona P (2004). Self-Tuning Spectral Clustering. *Advances in Neural Information Processing Systems*, 17, 1601-1608.

Examples

```
setup_SpectralLocal(k = 3L)
```

`setup_SpectralRBF` *Setup SpectralRBFConfig*

Description

Setup a `SpectralRBFConfig` object for spectral clustering with a Gaussian (radial basis function) kernel, via the 'kernlab' package.

Usage

```
setup_SpectralRBF(
  k = 3L,
  sigma = NULL,
  sigma_sample_fraction = NULL,
  iterations = 200L,
  nystrom = NULL,
  features = NULL
)
```

Arguments

`k`	Integer [2, Inf): Number of clusters. Must be no more than the number of cases.
`sigma`	Optional Numeric (0, Inf): Inverse kernel width. Unset estimates it from the data.
`sigma_sample_fraction`	Optional Numeric (0, 1]: Fraction of the cases the kernel width is estimated from when `sigma` is unset. Unset uses three quarters of them.
`iterations`	Integer [1, Inf): Maximum number of k-means iterations run on the spectral embedding.
`nystrom`	Optional `NystromConfig`: Approximate the affinity matrix from a sample of the cases, from setup_Nystrom. `NULL` decomposes it in full.

`features` Optional Character vector: Names of at least 2 distinct feature columns to cluster on. `NULL` clusters on all numeric features.

Details

Spectral clustering builds a similarity graph over the cases, embeds them in the leading eigenvectors of that graph's normalized Laplacian, and runs k-means there. Groups that are connected but not compact – concentric rings, elongated bands – separate in that embedding while k-means on the raw features cannot find them. This variant measures similarity with a Gaussian kernel whose width is shared by every case; see setup_SpectralLocal for a per-case width and setup_SpectralLaplace for an exponential kernel.

The cost is cubic in the number of cases, since the whole affinity matrix is decomposed. Pass `nystrom` to approximate the decomposition from a sample instead.

Argument names are rtemis' own. They map to `kernlab::specc()` as: `k` -> `centers`, `sigma` -> `kpar = list(sigma)` with `kernel = "rbfdot"`, `sigma_sample_fraction` -> `mod.sample` (with `kpar = "automatic"` when `sigma` is unset), `nystrom` -> `nystrom.red` and `nystrom.sample`; `iterations` keeps its name.

Value

`SpectralRBFConfig` object.

Author(s)

EDG

References

Ng AY, Jordan MI, Weiss Y (2001). On Spectral Clustering: Analysis and an Algorithm. *Advances in Neural Information Processing Systems*, 14, 849-856.

Examples

```
spectral_config <- setup_SpectralRBF(k = 3L)
spectral_config
setup_SpectralRBF(k = 3L, nystrom = setup_Nystrom(sample = 100L))
```

---

setup_SplitConformal *Setup Split Conformal Prediction*

Description

Configure split conformal prediction for conformal.

Usage

```
setup_SplitConformal(alpha = 0.1, score = NULL, seed = NULL)
```

Arguments

alpha	Numeric (0, 1): Miscoverage rate. The region covers with probability at least 1 - alpha.
score	Optional Character {"absolute", "LAC", "APS"}: Nonconformity score. NULL resolves to "absolute" for a regression and "APS" for a classification.
seed	Optional Integer [0, Inf): Random seed for the APS draw. NULL draws one and records it.

Details

The reference construction: score every calibration case, take q to be the score at rank ceiling((n + 1) * (1 - alpha)) from the bottom, and return every outcome whose score would not exceed it. Coverage is at least 1 - alpha, and needs only that the calibration cases and the test case be exchangeable.

alpha **has a floor set by the calibration count.** A finite region needs ceiling((n + 1) * (1 - alpha)) <= n, so alpha = 0.05 needs at least 19 calibration cases and alpha = 0.01 at least 99. conformal() aborts naming the count.

score **is a choice for a classification only.** A regression has one score, abs(y - yhat), which is what NULL resolves to.

- "APS", the default, accumulates the predicted probabilities from the most probable class down to the true one. It produces larger sets, with better coverage of the cases where the model is uncertain.
- "LAC" scores 1 - p(y). It produces the smallest sets attaining valid *marginal* coverage, and under-covers the cases where the model is uncertain.

APS is randomized (Romano, Sesia and Candes, 2020): the true label's own probability enters its score multiplied by a uniform draw, one per case. Without it, every case whose true label ranks last scores exactly 1, so a model erring on more than alpha of its cases puts the threshold at 1 and every set holds every label – valid coverage carrying no information. seed keeps the draw reproducible: left NULL, one is drawn and recorded on the returned config, so an unseeded region can still be reproduced from what it reports. "LAC" draws nothing and ignores it.

Value

SplitConformalConfig object.

Author(s)

EDG

Examples

```
setup_SplitConformal()
setup_SplitConformal(alpha = 0.05)
setup_SplitConformal(score = "LAC")
```

setup_SPLS *Setup SPLS Hyperparameters*

Description

Setup hyperparameters for Sparse Partial Least Squares training.

Usage

```
setup_SPLS(
  k = 2L,
  eta = 0.5,
  kappa = 0.5,
  select = "pls2",
  fit = "simpls",
  classifier = "lda",
  scale_x = TRUE,
  scale_y = FALSE,
  eps = 1e-04,
  maxstep = 100L,
  ifw = FALSE
)
```

Arguments

k	(Tunable) Integer [1, Inf): Number of latent components.
eta	(Tunable) Numeric [0, 1): Sparsity threshold. Higher values select fewer features.
kappa	(Tunable) Numeric [0, 0.5]: Concavity of the surrogate direction vector problem.
select	Character {"pls2", "simpls"}: Feature selection algorithm (regression only).
fit	Character {"kernelpls", "widekernelpls", "simpls", "oscorespls"}: PLS algorithm used for model fitting (regression only).
classifier	Character {"lda", "logistic"}: Classifier fit on the latent components (classification only).
scale_x	Logical: If TRUE, scale features to unit variance.
scale_y	Logical: If TRUE, scale the outcome to unit variance (regression only).
eps	Numeric (0, Inf): Convergence tolerance (regression only).
maxstep	Integer [1, Inf): Maximum number of iterations per component (regression only).
ifw	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Regression is fit with spls::spls and classification with spls::splsda, chosen from the outcome type. Parameters marked "regression only" or "classification only" are passed to the backend that accepts them and ignored by the other.

`spls` provides no case weights, so `ifw` cannot be honored: enabling it makes training abort.

Value

SPLSHyperparameters object.

Author(s)

EDG

Examples

```
spls_hyperparams <- setup_SPLS(k = 3L, eta = 0.7)
spls_hyperparams
```

`setup_StratBoot` *Setup StratBoot Resampler*

Description

Setup StratBoot Resampler

Usage

```
setup_StratBoot(
  n_resamples = 10L,
  stratify_var = NULL,
  train_p = 0.75,
  strat_n_bins = 4L,
  target_length = NULL,
  id_strat = NULL,
  seed = NULL
)
```

Arguments

`n_resamples`	Integer [1, Inf): Number of resamples to make.
`stratify_var`	Optional Character: Variable to stratify by.
`train_p`	Numeric (0, 1): Training set percentage.
`strat_n_bins`	Integer [1, Inf): Number of bins to stratify by.
`target_length`	Optional Integer [1, Inf): Target length for stratified bootstraps.

id_strat	Optional Character: Name of the column holding per-case grouping IDs, e.g. subject IDs when the dataset contains repeated measurements. Cases sharing an ID stay in the same resample, so a case can only be present in the training or the test set, not both. Requires that resample() be given the data frame the column lives in. train takes the column as a case identifier, so it groups the resamples by it and excludes it from the features.
seed	Optional Integer [0, Inf): Random seed.

Value

StratBootConfig object.

Author(s)

EDG

Examples

```
strat_boot_resampler <- setup_StratBoot(n_resamples = 10L, seed = 2026L)
strat_boot_resampler
```

---

setup_StratSub *Setup StratSub Resampler*

Description

Setup StratSub Resampler

Usage

```
setup_StratSub(
  n_resamples = 10L,
  train_p = 0.75,
  stratify_var = NULL,
  strat_n_bins = 4L,
  id_strat = NULL,
  seed = NULL
)
```

Arguments

n_resamples	Integer [1, Inf): Number of resamples to make.
train_p	Numeric (0, 1): Training set percentage.
stratify_var	Optional Character: Variable to stratify by.
strat_n_bins	Integer [1, Inf): Number of bins to stratify by.

id_strat	Optional Character: Name of the column holding per-case grouping IDs, e.g. subject IDs when the dataset contains repeated measurements. Cases sharing an ID stay in the same resample, so a case can only be present in the training or the test set, not both. Requires that `resample()` be given the data frame the column lives in. train takes the column as a case identifier, so it groups the resamples by it and excludes it from the features.
seed	Optional Integer [0, Inf): Random seed.

Value

`StratSubConfig` object.

Author(s)

EDG

Examples

```
strat_sub_resampler <- setup_StratSub(n_resamples = 10L, seed = 2026L)
strat_sub_resampler
```

`setup_SuperConfig` *Setup SuperConfig*

Description

Setup `SuperConfig` object. `SuperConfig` is a portable, data-agnostic recipe: `dat_training_path` is optional, so the same config can be validated, shared, or described without data and have a path bound later (e.g. by the `rtemis` CLI) before train.

