

# Package ‘daltoolbox’

September 2, 2026

**Title** Leveraging Experiment Lines to Data Analytics

**Version** 1.3.787

## Description

The natural increase in the complexity of current research experiments and data demands better tools to enhance productivity in Data Analytics. The package is a framework designed to address the modern challenges in data analytics workflows. The package is inspired by Experiment Line concepts. It aims to provide seamless support for users in developing their data mining workflows by offering a uniform data model and method API. It enables the integration of various data mining activities, including data preprocessing, classification, regression, clustering, and time series prediction. It also offers options for hyper-parameter tuning and supports integration with existing libraries and languages. Overall, the package provides researchers with a comprehensive set of functionalities for data science, promoting ease of use, extensibility, and integration with various tools and libraries. Information on Experiment Line is based on Ogasawara et al. (2009) <[doi:10.1007/978-3-642-02279-1\\_20](https://doi.org/10.1007/978-3-642-02279-1_20)>.

**License** MIT + file LICENSE

**URL** <https://cefet-rj-dal.github.io/daltoolbox/>,  
<https://github.com/cefet-rj-dal/daltoolbox>

**BugReports** <https://github.com/cefet-rj-dal/daltoolbox/issues>

**Encoding** UTF-8

**Depends** R (>= 4.1.0)

**Imports** methods, FNN, caret, arules, arulesSequences, class, cluster,  
dbscan, dplyr, e1071, ggplot2, mclust, nnet, randomForest,  
reshape, tree

**Suggests** adabag, glmnet, leaps, ipred, xgboost, arulesViz, GGally,  
igraph, rpart, MASS

**Config/roxygen2/version** 8.0.0

**NeedsCompilation** no

**Author** Eduardo Ogasawara [aut, ths, cre] (ORCID:  
<<https://orcid.org/0000-0002-0466-0626>>),  
Ana Carolina Sa [aut],  
Antonio Castro [aut],

Caio Santos [aut],  
 Diego Carvalho [ctb],  
 Diego Salles [aut],  
 Eduardo Bezerra [ctb],  
 Esther Pacitti [ctb],  
 Fabio Porto [ctb],  
 Janio Lima [aut],  
 Lucas Tavares [aut],  
 Rafaelli Coutinho [ctb],  
 Rebecca Salles [aut],  
 Vinicius Saidy [aut],  
 CEFET/RJ [cph]

**Maintainer** Eduardo Ogasawara <eogasawara@ieee.org>

**Repository** CRAN

**Date/Publication** 2026-09-02 15:00:16 UTC

## Contents

|                                |    |
|--------------------------------|----|
| action . . . . .               | 5  |
| action.dal_transform . . . . . | 6  |
| adjust_class_label . . . . .   | 6  |
| adjust_data.frame . . . . .    | 7  |
| adjust_factor . . . . .        | 7  |
| adjust_matrix . . . . .        | 8  |
| aggregation . . . . .          | 8  |
| autoenc_base_e . . . . .       | 9  |
| autoenc_base_ed . . . . .      | 10 |
| bal_oversampling . . . . .     | 11 |
| bal_subsampling . . . . .      | 11 |
| Boston . . . . .               | 12 |
| categ_mapping . . . . .        | 13 |
| classification . . . . .       | 14 |
| cla_bagging . . . . .          | 14 |
| cla_boosting . . . . .         | 15 |
| cla_dtree . . . . .            | 16 |
| cla_glm . . . . .              | 17 |
| cla_glmnet . . . . .           | 17 |
| cla_knn . . . . .              | 18 |
| cla_majority . . . . .         | 19 |
| cla_mlp . . . . .              | 20 |
| cla_multinom . . . . .         | 21 |
| cla_nb . . . . .               | 22 |
| cla_rf . . . . .               | 23 |
| cla_rpart . . . . .            | 24 |
| cla_svm . . . . .              | 25 |
| cla_tune . . . . .             | 26 |
| cla_xgboost . . . . .          | 27 |