Usage

```
setup_SuperConfig(
  dat_training_path = NULL,
  dat_validation_path = NULL,
  dat_test_path = NULL,
  character2factor = TRUE,
  outcome = NULL,
  features = NULL,
  weights = NULL,
  positive_class = NULL,
  preprocessor_config = NULL,
  decomposition_config = NULL,
  hyperparameters = NULL,
  tuner_config = NULL,
  outer_resampling_config = NULL,
  execution_config = setup_FutureExecution(),
```

```
  question = NULL,
  outdir = "results/",
  verbosity = 1L
)
```

Arguments

dat_training_path	Character or NULL: Path to training data file. NULL leaves the recipe unbound; set it (or supply data) before train.
dat_validation_path	Optional Character: Path to validation data file.
dat_test_path	Optional Character: Path to test data file.
character2factor	Logical: If TRUE, read character columns as factors. A delimited file carries no type information, so this is what decides whether a column of labels is a categorical predictor or an unusable string one. Applied to all three datasets.
outcome	Optional Character: Name of the outcome column; NULL for rtemis's convention, the last column.
features	Optional Character vector: Names of the predictor columns; NULL for every column except the outcome.
weights	Optional Character: Column name in training data to use as observation weights. If NULL, no weights are used.
positive_class	Character or NULL: For binary classification, the outcome level to treat as positive. NULL keeps the existing factor level order.
preprocessor_config	Optional SupervisedPreprocessorConfig: Configuration for data preprocessing.
decomposition_config	Optional DecompositionConfig: Configuration for data decomposition.
hyperparameters	Optional Hyperparameters or HyperparametersSet: One model configuration or a named set of configurations searched as a union.
tuner_config	Optional TunerConfig: Configuration for hyperparameter tuning.
outer_resampling_config	Optional ResamplerConfig: Configuration for outer res resampling during model training.
execution_config	ExecutionConfig object: Configuration for execution settings. Setup with setup_SerialExecution, setup_FutureExecution or setup_MiraiExecution.
question	Optional Character: Question to answer with the supervised learning analysis.
outdir	Optional Character: Output directory for results; NULL to write nothing to disk.
verbosity	Integer [0, Inf): Verbosity level.

Value

SuperConfigPaths object.

Author(s)

EDG

Examples

```
## Not run:
# requires local data file
sc <- setup_SuperConfig(
  dat_training_path = "train.csv",
  preprocessor_config = setup_SupervisedPreprocessor(scale = TRUE),
  hyperparameters = setup_LightRF(),
  tuner_config = setup_GridSearch(),
  outer_resampling_config = setup_KFold(),
  execution_config = setup_FutureExecution(),
  question = "Can we tell iris species apart given their measurements?",
  outdir = "models/"
)

## End(Not run)
```

---

setup_SuperConfigLive *Setup SuperConfigLive*

Description

Build a SuperConfigTabular – same shape as setup_SuperConfig but with in-memory tabular data instead of file paths.

Usage

```
setup_SuperConfigLive(
  dat_training,
  dat_validation = NULL,
  dat_test = NULL,
  outcome = NULL,
  features = NULL,
  weights = NULL,
  positive_class = NULL,
  preprocessor_config = NULL,
  decomposition_config = NULL,
  hyperparameters = NULL,
  tuner_config = NULL,
  outer_resampling_config = NULL,
  execution_config = setup_FutureExecution(),
```

```
  question = NULL,
  outdir = NULL,
  verbosity = 1L
)
```

Arguments

dat_training	data.frame or data.table. Training data.
dat_validation	data.frame, data.table, or NULL.
dat_test	data.frame, data.table, or NULL.
outcome	Optional Character: Name of the outcome column; NULL for rtemis's convention, the last column.
features	Optional Character vector: Names of the predictor columns; NULL for every column except the outcome.
weights	Optional Character: Column name in dat_training used as observation weights.
positive_class	Optional Character: For binary classification, the outcome level to treat as positive; forwarded to train which reorders the outcome factor via set_positive_class. NULL keeps the existing level order.
preprocessor_config, hyperparameters, tuner_config, outer_resampling_config, execution_config, question, verbosity	See setup_SuperConfig.
decomposition_config	Optional DecompositionConfig: Configuration for data decomposition.
outdir	Optional Character: Output directory; NULL means "do not write to disk" (the rtemislive case).

Value

SuperConfigTabular object.

Author(s)

EDG

Examples

```
scl <- setup_SuperConfigLive(
  dat_training = iris,
  hyperparameters = setup_LightGBM(),
  outer_resampling_config = setup_KFold(),
  question = "Can we tell iris species apart given their measurements?"
)
```

`setup_SuperLearner` *Setup SuperLearner Hyperparameters*

Description

Setup hyperparameters for a SuperLearner: the cross-validated stacked ensemble of van der Laan, Polley & Hubbard (2007).

Usage

```
setup_SuperLearner(
  base_learners = list(setup_GLM(), setup_GLMNET(), setup_Ranger()),
  meta_learner = setup_NNLS(),
  inner_resampling_config = setup_KFold(n_resamples = 10L),
  discrete = FALSE,
  expand_search_spaces = TRUE,
  ifw = FALSE
)
```

Arguments

`base_learners`	List of `Hyperparameters` objects: The library. Names label the level-one predictions and the reported weights; unnamed entries are named after their algorithm.
`meta_learner`	`Hyperparameters` object: Learner fitted on the base learners' cross-validated predictions. The default is non-negative least squares normalized to sum to 1, i.e. a convex combination.
`inner_resampling_config`	`ResamplerConfig` object: Cross-validation scheme used to build the level-one predictions.
`discrete`	(Tunable) Logical: If TRUE, keep the single lowest-risk library entry rather than the weighted combination (the discrete SuperLearner).
`expand_search_spaces`	Logical: If TRUE, expand a base learner's search space into one library entry per combination.
`ifw`	(Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Details

Each base learner is fitted on every training fold of `inner_resampling_config` and predicts that fold's held-out cases, producing one cross-validated prediction per case per learner. The meta learner is fitted on those predictions against the outcome, and its coefficients are the ensemble weights. Every base learner is then refitted on the whole training set, so a prediction is the meta learner applied to the base learners' predictions.

Search spaces become library entries. A base learner holding more than one value for a tunable hyperparameter is expanded into one library entry per combination, and the ensemble weights

choose between them – the ensemble's own cross-validation is the model selection, so no inner tuning is needed. Set `expand_search_spaces = FALSE` to tune each such learner by inner resampling within every fold instead, which is far more expensive. A learner that tunes *itself* (GLMNET choosing `lambda` by `cv.glmnet`) is unaffected either way.

The cross-validated predictions and the resampler are kept on the fitted model, which is what a cross-fitting estimator needs from it.

Supports regression and binary classification.

Value

SuperLearnerHyperparameters object.

Author(s)

EDG

Examples

```
superlearner_hyperparams <- setup_SuperLearner(
  base_learners = list(setup_GLM(), setup_CART())
)
superlearner_hyperparams
```

---

setup_SupervisedPreprocessor

Setup Supervised Preprocessor

Description

Creates a `SupervisedPreprocessorConfig` object: the preprocessing a train run can fit, which is setup_Preprocessor without the four operations a fitted preprocessor cannot carry.

Usage

```
setup_SupervisedPreprocessor(
  missingness = FALSE,
  impute = FALSE,
  impute_type = c("meanMode", "missRanger"),
  impute_missRanger_params = list(pmm.k = 3L, maxiter = 10L, num.trees = 500L),
  impute_numeric = c("mean", "median"),
  impute_categorical = "mode",
  impute_numeric_values = NULL,
  impute_level_values = NULL,
  integer2factor = FALSE,
  integer2numeric = FALSE,
  logical2factor = FALSE,
  logical2numeric = FALSE,
```

```
  numeric2factor = FALSE,
  numeric2factor_levels = NULL,
  numeric_cut_n = 0L,
  numeric_cut_labels = FALSE,
  numeric_quant_n = 0L,
  numeric_quant_NAonly = FALSE,
  unique_len2factor = 0L,
  character2factor = FALSE,
  factorNA2missing = FALSE,
  factorNA2missing_level = "missing",
  factor2integer = FALSE,
  factor2integer_startat0 = TRUE,
  factor2integer_levels = NULL,
  scale = FALSE,
  center = scale,
  scale_centers = NULL,
  scale_coefficients = NULL,
  remove_constants = FALSE,
  remove_constants_skip_missing = TRUE,
  remove_features = NULL,
  one_hot = FALSE,
  one_hot_levels = NULL,
  add_date_features = FALSE,
  date_features = c("weekday", "month", "year"),
  add_holidays = FALSE,
  holidays = c("USLaborDay", "NewYearsDay", "ChristmasDay"),
  exclude = NULL
)
```

Arguments

`missingness`	Logical: If TRUE, generate new boolean columns for each feature with missing values, indicating which cases were missing data.
`impute`	Logical: If TRUE, impute missing cases. See `impute_type`.
`impute_type`	Character {"meanMode", "missRanger"}: Imputation method. "meanMode" imputes the values that `impute_numeric` and `impute_categorical` compute on the training data. "missRanger" fits one random forest per feature with missing values, applied to validation, test and new data with `predict()`; it multiplies the memory and computation of a train run. See the Imputation section.
`impute_missRanger_params`	List: Parameters passed to `missRanger::missRanger`, e.g. "pmm.k", "maxiter", "num.trees". `pmm.k` greater than 0 results in predictive mean matching. Reduce `num.trees` for faster imputation especially in large datasets. Set `pmm.k = 0` to disable predictive mean matching.
`impute_numeric`	Character {"mean", "median"}: Statistic imputed into numeric and logical features for `impute_type = "meanMode"`, and into such features with no missing training values for `impute_type = "missRanger"`. Integer and `bit64::integer64` features take the statistic rounded half to even, so they remain integer. Logical

features take the statistic of their 0/1 values rounded half to even: the more frequent value, FALSE on a tie. Integer features that code categories belong in factors: see `integer2factor`.

`impute_categorical`
: Character {"mode"}: Statistic imputed into factor and character features for `impute_type = "meanMode"`, and into such features with no missing training values for `impute_type = "missRanger"`. "mode" is the most frequent value. A tie goes to the first tied level of a factor and to the first tied value in Unicode code point order of a character feature.

`impute_numeric_values`
: Optional Named vector: Value imputed into numeric or logical features, keyed by feature name; a logical feature takes 1 as TRUE and 0 as FALSE. Covers every such feature for `impute_type = "meanMode"` and features with no missing training values for `impute_type = "missRanger"`. Used in place of the value `impute_numeric` computes. A value for an integer feature must be a whole number, and a value for a logical feature 0 or 1. preprocess stores the values it uses in the `Preprocessor`, and apply_preprocessor imputes new data with them.

`impute_level_values`
: Optional Named vector: Value imputed into factor or character features, keyed by feature name. Covers every such feature for `impute_type = "meanMode"` and features with no missing training values for `impute_type = "missRanger"`. Used in place of the value `impute_categorical` computes. preprocess stores the values it uses in the `Preprocessor`, and apply_preprocessor imputes new data with them.

`integer2factor` Logical: If TRUE, convert all integers to factors. This includes `bit64::integer64` columns.

`integer2numeric`
: Logical: If TRUE, convert all integers to numeric (will only work if `integer2factor = FALSE`). This includes `bit64::integer64` columns.

`logical2factor` Logical: If TRUE, convert all logical variables to factors.

`logical2numeric`
: Logical: If TRUE, convert all logical variables to numeric.

`numeric2factor` Logical: If TRUE, convert all numeric variables to factors.

`numeric2factor_levels`
: Optional Character vector: Will be passed to `levels` arg of `factor()` if `numeric2factor = TRUE`. For advanced/ specific use cases; need to know unique values of numeric vector(s) and given all numeric vars have same unique values.

`numeric_cut_n` Integer [0, Inf): If > 0, convert all numeric variables to factors by binning using `base::cut` with `breaks` equal to this number.

`numeric_cut_labels`
: Logical: The `labels` argument of base::cut.

`numeric_quant_n`
: Integer [0, Inf): If > 0, convert all numeric variables to factors by binning using `base::cut` with `breaks` equal to this number of quantiles, produced using `stats::quantile`. 0 disables; 1 is a single break, which bounds no bin, and is rejected.

numeric_quant_NAonly
: Logical: If TRUE, only bin numeric variables with missing values.

unique_len2factor
: Integer [0, Inf): Convert all variables with less than or equal to this number of unique values to factors. 0 and 1 both disable it: a feature with one unique value is a constant, which remove_constants covers. For example, unique_len2factor = 2 converts binary variables encoded as 1, 2 to factors.

character2factor
: Logical: If TRUE, convert all character variables to factors.

factorNA2missing
: Logical: If TRUE, make NA values in factors be of level factorNA2missing_level. This step runs before imputation, so one preprocess call can code missing values in categorical variables as a level and impute numeric variables.

factorNA2missing_level
: Character: Name of level if factorNA2missing = TRUE.

factor2integer
: Logical: If TRUE, convert all factors to integers.

factor2integer_startat0
: Logical: If TRUE, start integer coding at 0.

factor2integer_levels
: Optional List: Named list of the form "feature_name" = "levels". Used to code factors identically to the training data when applying factor2integer to validation or test data using Preprocessor. Values whose level is absent take the single index above the known levels.

scale
: Logical: If TRUE, scale columns of x.

center
: Logical: If TRUE, center columns of x. If unset, follows scale.

scale_centers
: Optional Named vector: Centering values for each feature.

scale_coefficients
: Optional Named vector: Scaling values for each feature.

remove_constants
: Logical: If TRUE, remove constant columns.

remove_constants_skip_missing
: Logical: If TRUE, skip missing values, before checking if feature is constant.

remove_features
: Optional Character vector: Features to remove.

one_hot
: Logical: If TRUE, convert all factors using one-hot encoding.

one_hot_levels
: Optional List: Named list of the form "feature_name" = "levels". Used when applying one-hot encoding to validation or test data using Preprocessor.

add_date_features
: Logical: If TRUE, extract date features from date columns.

date_features
: Character {"weekday", "month", "year"} vector: Features to extract from dates.

add_holidays
: Logical: If TRUE, extract holidays from date columns.

holidays
: Character vector: Holidays to flag when add_holidays is TRUE, named as timeDate holiday functions, e.g. "USLaborDay", "ChristmasDay". See timeDate::listHolidays() for the valid names. Note that unprefixed names are not US holidays: "LaborDay" is May 1.

exclude
: Optional Integer vector: Exclude these columns from preprocessing.

Details

complete_cases, remove_duplicates and remove_cases_thres drop cases. A preprocessor fitted by train is replayed on validation, test and whatever stats::predict() is handed, and a prediction call has nothing it may drop: asked for n rows it must return n predictions.

remove_features_thres decides which columns to drop from the data in front of it. Under resampling each fold would learn that set from its own training subset, so the folds would train over different feature spaces and their metrics would not average into an estimate of one model. Name the columns with remove_features instead, which drops the same ones in every fold.

impute_type = "micePMM" imputes only the data 'mice' is run on, so a fitted preprocessor cannot apply it to validation, test or new data. The default impute_type is "meanMode"; see the Imputation section.

All four operations and micePMM imputation are available in setup_Preprocessor, for preprocess on a dataset in hand, where the outcome is still attached and what was dropped is visible.

Value

SupervisedPreprocessorConfig object.

Imputation

The preferred way to handle missing values is an algorithm that accepts them, which needs no imputation: the tree-based CART, LightCART, LightGBM, LightRF, LightRuleFit and Ranger. For an algorithm that requires complete data, such as GLM or GLMNET, use impute_type = "meanMode", the default: it stores one value per feature, computed on the training data, and adds little memory or computation.

impute_type = "missRanger" fits one random forest per feature with missing values. train fits the preprocessor on every training set it uses: once per outer resample, and, when hyperparameters are tuned, once per tuning resample and hyperparameter combination as well. For example, 10 outer resamples with 5-fold tuning over 20 combinations fit missRanger 10 x (5 x 20 + 1) = 1010 times. Each fitted model, one per outer resample, keeps its forests and a copy of its training data, which predict() draws starting values from. Choose it only where a multivariate imputation is needed, and reduce num.trees and maxiter in impute_missRanger_params for large data.

A fitted missRanger imputer has three further properties:

- It is an R object, held by the fitted model. A config or record written to JSON carries the method and its parameters, not the forests, so a model rebuilt from JSON cannot impute new data with them.
- It has a forest only for features with missing values in the training data. Other features take their impute_numeric or impute_categorical value, computed on the training data, when they are missing in new data.
- predict() stops on a factor level of an imputed feature that is absent from the training data.

impute_type = "micePMM" imputes only the data 'mice' is run on and is available in setup_Preprocessor only.

Author(s)

EDG

Examples

```
preproc_config <- setup_SupervisedPreprocessor(impute = TRUE)
preproc_config
```

`setup_TabNet` *Setup TabNet Hyperparameters*

Description

Setup hyperparameters for TabNet training.

Usage

```
setup_TabNet(
  batch_size = 500L,
  penalty = 0.001,
  clip_value = NULL,
  loss = "auto",
  epochs = 50L,
  drop_last = FALSE,
  decision_width = NULL,
  attention_width = NULL,
  num_steps = 3L,
  feature_reusage = 1.3,
  mask_type = "sparsemax",
  virtual_batch_size = 65536L,
  valid_split = 0,
  learn_rate = 0.02,
  optimizer = "adam",
  lr_scheduler = NULL,
  lr_decay = 0.1,
  step_size = 30L,
  checkpoint_epochs = 10L,
  cat_emb_dim = 1L,
  num_independent = 2L,
  num_shared = 2L,
  num_independent_decoder = 1L,
  num_shared_decoder = 1L,
  momentum = 0.02,
  pretraining_ratio = 0.5,
  device = NULL,
  importance_sample_size = NULL,
  early_stopping_monitor = "auto",
  early_stopping_tolerance = 0,
  early_stopping_patience = 0L,
  num_workers = 0L,
  skip_importance = FALSE,
```

```
  ifw = FALSE
)
```

Arguments

batch_size	(Tunable) Integer [1, Inf): Batch size.
penalty	(Tunable) Numeric [0, Inf): Regularization penalty.
clip_value	(Tunable) Optional Numeric: Gradient clip value. NULL clips no gradients.
loss	(Tunable) Character: Loss function.
epochs	(Tunable) Integer [1, Inf): Number of epochs.
drop_last	(Tunable) Logical: If TRUE, drop last batch.
decision_width	(Tunable) Optional Integer [1, Inf): Decision prediction layer width. NULL uses attention_width, or 8 where both are NULL.
attention_width	(Tunable) Optional Integer [1, Inf): Attention embedding width. NULL uses decision_width, or 8 where both are NULL.
num_steps	(Tunable) Integer [1, Inf): Number of steps.
feature_reusage	(Tunable) Numeric [0, Inf): Feature reusage.
mask_type	(Tunable) Character {"sparsemax", "entmax"}: Mask type.
virtual_batch_size	(Tunable) Integer [1, Inf): Virtual batch size.
valid_split	(Tunable) Numeric [0, 1): Validation split.
learn_rate	(Tunable) Numeric (0, Inf): Learning rate.
optimizer	Character: Optimizer name, resolved by the tabnet backend.
lr_scheduler	Optional Character {"step", "reduce_on_plateau"}: Learning-rate scheduler.
lr_decay	(Tunable) Numeric [0, 1]: Learning rate decay.
step_size	(Tunable) Integer [1, Inf): Step size.
checkpoint_epochs	(Tunable) Integer [1, Inf): Checkpoint epochs.
cat_emb_dim	(Tunable) Integer [1, Inf): Categorical embedding dimension.
num_independent	(Tunable) Integer [1, Inf): Number of independent Gated Linear Units (GLU) at each step of the encoder.
num_shared	(Tunable) Integer [1, Inf): Number of shared Gated Linear Units (GLU) at each step of the encoder.
num_independent_decoder	(Tunable) Integer [1, Inf): Number of independent GLU layers for pretraining.
num_shared_decoder	(Tunable) Integer [1, Inf): Number of shared GLU layers for pretraining.
momentum	(Tunable) Numeric [0, Inf): Momentum.

pretraining_ratio
: (Tunable) Numeric [0, 1]: Pretraining ratio.

device
: Optional Character: Deprecated and ignored. Set the compute device in the `execution_config` passed to train, e.g. `setup_FutureExecution(device = "mps")`. Unset there, TabNet uses `"cuda"` where available and `"cpu"` otherwise; the Apple silicon GPU, slower than the CPU for TabNet at every size rtemis benchmarked, runs only when named.

importance_sample_size
: (Tunable) Optional Integer [1, Inf): Cases drawn, with replacement, from the training cases left after the validation split to compute feature importance. Unset draws as many as there are such cases, at most 100,000. Applies only when `skip_importance` is FALSE.

early_stopping_monitor
: (Tunable) Character {"auto", "valid_loss", "train_loss"}: Early stopping monitor.

early_stopping_tolerance
: (Tunable) Numeric [0, Inf): Minimum relative improvement to reset the patience counter.

early_stopping_patience
: (Tunable) Integer [0, Inf): Number of epochs without improving before stopping.

num_workers
: Integer [0, Inf): Number of subprocesses for data loading.

skip_importance
: Logical: If TRUE, skip importance calculation.

ifw
: (Tunable) Logical: If TRUE, use Inverse Frequency Weighting in classification.