|                                       |    |
|---------------------------------------|----|
| cluster . . . . .                     | 28 |
| clusterer . . . . .                   | 28 |
| cluster_cmeans . . . . .              | 29 |
| cluster_dbscan . . . . .              | 30 |
| cluster_gmm . . . . .                 | 32 |
| cluster_hclust . . . . .              | 33 |
| cluster_kmeans . . . . .              | 34 |
| cluster_louvain_graph . . . . .       | 35 |
| cluster_pam . . . . .                 | 36 |
| cluutils . . . . .                    | 38 |
| clu_tune . . . . .                    | 39 |
| dal_base . . . . .                    | 40 |
| dal_graphics . . . . .                | 40 |
| dal_learner . . . . .                 | 41 |
| dal_transform . . . . .               | 41 |
| dal_tune . . . . .                    | 42 |
| data_sample . . . . .                 | 43 |
| discover . . . . .                    | 43 |
| dt_pca . . . . .                      | 44 |
| evaluate . . . . .                    | 45 |
| feature_generation . . . . .          | 46 |
| feature_selection_corr . . . . .      | 46 |
| feature_selection_fss . . . . .       | 47 |
| feature_selection_info_gain . . . . . | 48 |
| feature_selection_lasso . . . . .     | 49 |
| feature_selection_relief . . . . .    | 50 |
| feature_selection_stepwise . . . . .  | 51 |
| fit . . . . .                         | 52 |
| fit.cla_tune . . . . .                | 52 |
| fit.cluster_dbscan . . . . .          | 53 |
| fit_curvature_max . . . . .           | 53 |
| fit_curvature_min . . . . .           | 54 |
| hierarchy_cut . . . . .               | 55 |
| imputation_predictive . . . . .       | 55 |
| imputation_simple . . . . .           | 56 |
| imputation_tree . . . . .             | 57 |
| inverse_transform . . . . .           | 58 |
| k_fold . . . . .                      | 59 |
| minmax . . . . .                      | 59 |
| na_removal . . . . .                  | 60 |
| outliers_boxplot . . . . .            | 61 |
| outliers_gaussian . . . . .           | 62 |
| pattern_miner . . . . .               | 63 |
| patutils . . . . .                    | 63 |
| pat_apriori . . . . .                 | 64 |
| pat_confidence_threshold . . . . .    | 66 |
| pat_cspade . . . . .                  | 67 |
| pat_dara . . . . .                    | 68 |

|                                    |     |
|------------------------------------|-----|
| pat_dara_rules . . . . .           | 69  |
| pat_eclat . . . . .                | 70  |
| pat_filter_rules . . . . .         | 71  |
| pat_rules_tidy . . . . .           | 72  |
| pat_rule_filter_dara . . . . .     | 73  |
| pat_rule_filter_interest . . . . . | 74  |
| pat_rule_filter_none . . . . .     | 75  |
| pat_support_threshold . . . . .    | 75  |
| plot_bar . . . . .                 | 76  |
| plot_boxplot . . . . .             | 77  |
| plot_boxplot_class . . . . .       | 78  |
| plot_correlation . . . . .         | 79  |
| plot_dendrogram . . . . .          | 80  |
| plot_density . . . . .             | 81  |
| plot_density_class . . . . .       | 82  |
| plot_groupedbar . . . . .          | 83  |
| plot_hist . . . . .                | 84  |
| plot_lollipop . . . . .            | 84  |
| plot_pair . . . . .                | 86  |
| plot_pair_adv . . . . .            | 86  |
| plot_parallel . . . . .            | 87  |
| plot_pieplot . . . . .             | 88  |
| plot_pixel . . . . .               | 89  |
| plot_points . . . . .              | 90  |
| plot_radar . . . . .               | 91  |
| plot_scatter . . . . .             | 92  |
| plot_series . . . . .              | 92  |
| plot_stackedbar . . . . .          | 93  |
| plot_ts . . . . .                  | 94  |
| plot_ts_pred . . . . .             | 95  |
| predictor . . . . .                | 96  |
| predictor_prepare_fit . . . . .    | 96  |
| regression . . . . .               | 97  |
| reg_dtree . . . . .                | 97  |
| reg_knn . . . . .                  | 98  |
| reg_lm . . . . .                   | 99  |
| reg_mlp . . . . .                  | 100 |
| reg_rf . . . . .                   | 101 |
| reg_svm . . . . .                  | 102 |
| reg_tune . . . . .                 | 103 |
| sample_balance . . . . .           | 104 |
| sample_groups . . . . .            | 105 |
| sample_random . . . . .            | 106 |
| sample_simple . . . . .            | 107 |
| sample_stratified . . . . .        | 107 |
| select_hyper . . . . .             | 108 |
| select_hyper.cla_tune . . . . .    | 109 |
| set_params . . . . .               | 109 |

|                                  |            |
|----------------------------------|------------|
| set_params.default . . . . .     | 110        |
| smoothing . . . . .              | 110        |
| smoothing_cluster . . . . .      | 111        |
| smoothing_freq . . . . .         | 112        |
| smoothing_inter . . . . .        | 113        |
| smoothing_quantization . . . . . | 113        |
| train_test . . . . .             | 114        |
| train_test_from_folds . . . . .  | 115        |
| transform . . . . .              | 116        |
| zscore . . . . .                 | 117        |
| <b>Index</b>                     | <b>118</b> |