Value

TabNetHyperparameters object.

Author(s)

EDG

Examples

```
tabnet_hyperparams <- setup_TabNet(epochs = 100L, learn_rate = 0.01)
tabnet_hyperparams
```

---

setup_TimePartition *Set up a* TimePartitionConfig

Description

A split by time order: the earliest-ordered cases train, the latest-ordered are held out.

Usage

```
setup_TimePartition(column = NULL, train_p = 0.75)
```

Arguments

column	Optional Character: Name of the column to order cases by before splitting.
train_p	Numeric (0, 1): Fraction of the earliest-ordered cases assigned to training.

Value

`TimePartitionConfig` object.

Author(s)

EDG

Examples

```
setup_TimePartition(column = "visit_date", train_p = 0.8)
```

---

setup_tSNE *Setup tSNE config.*

Description

Setup tSNE config.

Usage

```
setup_tSNE(
  k = 2L,
  initial_dims = 50L,
  perplexity = 30,
  theta = 0.5,
  check_duplicates = TRUE,
  pca = TRUE,
  partial_pca = FALSE,
  max_iter = 1000L,
  verbose = getOption("verbose", FALSE),
  is_distance = FALSE,
  Y_init = NULL,
  pca_center = TRUE,
  pca_scale = FALSE,
  normalize = TRUE,
  stop_lying_iter = if (is.null(Y_init)) 250L else 0L,
  mom_switch_iter = if (is.null(Y_init)) 250L else 0L,
  momentum = 0.5,
```

```
  final_momentum = 0.8,
  eta = 200,
  exaggeration_factor = 12,
  num_threads = NULL,
  features = NULL
)
```

Arguments

k	Integer [1, Inf): Number of components.
initial_dims	Integer [1, Inf): Initial dimensions.
perplexity	Numeric [0, Inf): Perplexity.
theta	Numeric [0, 1]: Speed/accuracy trade-off.
check_duplicates	Logical: If TRUE, check for duplicates.
pca	Logical: If TRUE, perform PCA.
partial_pca	Logical: If TRUE, perform partial PCA.
max_iter	Integer [1, Inf): Maximum number of iterations.
verbose	Logical: If TRUE, print messages.
is_distance	Logical: If TRUE, x is a distance matrix.
Y_init	Optional Matrix: Initial Y matrix.
pca_center	Logical: If TRUE, center PCA.
pca_scale	Logical: If TRUE, scale PCA.
normalize	Logical: If TRUE, normalize.
stop_lying_iter	Integer [0, Inf): Stop lying iterations.
mom_switch_iter	Integer [0, Inf): Momentum switch iterations.
momentum	Numeric: Momentum.
final_momentum	Numeric: Final momentum.
eta	Numeric: Eta.
exaggeration_factor	Numeric: Exaggeration factor.
num_threads	Optional Integer: Deprecated and ignored. Threads come from the execution config: set n_workers_algorithm in the execution_config passed to decomp.
features	Optional Character vector: Names of at least 2 distinct feature columns to decompose. NULL decomposes all numeric features.

Details

Get more information on the config by running ?Rtsne::Rtsne.

Value

tSNEConfig object.

Author(s)

EDG

Examples

```
tSNE_config <- setup_tSNE(k = 3L)
tSNE_config
```

`setup_UMAP` *Setup UMAP config.*

Description

Setup UMAP config.

Usage

```
setup_UMAP(
  k = 2L,
  n_neighbors = 15L,
  init = "spectral",
  metric = "euclidean",
  n_epochs = NULL,
  learning_rate = 1,
  scale = TRUE,
  features = NULL
)
```

Arguments

`k`	Integer [1, Inf): Number of components.
`n_neighbors`	Integer [1, Inf): Number of neighbors.
`init`	Character: Initialization type. See `uwot::umap "init"`.
`metric`	Character {"euclidean", "cosine", "manhattan", "hamming", "categorical"}: Distance metric.
`n_epochs`	Optional Integer [1, Inf): Number of epochs.
`learning_rate`	Numeric [0, Inf): Learning rate.
`scale`	Logical: If TRUE, scale input data before doing UMAP.
`features`	Optional Character vector: Names of at least 2 distinct feature columns to decompose. `NULL` decomposes all numeric features.

Details

A high n_neighbors value may give error in some systems: "Error in irlba::irlba(L, nv = n, nu = 0, maxit = iters) : function 'as_cholmod_sparse' not provided by package 'Matrix'"

Value

UMAPConfig object.

Author(s)

EDG

Examples

```
umap_config <- setup_UMAP(k = 3L)
umap_config
```

---

setup_VariationalAutoencoder

Setup Variational Autoencoder config.

Description

A fully connected variational autoencoder built and trained with torch: the encoder maps each case to a normal distribution over k latent dimensions, and the decoder maps a draw from it back to the features.

Usage

```
setup_VariationalAutoencoder(
  k = 2L,
  hidden_units = NULL,
  activation = "relu",
  norm = NULL,
  dropout = 0,
  input_dropout = 0,
  weight_decay = 0,
  input_noise = 0,
  loss = "mse",
  optimizer = "adamw",
  lr = 0.001,
  beta1 = NULL,
  beta2 = NULL,
  eps = NULL,
  momentum = NULL,
  lr_scheduler = NULL,
  batch_size = NULL,
```

```
  max_epochs = 100L,
  patience = 10L,
  max_grad_norm = NULL,
  validation_fraction = 0.1,
  beta = 1,
  features = NULL
)
```

Arguments

k	Integer [1, Inf): Number of components, the width of the latent layer.
hidden_units	Optional Integer [1, Inf) vector: Units in each hidden layer of the encoder, input side first.
activation	Character {"relu", "gelu", "silu", "elu", "selu", "leaky_relu", "tanh"}: Activation applied after every hidden layer.
norm	Optional Character {"batch_norm", "layer_norm"}: Normalization applied in every hidden layer. NULL applies none.
dropout	Numeric [0, 1): Dropout probability applied after every hidden layer.
input_dropout	Numeric [0, 1): Probability of zeroing each standardized input during training.
weight_decay	Numeric [0, Inf): L2 penalty, decoupled from the gradient under the adamw optimizer.
input_noise	Numeric [0, Inf): Standard deviation of the Gaussian noise added to the standardized inputs during training.
loss	Character {"mse", "l1", "smooth_l1"}: Reconstruction loss.
optimizer	Character {"adamw", "adam", "sgd", "rmsprop"}: Optimization algorithm.
lr	Numeric (0, Inf): Learning rate.
beta1	Optional Numeric [0, 1): Exponential decay rate of the first moment estimate. Applies to the adam and adamw optimizers.
beta2	Optional Numeric [0, 1): Exponential decay rate of the second moment estimate. Applies to the adam and adamw optimizers.
eps	Optional Numeric (0, Inf): Term added to the denominator for numerical stability. Applies to the adam, adamw and rmsprop optimizers.
momentum	Optional Numeric [0, Inf): Momentum factor. Applies to the sgd and rmsprop optimizers.
lr_scheduler	Optional Character {"step", "cosine_annealing", "one_cycle", "reduce_on_plateau"}: Learning-rate schedule. NULL holds the learning rate fixed.
batch_size	Optional Integer [1, Inf): Cases per optimization step.
max_epochs	Integer [1, Inf): Largest number of passes over the training cases.
patience	Integer [1, Inf): Epochs without improvement of the held-out loss before stopping early.
max_grad_norm	Optional Numeric (0, Inf): Clip the gradient norm to this value before each step. NULL does not clip.

`validation_fraction`
: Numeric [0, 1): Fraction of the cases held out for early stopping.

`beta`
: Numeric [0, Inf): Weight of the KL divergence from the prior.

`features`
: Optional Character vector: Names of at least 2 distinct feature columns to decompose. `NULL` decomposes all numeric features.

Details

Objective. For each case, the reconstruction loss summed over features plus `beta` times the KL divergence of the case's latent distribution from the standard normal, summed over the `k` dimensions; the mean over cases is minimized. `beta = 1` is the standard variational autoencoder (with `loss = "mse"`, up to constants, the negative evidence lower bound of a unit-variance Gaussian decoder); `beta > 1` is a beta-VAE, trading reconstruction for components closer to independent; `beta = 0` drops the prior and leaves a noisy autoencoder.

Components are the means of the latent distributions, so decomp, `apply_decomposition()` and reconstruct are deterministic: sampling happens only in training. reconstruct decodes the means.

Architecture, input standardization, denoising, early stopping, and the execution config's seed, threads and device work as in setup_Autoencoder. The loss reported during training and for early stopping is the full objective, so it is not on the scale of setup_Autoencoder's, which averages the reconstruction loss over features.

Value

VariationalAutoencoderConfig object.

Author(s)

EDG

Examples

```
vae_config <- setup_VariationalAutoencoder(k = 2L, beta = 4)
vae_config
```

---

setup_XLSXIngest *Set up an* XLSXIngestConfig

Description

Creates the settings `ingest()` uses to read a spreadsheet.

Usage

```
setup_XLSXIngest(
  columns = NULL,
  character2factor = TRUE,
  clean_colnames = TRUE,
  remove_duplicates = FALSE,
  sheet = 1L,
  na_strings = ""
)
```

Arguments

columns	Optional Named vector: Declared column types, keyed by column name, in the profile/r/v1 vocabulary – c(Age = "integer"). A column named here is converted to the declared type after reading, replacing the type the reader assigned.
character2factor	Logical: If TRUE, read character columns as factors.
clean_colnames	Logical: If TRUE, normalize column names.
remove_duplicates	Logical: If TRUE, drop rows repeating an earlier row.
sheet	Integer [1, Inf): Sheet number to read.
na_strings	Character vector: Strings to read as missing.

Value

XLSXIngestConfig object.

Author(s)

EDG

Examples

```
setup_XLSXIngest(sheet = 2L)
```

---

set_outcome *Move outcome to last column*

Description

Move outcome to last column

Usage

```
set_outcome(dat, outcome_column)
```

Arguments

dat	tabular dataset.
outcome_column	Character: Name of outcome column.

Value

object of same class as data

Author(s)

EDG

Examples

```
ir <- set_outcome(iris, "Sepal.Length")
head(ir)
```

---

set_positive_class *Set positive class*

Description

Checks and sets the positive class for binary classification in a tabular dataset.

Usage

```
set_positive_class(x, positive_class)
```

Arguments

x	Tabular data, i.e. data.frame, data.table, or tbl_df (tibble): Input dataset. The last column will be treated as the outcome variable.
positive_class	Character scalar: The name of the positive class. Must be one of the levels of the outcome variable.

Value

Tabular data of the same class as x, with the positive class set as the second level of the outcome factor.

Author(s)

EDG

Examples

```
dat <- data.frame(x = rnorm(100), y = factor(sample(c("Case", "Control"), 100, replace = TRUE)))
levels(dat[["y"]])
dat <- set_positive_class(dat, "Case")
levels(dat[["y"]])
```

---

shap_by_level *Contributions of a categorical feature, by the level each case has*

Description

What a per-feature contribution hides once it is averaged across cases: that one level of a diagnosis drives the prediction and the others do nothing.

Usage

```
shap_by_level(x, newdata, features = NULL)
```

Arguments

x	SHAP object.
newdata	tabular data: The cases x explains.
features	Optional Character: Features to summarize. NULL takes every categorical feature in newdata.

Details

A case has exactly one level, so its per-feature contribution is the contribution of the level it has. Grouping cases by level reports how each level contributes.

Splitting a contribution across the columns of an encoding answers a different question. Those values include a term for every level the case does not have; they depend on the encoding, since a contrast-coded factor's reference level has no column; and for a linear model they are identical for every case at the same level. Grouping by level does not depend on the encoding and includes the reference level.

newdata must be the data the explanation was computed on, which is checked against its fingerprint, so contributions are never paired with the values of other rows.

Value

data.table with columns class, feature, level, n, mean, sd, and mean_abs.

Author(s)

EDG

Examples

```
x <- data.frame(
  age = rnorm(100),
  dx = factor(sample(c("none", "diabetes"), 100, TRUE))
)
x[["y"]] <- x[["age"]] + 3 * (x[["dx"]] == "diabetes") + rnorm(100, sd = 0.3)
mod <- train(x, hyperparameters = setup_GLM(), verbosity = 0L)
contributions <- explain(
  mod,
  x[, c("age", "dx")],
  background = x[, c("age", "dx")],
  verbosity = 0L
)
shap_by_level(contributions, x[, c("age", "dx")])
```

---

shap_case *One case's contributions, ordered by how much they moved the prediction*

Description

The three parts of a per-case explanation: the baseline, each feature's contribution, and the prediction.

Usage

```
shap_case(x, newdata, case = 1L, class = NULL)
```

Arguments

x	SHAP object.
newdata	tabular data: The cases x explains.
case	Integer or Character: Which case, by position or row name.
class	Optional Character: Which class's contributions. NULL takes the first, which is the only one for a regression or binary outcome.

Details

Shaped here, following `session_timeline()`: the table is the shared input for rtemis.draw's chart and for rtemislive, so both show the same values.

Features are ordered by the magnitude of their contribution, which is the order a waterfall reads in, and each carries the value it took for this case so the row reads "age = 62, +0.4".

Value

List with `steps` (a data.table of `feature`, `value` and `contribution`), `baseline`, `predicted`, `case` and `class`.

Author(s)

EDG

Examples

```
x <- data.frame(age = rnorm(100), bmi = rnorm(100))
x[["y"]] <- x[["age"]] * 2 + rnorm(100, sd = 0.3)
mod <- train(x, hyperparameters = setup_GLM(), verbosity = 0L)
contributions <- explain(
  mod,
  x[, c("age", "bmi")],
  background = x[, c("age", "bmi")],
  verbosity = 0L
)
shap_case(contributions, x[, c("age", "bmi")], case = 1L)
```

---

shap_long *Every contribution, one row per case per feature*

Description

The long form a beeswarm reads: a point per case per feature, positioned by its contribution and colored by the value the feature took.

Usage

```
shap_long(x, newdata, class = NULL)
```

Arguments

x	SHAP object.
newdata	tabular data: The cases x explains.
class	Optional Character: Which class's contributions. NULL takes the first.

Details

Shaped here, for the reason `shap_case()` gives.

Value

data.table with columns `case`, `feature`, `value`, `numeric_value` and `contribution`. `numeric_value` is NA for a feature that is not numeric, which is what a color scale needs to skip it.

Author(s)

EDG

Examples

```
x <- data.frame(age = rnorm(100), bmi = rnorm(100))
x[["y"]] <- x[["age"]] * 2 + rnorm(100, sd = 0.3)
mod <- train(x, hyperparameters = setup_GLM(), verbosity = 0L)
contributions <- explain(
  mod,
  x[, c("age", "bmi")],
  background = x[, c("age", "bmi")],
  verbosity = 0L
)
head(shap_long(contributions, x[, c("age", "bmi")]))
```

---

show_color_key *Show rtemis color key*

Description

Show rtemis color key

Usage

```
show_color_key()
```

Value

Used for side-effects: prints color key to console. Returns invisible NULL.

Author(s)

EDG

Examples

```
show_color_key()
```

---

size *Size of object*

Description

Returns the size of an object

Usage

```
size(x, verbosity = 1L)
```

Arguments

x	any object with `length()` or `dim()`.
verbosity	Integer: Verbosity level. If > 0, print size to console

Details

If `dim(x)` is NULL, returns `length(x)`.

Value

Integer vector with length equal to the number of dimensions of `x`, invisibly.

Author(s)

EDG

Examples

```
x <- rnorm(20)
size(x)
# 20
x <- matrix(rnorm(100), 20, 5)
size(x)
# 20  5
```

---

table_column_attr *Tabulate column attributes*

Description

Tabulate column attributes

Usage

```
table_column_attr(x, attr = "source", useNA = "always")
```

Arguments

x	tabular data: Input data set.
attr	Character: Attribute to get
useNA	Character: Passed to `table`

Value

table.

Author(s)

EDG

Examples

```
library(data.table)
x <- data.table(
  id = 1:5,
  sbp = rnorm(5, 120, 15),
  dbp = rnorm(5, 80, 10),
  paO2 = rnorm(5, 90, 10),
  paCO2 = rnorm(5, 40, 5)
)
setattr(x[["sbp"]], "source", "outpatient")
setattr(x[["dbp"]], "source", "outpatient")
setattr(x[["paO2"]], "source", "icu")
setattr(x[["paCO2"]], "source", "icu")
table_column_attr(x, "source")
```

---

train *Train Supervised Learning Models*

Description

Preprocess, tune, train, and test supervised learning models using nested resampling in a single call.

Usage

```
train(
  x,
  dat_validation = NULL,
  dat_test = NULL,
  weights = NULL,
  preprocessor_config = NULL,
  decomposition_config = NULL,
  hyperparameters = NULL,
  tuner_config = NULL,
  outer_resampling_config = NULL,
  execution_config = setup_FutureExecution(),
  question = NULL,
  outdir = NULL,
  preflight = FALSE,
  verbosity = 1L,
  ...
)
```

Arguments

x	Tabular data, i.e. data.frame, data.table, or tbl_df (tibble): Training set data.
dat_validation	Tabular data: Validation set data.
dat_test	Tabular data: Test set data.
weights	Optional vector of case weights.
preprocessor_config	Optional SupervisedPreprocessorConfig object: Setup using setup_SupervisedPreprocessor.
decomposition_config	Optional DecompositionConfig object: Setup using a decomposition setup_* function.
hyperparameters	Hyperparameters object: Setup using one of setup_* functions.
tuner_config	TunerConfig object: Setup using setup_GridSearch.
outer_resampling_config	Optional ResamplerConfig object: Setup using one of the setup_* resampler functions, e.g. setup_KFold. This defines the outer resampling method, i.e. the splitting into training and test sets for the purpose of assessing model performance. If NULL, no outer resampling is performed; dat_test then provides a single test set for assessing model performance.
execution_config	ExecutionConfig object: Setup using setup_SerialExecution, setup_FutureExecution or setup_MiraiExecution. The setup function selects the backend; each sets the number of workers per dispatch level, and setup_FutureExecution additionally sets the future plan.
question	Optional character string defining the question that the model is trying to answer.
outdir	Character, optional: String defining the output directory. The fitted model is saved there as train_<algorithm>.rds, alongside a run record, train_<algorithm>.record.json, stating what the run did: every value resolved, where each came from, and what produced it. See write_record.
preflight	Logical: If TRUE, check the configuration against x with validate_config before training and stop on any finding of severity "error". Findings of severity "warning" are reported and training continues. Defaults to FALSE, except for a SuperConfigTabular, where it defaults to TRUE. Pass FALSE explicitly to skip it.
verbosity	Integer: Verbosity level.
...	Not used.

Details

Online book & documentation

See docs.rtemis.org/r for detailed documentation.

Preprocessing

There are many different stages at which preprocessing could be applied, when running a supervised learning pipeline with nested resampling. Some operations are best done before passing data to train():

- Duplicate rows should be removed before resampling, so that duplicates do not end up in different resamples, e.g. one in training and one in test.
- Constant columns should be removed before resampling. A column may appear constant in a small resample, even if it is not constant in the full dataset. Removing it inconsistently will throw an error during prediction.
- All data-dependent preprocessing steps need to be performed on training data only and applied on validation and test data, e.g. scaling, centering, imputation.

User-defined preprocessing through `preprocessor_config` is applied on training set data, the learned parameters are stored in the returned Supervised or SupervisedRes object, and the preprocessing is applied on validation and test data.

For missing values, prefer an algorithm that accepts them, such as LightGBM, LightRF, Ranger or CART. For an algorithm that requires complete data, impute with `setup_SupervisedPreprocessor(impute = TRUE)`, which uses mean and mode imputation by default. missRanger imputation is fitted on every training set, including every tuning resample, and multiplies the memory and computation of a run; see the Imputation section of setup_SupervisedPreprocessor.

Binary Classification

For binary classification, the outcome should be a factor where *the 2nd level corresponds to the positive class*.

Resampling

An outer resampling method with replacement combined with inner resampling (for tuning) can place duplicated cases from the outer resampling in both the training and test sets of the inner resamples, which underestimates test error.