---

|        |               |
|--------|---------------|
| action | <i>Action</i> |
|--------|---------------|

---

**Description**

Generic to apply the object to data (e.g., predict, transform).

**Usage**

action(obj, ...)

**Arguments**

- obj                    object: a dal\_base object to apply the transformation on the input dataset.
- ...                   optional arguments.

**Value**

returns the result of an action of the model applied in provided data

**Examples**

```
data(iris)
# an example is minmax normalization
trans <- minmax()
trans <- fit(trans, iris)
tiris <- action(trans, iris)
```

---

`action.dal_transform`    *Action implementation for transform*

---

### Description

Default `action()` implementation that proxies to `transform()` for transforms.

### Usage

```
## S3 method for class 'dal_transform'
action(obj, ...)
```

### Arguments

|                  |                    |
|------------------|--------------------|
| <code>obj</code> | object             |
| <code>...</code> | optional arguments |

### Value

returns a transformed data

### Examples

```
#See ?minmax for an example of transformation
```

---

`adjust_class_label`    *Adjust categorical mapping*

---

### Description

One-hot encode a factor vector into a matrix of indicator columns.

### Usage

```
adjust_class_label(x, valTrue = 1, valFalse = 0)
```

### Arguments

|                       |                          |
|-----------------------|--------------------------|
| <code>x</code>        | vector to be categorized |
| <code>valTrue</code>  | value to represent true  |
| <code>valFalse</code> | value to represent false |

### Details

Values are mapped to `valTrue`/`valFalse` (default 1/0). The resulting matrix has column names equal to `levels(x)`.

**Value**

returns an adjusted categorical mapping

---

|                   |                             |
|-------------------|-----------------------------|
| adjust_data.frame | <i>Adjust to data frame</i> |
|-------------------|-----------------------------|

---

**Description**

Coerce an object to `data.frame` if needed (useful for S3 methods in this package).

**Usage**

```
adjust_data.frame(data)
```

**Arguments**

|      |         |
|------|---------|
| data | dataset |
|------|---------|

**Value**

returns a `data.frame`

**Examples**

```
data(iris)
df <- adjust_data.frame(iris)
```

---

|               |                       |
|---------------|-----------------------|
| adjust_factor | <i>Adjust factors</i> |
|---------------|-----------------------|

---

**Description**

Convert a vector to a factor with specified internal levels (`ilevels`) and labels (`slevels`).

**Usage**

```
adjust_factor(value, ilevels, slevels)
```

**Arguments**

|         |                                    |
|---------|------------------------------------|
| value   | vector to be converted into factor |
| ilevels | order for categorical values       |
| slevels | labels for categorical values      |

**Details**

Numeric vectors are first converted to factors with `ilevels` as the level order, then relabeled to `slevels`.

**Value**

returns an adjusted factor

---

|                            |                         |
|----------------------------|-------------------------|
| <code>adjust_matrix</code> | <i>Adjust to matrix</i> |
|----------------------------|-------------------------|

---

**Description**

Coerce an object to `matrix` if needed (useful before algorithms that expect matrices).

**Usage**

```
adjust_matrix(data)
```

**Arguments**

|                   |         |
|-------------------|---------|
| <code>data</code> | dataset |
|-------------------|---------|

**Value**

returns an adjusted matrix

**Examples**

```
data(iris)
mat <- adjust_matrix(iris)
```

---

|                          |                              |
|--------------------------|------------------------------|
| <code>aggregation</code> | <i>Aggregation by groups</i> |
|--------------------------|------------------------------|

---

**Description**

Aggregate data by a grouping attribute using named expressions.