Reproducibility

For **outer resampling**, set a seed when defining `outer_resampling_config`, e.g.

```
outer_resampling_config = setup_KFold(n_resamples = 10L, seed = 2026L)
```

For **tuning with inner resampling**, set a seed when defining `tuner_config`, e.g.

```
tuner_config = setup_GridSearch(
  resampler_config = setup_KFold(n_resamples = 5L, seed = 2027L)
)
```

Parallelization

There are three levels of parallelization that may be used during training:

1. Algorithm training (e.g. a parallelized learner like LightGBM)
2. Tuning (where the full hyperparameter combination x inner resample cross product is processed in parallel, not just the resamples)
3. Outer resampling (where multiple outer resamples can be processed in parallel)

`train()` assigns all workers to exactly one of these levels, so that no two levels compete for the same cores. The level is chosen by the first of these that applies: the algorithm parallelizes itself, tuning is needed, outer resampling is set. A run that needs none of them is sequential.

The workers, backend, and future plan all come from `execution_config`; see setup_FutureExecution.

Reproducibility under parallelization

`execution_config` carries the master seed for the run's computation RNG, from which one independent substream per outer resample is derived and assigned by fold index. Results therefore do not depend on the backend, the number of workers, or the order folds happened to finish in: a parallel run reproduces its sequential counterpart exactly.

Value

Object of class `Regression(Supervised)`, `RegressionRes(SupervisedRes)`, `Classification(Supervised)`, or `ClassificationRes(SupervisedRes)`.

Author(s)

EDG

Examples

```
iris_c_lightRF <- train(
   iris,
   hyperparameters = setup_LightRF(),
   outer_resampling_config = setup_KFold(n_resamples = 3L)
)
```

---

transformed *Get transformed data from* `Decomposition`

Description

Returns the components a `Decomposition` computed for the data it was fitted on.

Usage

```
transformed(x)
```

Arguments

x	`Decomposition`: A `Decomposition` object.

Details

Each call returns a new data.table, so modifying the result by reference, with `:=` or `set()`, leaves `x` unchanged. The columns are the components only, named as in the output of `apply_decomposition()`, which computes the components of new data.

Value

data.table: One row per case, one column per component.

Author(s)

EDG

Examples

```
iris_pca <- decomp(exc(iris, "Species"), algorithm = "PCA", verbosity = 0L)
transformed(iris_pca)
```

---

tune_over *Mark the values to tune a hyperparameter over*

Description

Declares that a hyperparameter is *searched over*. It is the only way to express a search space: a bare vector is always a value, so nothing has to be inferred from how many elements it happens to have.

Usage

```
tune_over(...)
```

Arguments

`...`	The candidates to search over: one per argument, or a single list or vector read against the type the hyperparameter declares.

Details

Give it the candidates. Several arguments are one candidate each; a single argument holding several values *is* the candidates, so a computed grid is passed directly:

```
setup_LightRF(max_depth = tune_over(3L, 4L, 5L))
setup_LightRF(lambda_l2 = tune_over(10^seq(-4, 0, length.out = 5)))
setup_MLP(hidden_units = tune_over(c(12L, 6L, 2L), c(14L, 12L, 6L, 2L)))
```

For a **vector-valued** hyperparameter one candidate is itself a vector, so pass the candidates as separate arguments or as a list – `tune_over(list(c(12L, 6L), c(24L, 12L)))`. A single bare vector there would be one architecture, and is an error saying so.

Passing it to a hyperparameter that is not tunable is an error: that hyperparameter accepts a value only.

Value

`HyperparameterCandidates` object.

Author(s)

EDG

Examples

```
hp <- setup_LightRF(max_depth = tune_over(3L, 4L, 5L))
hp@max_depth
# A bare vector is a value, never a search space.
setup_LightRF(max_depth = 3L)@max_depth
```

---

tuning_grid *Tuning grid of a Hyperparameters object*

Description

The set of hyperparameter combinations a grid search would fit, one per row. Derived from the object, so it can be inspected before `train()` is called.

Usage

```
tuning_grid(x)
```

Arguments

x	`Hyperparameters` object.

Details

The grid is the cross product of the search values, reduced by two rules:

- A hyperparameter that declares it applies only under certain values of another is set to `NA` – meaning "left unset for this fit" – in the rows that do not meet them. `reduce_basis` applies only at `smoothness_orders` of 0, so a search over both drops it from the higher-order rows.
- Rows that become identical once that is done are collapsed, so a combination is never fit, scored, and ranked twice.

`tune_GridSearch()` fits exactly these rows, so what this prints is what will run.

Value

data.frame with one row per combination and one column per searched hyperparameter, or NULL if nothing needs tuning.

Author(s)

EDG

Examples

```
# reduce_basis applies only at smoothness order 0, so the six combinations
# of the cross product reduce to four.
tuning_grid(
  setup_HAL(
    smoothness_orders = tune_over(0L, 1L, 2L),
    reduce_basis = tune_over(0.1, 0.5)
  )
)
```

`validate_config` *Validate an rtemis config, against the schema and optionally against data*

Description

Checks a config document – as parsed from JSON, or already resolved – and reports what is wrong with it. With no `data`, only the schema is checked: whether the document reconstructs into a valid rtemis config object. With `data`, the config is additionally checked against the dataset it would run on, catching the errors a schema cannot see: an outcome column that is not there, more folds than the rarer class has cases, predictors that never vary.

Usage

```
validate_config(config, data = NULL, outcome = NULL, step = NULL)
```

Arguments

`config`	Named list parsed from a config document (carrying its `$schema`), or a resolved rtemis config object. A resolved object is valid by construction, so the schema check is skipped for it.
`data`	Optional tabular data: The dataset the config would run on. NULL checks the schema only.
`outcome`	Optional Character: Name of the outcome column in `data`. NULL takes rtemis's convention, the last column – but only for a config that designates an outcome at all; a decomposition or clustering config has no outcome and every column is a feature.
`step`	Optional Integer [1, Inf): Position of this config in the plan it came from, recorded on every finding. NULL when the config stands alone.

Details

Nothing is thrown. Every finding is a `Diagnostic` carrying a stable `code`, a `severity`, a technical `message`, a plain-language `plain` text, the `evidence` behind it, and – where a deterministic one exists – a `fix`, an RFC 6902 JSON Patch that resolves it. An empty result means the config is clean.

Each data check runs only when the config carries the part it is about: the resampling checks need a resampler, the outcome checks need an outcome. A config carrying neither is checked for what it

does declare and reports nothing about what it does not, so a preprocessing step and a training step each get the checks that apply to them.

Value

`Diagnostics` object. Empty when nothing is wrong.

Author(s)

EDG

Examples

```
# A config that does not reconstruct: one finding, and no data checks.
bad <- list(
  `$schema` = "https://schema.rtemis.org/supervised/r/v1/schema.json",
  n_foldz = 10L
)
validate_config(bad)

# A resolved config with nothing wrong: an empty Diagnostics.
validate_config(setup_SuperConfig(hyperparameters = setup_LightRF()))
```

---

varimp_ranking *Rank a Model's Variable Importance*

Description

Summarize each importance measure across resamples and rank the variables by it, as `plot_varimp()` draws them.

Usage

```
varimp_ranking(x)
```

Arguments

`x`	`Supervised`, `SupervisedRes` or `SHAP`: Fitted model result or explanation.

Details

Each measure is ranked as its `direction` declares: by larger values, by larger absolute values, or by smaller values; a measure with direction `"none"` is listed by decreasing value. A variable's summary is the mean of its available scores across resamples, computed by `rtemis.draw::summarize_varimp()`; an `"absolute"` measure ranks by the absolute value of that mean. NA scores and resamples without importance are excluded. A measure with no available score is omitted.

The table is in rank order within each measure, and in resample order within each variable, so the rows of the first `n` ranks of a measure are the rows of the `n` most important variables.

Value

data.frame with one row per measure, variable and resample with an available score: `measure`, `variable`, `fold` (resampled results only), `value` (the score), `summary` (the variable's mean across resamples), and `rank` (1 for the most important variable of the measure).

Author(s)

EDG

Examples

```
mod <- train(iris, hyperparameters = setup_CART(), verbosity = 0L)
head(varimp_ranking(mod))
```

---

varimp_table *Variable importance as one table*

Description

Combine the variable importance measures of a model into one table: one row per predictor or model term and one column per measure.

Usage

```
varimp_table(x)
```

Arguments

`x`	`VariableImportance` object, as returned by get_varimp.

Details

Columns follow the order of the measures. A predictor a measure does not cover is `NA` in that measure's column: some backends report only the predictors used in a split. A measure named `variable` or `fold`, or one repeating an earlier column, is labeled "`<name> (measure)`", since those column names carry the table's structure.

The table holds the values only. What each measure quantifies, the cases it was computed on, and how it ranks predictors are on the measure itself: `get_varimp(model)@measures[["<name>"]]`.

Value

data.frame with a `variable` column and one numeric column per measure.

Author(s)

EDG

Examples

```
mod <- train(iris, hyperparameters = setup_CART(), verbosity = 0L)
varimp_table(get_varimp(mod))
```

`writeup` *Write the Methods and Results sections for a trained model*

Description

Describe a trained supervised model as the Methods and Results sections of a paper: the data, preprocessing, algorithm and settings, tuning, evaluation design, performance measures and statistical methods, and software, then the sample, performance with its uncertainty, the comparison with a baseline, and the hyperparameters tuning selected.

Two tables report the hyperparameters. The main table lists the algorithm's primary hyperparameters, every hyperparameter that was tuned or specified, and the values selected during fitting (such as a number of boosting rounds chosen by early stopping); the supplementary table lists every hyperparameter that applied to the fit, its value, how it was chosen and the values tuning evaluated. A hyperparameter left unset is reported with what that means for the fit. `@hyperparameters` holds the rows of both tables.

Usage

```
writeup(x, confidence_level = NULL, include_hyperparameters = NULL, ...)
```

Arguments

`x`	`Supervised` or `SupervisedRes` object: A trained model, as returned by train.
`confidence_level`	Optional Numeric (0, 1): Confidence level of the intervals of a single-split model. NULL uses 0.95.
`include_hyperparameters`	Optional Character: Hyperparameters the main hyperparameter table lists in place of the algorithm's primary hyperparameters, beside those that were tuned or specified and the values selected during fitting. NULL uses the primary hyperparameters; `character(0)` adds none.
`...`	Not used.

Details

The text is assembled from the model and its review, so it states only what rtemis recorded. Every number in the text is read from the model or the review: the returned object holds each one in `@values` with the path of the field it came from, and the paragraphs in `@sections` are templates in which each number is a `{key}` token naming a row of `@values`. Statistical methods are those `review()` used, cited in `@references` together with R, rtemis and the packages that fitted the model, at the versions recorded when the model was trained.

A Methods section also states the source of the data, the study design, how the outcome and predictors were measured, and anything done to the data before it was passed to rtemis. The model does not record these, so the writeup lists them in `@not_reported` for the author to add.

Print the result to read it; write_text writes it as Markdown.

Value

`SupervisedWriteup` object.

Author(s)

EDG

References

Clopper CJ, Pearson ES (1934). The use of confidence or fiducial limits illustrated in the case of the binomial. Biometrika, 26(4), 404-413.

DeLong ER, DeLong DM, Clarke-Pearson DL (1988). Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach. Biometrics, 44(3), 837-845.

McNemar Q (1947). Note on the sampling error of the difference between correlated proportions or percentages. Psychometrika, 12(2), 153-157.

Examples

```
idx <- c(1:40, 51:90, 101:140)
mod <- train(
  iris[idx, ],
  dat_test = iris[-idx, ],
  hyperparameters = setup_CART(),
  verbosity = 0L
)
writeup(mod)

# Resampled
mod_res <- train(
  iris,
  hyperparameters = setup_CART(),
  outer_resampling_config = setup_KFold(5L),
  verbosity = 0L
)
writeup(mod_res)
```

`write_config` *Write an rtemis config to a JSON file*

Description

Write a config object to a schema.rtemis.org JSON file that can be read back with read_config and consumed by rtemislive and the `rtemis` CLI. The emitted JSON carries a `$schema` field identifying the config family, so the reader can dispatch to the right object:

Usage

```
write_config(x, file, overwrite = FALSE, verbosity = 1L)
```

Arguments

`x`	A `SuperConfig`, `DecomposeConfig`, `ClusterConfig`, `DecompositionConfig`, `ClusteringConfig`, or `PreprocessorConfig` object.
`file`	Character: Path to output JSON file.
`overwrite`	Logical: If TRUE, overwrite an existing file.
`verbosity`	Integer: Verbosity level.

Details

- `SuperConfig` (from setup_SuperConfig) -> supervised schema.
- `DecomposeConfig` (from setup_DecomposeConfig) -> decompose schema.
- `ClusterConfig` (from setup_ClusterConfig) -> cluster schema.
- `DecompositionConfig` (from `setup_PCA()`, `setup_ICA()`, ...) -> decomposition schema.
- `ClusteringConfig` (from `setup_KMeans()`, `setup_DBSCAN()`, ...) -> clustering schema.
- `PreprocessorConfig` (from setup_Preprocessor) -> preprocessor schema.

A config is an **input**: what a run is being asked to do. Fields it omits fall back to their `setup_*` defaults when read, so a config stays valid as those defaults change. A **record**, by contrast, states what one run did, with every value resolved – see write_record. The two are separate functions, so a caller names the artifact they want.

Value

`x`, invisibly.

Author(s)

EDG

Examples

```
x <- setup_SuperConfig(
  dat_training_path = "~/Data/iris.csv",
  hyperparameters = setup_LightRF(),
  outer_resampling_config = setup_KFold(),
  question = "Can we tell iris species apart given their measurements?",
  outdir = "models/"
)
tmpfile <- file.path(tempdir(), "rtemis.json")
write_config(x, tmpfile, overwrite = TRUE)
```

---

write_record *Write a run record to a JSON file*

Description

A record states what a run did: every value resolved, where each one came from, and what produced it. A separate function from write_config, since the two artifacts answer different questions.

Usage

```
write_record(x, file, overwrite = FALSE, verbosity = 1L)
```

Arguments

x	Fitted model object, or a record list from record.
file	Character: Path to write to.
overwrite	Logical: If TRUE, overwrite an existing file.
verbosity	Integer: Verbosity level.

Details

Three things distinguish a record from the config it came from:

- **Every field is present and resolved.** A config omits what the user did not set, so a reader applies defaults; a record leaves nothing to them, and writes an unset field as an explicit `null`.
- **Each value says where it came from**, in a parallel `origin` map: `user`, `default`, `derived` (computed from the data), `tuned`, or `unset` for a field a failed run never reached.
- `provenance` records the rtemis and R versions, platform, timing, outcome, and a fingerprint of the data.

For a supervised run the top level is what was *asked for* and `folds` is what *ran*, one entry per model fitted – outer resampling resolves different values in each fold, so a single resolved value at the top level would state something no fold did.

A supervised record also states **what the run scored**. `metrics` holds each sample's headline row as a flat metric-to-value map, averaged across outer resamples, with `metrics_sd` beside it for the

spread (null for a single fit, which has none). The full metrics – the confusion matrix, the per-class rows – are in each fold's own metrics. The flat block is read in one lookup, without averaging, so a directory of records can be ranked.

train, decomp and cluster call this automatically when given an outdir. A record is not a config: feeding one to read_config is an error, since its resolved values would silently pin settings the new call should decide for itself.

Unlike a config, a record is **not compacted**: an unset field is written as an explicit null, so nothing in it falls back to a reader's defaults.

Value

x, invisibly.

Author(s)

EDG

Examples

```
## Not run:
mod <- train(iris, hyperparameters = setup_CART())
write_record(mod, "train_CART.record.json")

## End(Not run)
```

---

write_result *Write a portable result with optional Parquet payloads*

Description

Eligible outcomes, predictions, probabilities and resampling indices can stay inline or reference compressed Parquet files beside the JSON document. The outer resample structure remains in JSON. Fitted runtime objects are excluded. Move the JSON file and its accompanying directory together.

Usage

```
write_result(
  x,
  file,
  storage = c("auto", "inline", "parquet"),
  inline_max_bytes = 65536,
  base_dir = dirname(file),
  overwrite = FALSE
)
```

Arguments

x	Published S7 report object.
file	Character: Output JSON path.
storage	Character: "inline", "parquet", or "auto".
inline_max_bytes	Numeric: In auto mode, maximum estimated uncompressed payload bytes kept inline: eight per numeric value or categorical code, one per Boolean, or UTF-8 string bytes plus eight per string.
base_dir	Character: Directory resolving existing DataRef paths. Defaults to the output directory.
overwrite	Logical: Replace an existing JSON file.

Value

x, invisibly.

Examples

```
result <- resample(1:6, setup_KFold(n_resamples = 2L), verbosity = 0L)
directory <- tempfile("result-")
dir.create(directory)
write_result(result, file.path(directory, "result.json"), storage = "parquet")
unlink(directory, recursive = TRUE)
```

---

write_text *Write a report as text*

Description

Write the human-readable rendering of a writeup, review or AI review to a file: the text print shows, laid out for a document. write_result writes the same objects as JSON documents for programs to read.

Usage

```
write_text(x, file, format = NULL, overwrite = FALSE, verbosity = 1L, ...)
```

Arguments

x	SupervisedWriteup (from writeup), SupervisedReview (from review) or AISupervisedReview (from ai_review) object.
file	Character: Path of the file to write.
format	Optional Character {"markdown"}: Output format. NULL selects the format from the extension of file: ".md" or ".markdown" for Markdown.
overwrite	Logical: If TRUE, replace an existing file.
verbosity	Integer: Verbosity level.
...	Not used.

Details

Markdown output starts at ## headings and uses pipe tables, ready to convert with Pandoc or Quarto or to paste into a manuscript.

Value

The path of the file written, invisibly.

Author(s)

EDG

Examples

```
idx <- c(1:40, 51:90, 101:140)
mod <- train(
  iris[idx, ],
  dat_test = iris[-idx, ],
  hyperparameters = setup_CART(),
  verbosity = 0L
)
write_text(writeup(mod), file.path(tempdir(), "writeup.md"))
write_text(review(mod), file.path(tempdir(), "review.md"))
```

%BC% *Binary matrix times character vector*

Description

Collapse a binary indicator matrix into a character vector of labels. For each row, the labels of all columns equal to 1 are pasted together, comma-separated. This is the inverse of multi-hot encoding: unlike one_hot2factor, which assumes a single 1 per row, `%BC%` supports rows with multiple 1s (multi-label data, e.g. multiple-choice survey responses, tags, or set membership). Rows of all zeros return `NA`.

Usage

```
x %BC% labels
```

Arguments

`x`	A binary matrix or data.frame
`labels`	Character vector length equal to `ncol(x)`

Value

a character vector

Author(s)

EDG

Examples

```
# Multi-hot matrix: each row can belong to multiple categories
x <- rbind(
  c(1, 0, 1),
  c(0, 1, 0),
  c(1, 1, 1),
  c(0, 0, 0)
)
labels <- c("apple", "banana", "cherry")
x %BC% labels
# -> "apple,cherry", "banana", "apple,banana,cherry", NA
```

Index

%BC%, 285

ai_review, 8, *284*
applicable_metrics, 10
apply_decomp, 10
apply_decomposition, 11
apply_decomposition(), *10, 11*, *35*, *37, 38*, *107*, *129*, *262*, *273*
apply_preprocessor, 12, *67*, *100*, *222, 223*, *225*, *251*
assign_clusters, 13
available_algorithms *(*available_supervised*)*, 14
available_calibration, *19*
available_calibration *(*available_supervised*)*, 14
available_clustering, *7*
available_clustering *(*available_supervised*)*, 14
available_decomposition, *7*
available_decomposition *(*available_supervised*)*, 14
available_supervised, *7*, 14
available_supervised(), *63*

base::cut, *223*, *251*
bias_variance, 15

calibrate, 17, *27*, *210*
check_data, 20
class_imbalance, 22
classification_metrics, 21
clean_colnames, *32*, *103*
cluster, *15*, 23, *135*, *283*
clustering_metrics, 24
clusters, 25
clusters(), *14*, *70*
conformal, 26, *28*, *138*, *141*, *239*
conformal_metrics, 28

data.table::copy(), *101*
data_fingerprint, 29, *30*
data_profile, 30
ddb_collect, 31
ddb_data, *31*, 31
decomp, *15*, 33, *37*, *107*, *129*, *143*, *258*, *262*, *283*
decomp_metrics, 38
decomposition_metric_traits, *35*, 36
decomposition_metric_traits(), *10*
decomposition_metrics, 34
decomposition_metrics(), *38*, *71*
decomposition_traits, *12*, *14*, *34*, *36*, 37, *69*, *107*
describe, 39
df_movecolumn, 40
df_nunique_perfeat, 41
draw_linad, 41
dt_describe, 44
dt_inspect_types, 45
dt_keybin_reshape, 46
dt_merge, 47
dt_names_by_attr, 48
dt_nunique_perfeat, 49
dt_pctmatch, 50
dt_pctmissing, 51
dt_set_autotypes, 51
dt_set_clean_all, 53
dt_set_cleanfactorlevels, 52
dt_set_logical2factor, 54
dt_set_one_hot, 55

e1071::svm, *198*, *226*
earth::earth, *200*
exc, 56
explain, 57, *59*
explanation_methods, 59
explanation_traits, *57*, 59
explanation_traits(), *59*

feature_matrix, 61

feature_names, 62
features, 60
features(), *61*, *89*
fit_predict, 63
fitted_config, 62

get_factor_names, 66
get_fitted_config, 67
get_fitted_config(), *62, 63*
get_learning_curve, 68
get_loadings, *37*, 69
get_membership, 70
get_membership(), *25*
get_metrics, 71
get_metrics(), *25*, *73*
get_mode, 72
get_predictions, 73
get_varimp, 74, *98*, *138*, *278*
getcharacternames *(*getnames*)*, 64
getdatenames *(*getnames*)*, 64
getfactornames *(*getnames*)*, 64
getfactornames(), *66*
getlogicalnames *(*getnames*)*, 64
getnames, 64
getnamesandtypes, 65
getnumericnames *(*getnames*)*, 64
glmnet::glmnet, *150*

hal9001::fit_hal, *157*, *210*
HyperparameterCandidates, 75

inc, 75
index_col_by_attr, 76
ingest, 77
init_project_dir, 78
inspect, 79
inspect_type, *45*, 79
is_constant, 80
is_same_data, 81
is_same_data(), *123*

JSONSchema_to_S7, 82

kknn::train.kknn, *165*

lightgbm::lgb.train, *170*, *176*, *181*, *187*

massGLM, 83
matchcases, 84
mgcv::gam, *148*
mgetnames, 86
mode_value, 87
mode_value(), *72*

names_by_class, 88
numeric_features, 88

one_hot2factor, 89, *285*
outcome, 90
outcome_name, 91

partition, 91
plot.MassGLM *(*plot_manhattan*)*, 93
plot_learning, *68*, 92
plot_manhattan, 93
plot_metric, 94
plot_metric(), *102*
plot_roc, 95
plot_roc(), *101*, *115*
plot_session, 96
plot_true_pred, 97
plot_true_pred(), *102*
plot_varimp, 98
plot_varimp(), *277*
preprocess, *12, 13*, 99, *216*, *220*, *222, 223*, *251*, *253*
preprocess(), *61*, *101*
preprocessed, 101
present, 101

ranger::ranger, *230*
read, 103
read_config, 105, *281*, *283*
read_result, 106
reconstruct, 107, *129*, *202*, *219*, *262*
record, 108, *282*
regression_metrics, 109
resample, 109
review, *8*, 111, *279*, *284*
rnormmat, 113
roc_curve, 114
roc_curve(), *115*
roc_table, 115
rpart::rpart, *134*
rpart::rpart.control, *134*
rtemis *(*rtemis-package*)*, 7
rtemis-deprecated *(*setup_Resampler*)*, 232
rtemis-package, 7, *117*
rtemis.draw::ConfusionConfig, *97*

rtemis.draw::draw_confusion(), *97*
rtemis.draw::draw_fit(), *64*, *97*
rtemis.draw::draw_gantt(), *96*
rtemis.draw::draw_learning_curve(), *92*
rtemis.draw::draw_manhattan(), *93*
rtemis.draw::draw_metric(), *94*
rtemis.draw::draw_panels(), *102*
rtemis.draw::draw_roc(), *95*
rtemis.draw::draw_scatter(), *64*
rtemis.draw::draw_scatter3d(), *64*
rtemis.draw::draw_varimp(), *98*
rtemis.draw::draw_volcano(), *93*
rtemis.draw::SignificanceConfig, *93*
rtemis.draw::summarize_varimp(), *277*
rtemis.draw::Theme, *43*, *97*
rtemis.draw::theme_light(), *43*
rtemis_conditions, *8*, 116
rtemis_data_error *(*rtemis_conditions*)*, 116
rtemis_dim_error *(*rtemis_conditions*)*, 116
rtemis_error *(*rtemis_conditions*)*, 116
rtemis_file_exists *(*rtemis_conditions*)*, 116
rtemis_file_not_found *(*rtemis_conditions*)*, 116
rtemis_input_error *(*rtemis_conditions*)*, 116
rtemis_io_error *(*rtemis_conditions*)*, 116
rtemis_length_error *(*rtemis_conditions*)*, 116
rtemis_level_mismatch *(*rtemis_conditions*)*, 116
rtemis_missing_data *(*rtemis_conditions*)*, 116
rtemis_null_input *(*rtemis_conditions*)*, 116
rtemis_outcome_error *(*rtemis_conditions*)*, 116
rtemis_range_error *(*rtemis_conditions*)*, 116
rtemis_runtime_error *(*rtemis_conditions*)*, 116
rtemis_type_error *(*rtemis_conditions*)*, 116
rtemis_unsupported_error *(*rtemis_conditions*)*, 116
rtemis_value_error *(*rtemis_conditions*)*, 116
rtversion, 117
runifmat, 118

S7_dispatcher_JSONSchema, 119, *122*
S7_to_JSONSchema, *82*, *120*, 121
same_data, 123
se, 123
session_timeline, 124
set_outcome, *7*, 263
set_positive_class, *247*, 264
setdiffsym, 125
setup_ARFFIngest, 126
setup_ASWCriterion, 127, *216*
setup_Autoencoder, *34*, 127, *262*
setup_BART, 130
setup_Bootstrap, 132
setup_Bootstrap(), *232*
setup_CART, 133, *138*
setup_CHCriterion, 134, *216*
setup_ClusterConfig, *23*, 135, *281*
setup_CMeans, 136
setup_ConditionalSuperLearner, 137
setup_CQR, *26*, *27*, 138
setup_CUDA, 139, *147*, *203*, *233*
setup_Custom, 140
setup_Custom(), *232*
setup_CVPlus, *26*, *27*, 141
setup_DBSCAN, 142
setup_DecomposeConfig, *33*, 143, *281*
setup_DelimitedIngest, 144
setup_DTAIngest, 145
setup_FutureExecution, *18*, *24*, *34*, *135*, *143*, 146, *203*, *206*, *234*, *245*, *271*, *273*
setup_GAM, 148
setup_GLM, 149
setup_GLMNET, 149
setup_GLMTree, 150
setup_GMM, 153
setup_GridSearch, 154, *271*
setup_GroupPartition, 155
setup_HAL, 156
setup_HardCL, 158
setup_HOPACH, 159
setup_ICA, 161
setup_Isomap, 162
setup_Isotonic, *18*, *19*, 163, *210*
setup_KFold, *110*, *155*, 163, *271*

setup_KFold(), *232*
setup_KMeans, *135*, 164
setup_KNN, 165
setup_LightCART, 166
setup_LightGBM, 170, *181*
setup_LightRF, 176
setup_LightRuleFit, 181
setup_LINAD, 187, *196*
setup_LINAD(), *63*
setup_LINADForest, 193
setup_LinearSVM, 198
setup_LOOCV, 198
setup_LOOCV(), *232*
setup_MARS, 199
setup_MDS, 201
setup_MiraiExecution, *18*, 202, *245*, *271*
setup_MLP, *129*, 204
setup_ModalityStacking, 207, *213*
setup_MonotonicHAL, *19*, 209
setup_MultiASWCriterion, 211, *216*
setup_NeuralGas, 211
setup_NMF, 212
setup_NNLS, 213
setup_Nystrom, 214, *236*, *238*
setup_PAM, 214
setup_PAMK, *127*, *134*, *211*, 216
setup_ParquetIngest, 217
setup_PCA, *143*, 218
setup_PCoA, *201*, 219
setup_PredefinedPartition, 220
setup_Preprocessor, *13*, *67*, *99*, *100*, 220, *249*, *253*, *281*
setup_RadialSVM, 226
setup_RandomPartition, 227
setup_Ranger, 227
setup_RDSIngest, 231
setup_Resampler, 232
setup_SerialExecution, *18*, *24*, *34*, 233, *245*, *271*
setup_SHAP, *57*, 234
setup_SpectralLaplace, *214*, 236, *239*
setup_SpectralLocal, 237, *239*
setup_SpectralRBF, *214*, *236*, 238
setup_SplitConformal, *26*, *27*, *141*, 239
setup_SPLS, 241
setup_StratBoot, 242
setup_StratBoot(), *232*
setup_StratSub, 243
setup_StratSub(), *232*
setup_SuperConfig, 244, *246*, *247*, *281*
setup_SuperConfigLive, 246
setup_SuperLearner, *208*, *213*, 248
setup_SupervisedPreprocessor, *225*, 249, *271*, *272*
setup_TabNet, 254
setup_TimePartition, 256
setup_tSNE, 257
setup_UMAP, 259
setup_VariationalAutoencoder, 260
setup_XLSXIngest, 262
shap_by_level, *58*, 265
shap_case, *58*, 266
shap_long, *58*, 267
show_color_key, 268
size, 268
spls::spls, *242*
spls::splsda, *242*
stats::fitted(), *73*
stats::predict(), *26*, *57*, *58*, *253*
stochtree::bart, *131*

table_column_attr, 269
train, *15*, *71*, *73*, *111*, *132*, *154*, *164*, *176*, *181*, *191*, *202*, *206–208*, *219*, *222*, *224*, *225*, *230*, *232*, *243–245*, *247*, *249*, *250*, *253*, *256*, 270, *279*, *283*
train(), *64*, *124*
transformed, 273
transformed(), *12*
tryCatch(), *8*, *116*
tune_over, 274
tune_over(), *75*
tuning_grid, 275

utils::download.file(), *104*

validate_config, *30*, *271*, 276
varimp_ranking, 277
varimp_ranking(), *98*
varimp_table, *74*, 278

withCallingHandlers(), *116*
write_config, *105*, *121*, 281, *282*
write_JSONSchema, *120*, *122*
write_record, *24*, *34*, *105*, *271*, *281*, 282
write_result, 283, *284*
write_text, *9*, *112*, *280*, 284
writeup, 279, *284*