**Usage**

```
aggregation(group, ...)
```

**Arguments**

|                    |                                       |
|--------------------|---------------------------------------|
| <code>group</code> | grouping column name (string)         |
| <code>...</code>   | named expressions evaluated per group |



**Value**

returns an object of class aggregation

**Examples**

```
data(iris)
agg <- aggregation(
  "Species",
  mean_sepal = mean(Sepal.Length),
  sd_sepal = sd(Sepal.Length),
  n = n()
)
iris_agg <- transform(agg, iris)
iris_agg
```

---

|                |                                   |
|----------------|-----------------------------------|
| autoenc_base_e | <i>Autoencoder base (encoder)</i> |
|----------------|-----------------------------------|

---

**Description**

Base class for encoder-only autoencoders. Intended to be subclassed by concrete implementations that learn a lower-dimensional latent representation.

**Usage**

```
autoenc_base_e(input_size, encoding_size)
```

**Arguments**

input\_size      dimensionality of the input vector  
 encoding\_size   dimensionality of the latent (encoded) vector

**Details**

This base does not train or transform by itself (identity). Implementations should override `fit()` to learn parameters and `transform()` to output the encoded representation.

**Value**

returns an autoenc\_base\_e object

**References**

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the Dimensionality of Data with Neural Networks. Science.

## Examples

```
# See an end-to-end example at:  
# https://github.com/cefet-rj-dal/daltoolbox/blob/main/autoencoder/autoenc\_base\_e.md
```

---

|                 |                                             |
|-----------------|---------------------------------------------|
| autoenc_base_ed | <i>Autoencoder base (encoder + decoder)</i> |
|-----------------|---------------------------------------------|

---

## Description

Base class for autoencoders that both encode and decode. Intended to be subclassed by concrete implementations that learn to compress and reconstruct inputs.

## Usage

```
autoenc_base_ed(input_size, encoding_size)
```

## Arguments

|               |                                               |
|---------------|-----------------------------------------------|
| input_size    | dimensionality of the input vector            |
| encoding_size | dimensionality of the latent (encoded) vector |

## Details

This base does not train or transform by itself (identity). Implementations should override `fit()` to learn parameters and `transform()` to perform encode+decode.

## Value

returns an `autoenc_base_ed` object

## References

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the Dimensionality of Data with Neural Networks. Science.

## Examples

```
# See an end-to-end example at:  
# https://github.com/cefet-rj-dal/daltoolbox/blob/main/autoencoder/autoenc\_base\_ed.md
```

---

|                  |                                                 |
|------------------|-------------------------------------------------|
| bal_oversampling | <i>Random or SMOTE-based class oversampling</i> |
|------------------|-------------------------------------------------|

---

**Description**

Balance class distributions by randomly replicating minority examples or by generating synthetic samples with a local SMOTE implementation.

**Usage**

```
bal_oversampling(attribute, method = c("smote", "random"), k = 5)
```

**Arguments**

|           |                                                        |
|-----------|--------------------------------------------------------|
| attribute | target class attribute name                            |
| method    | oversampling strategy: "smote" or "random"             |
| k         | number of nearest neighbors used by the SMOTE strategy |

**Value**

returns an object of class bal\_oversampling

**References**

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique.

**Examples**

```
data(iris)
iris_imb <- iris[c(1:50, 51:71, 101:111), ]
bal <- bal_oversampling("Species", method = "smote")
iris_bal <- transform(bal, iris_imb)
table(iris_bal$Species)
```

---

|                 |                                   |
|-----------------|-----------------------------------|
| bal_subsampling | <i>Random class undersampling</i> |
|-----------------|-----------------------------------|

---

**Description**

Balance class distributions by randomly reducing all classes to the minority count.

**Usage**

```
bal_subsampling(attribute)
```

**Arguments**

attribute            target class attribute name

**Value**

returns an object of class `bal_subsampling`

**Examples**

```
data(iris)
iris_imb <- iris[c(1:50, 51:71, 101:111), ]
bal <- bal_subsampling("Species")
iris_bal <- transform(bal, iris_imb)
table(iris_bal$Species)
```

---

Boston

*Boston Housing Data (Regression)*

---

**Description**

housing values in suburbs of Boston.

- crim: per capita crime rate by town.
- zn: proportion of residential land zoned for lots over 25,000 sq.ft.
- indus: proportion of non-retail business acres per town
- chas: Charles River dummy variable (= 1 if tract bounds)
- nox: nitric oxides concentration (parts per 10 million)
- rm: average number of rooms per dwelling
- age: proportion of owner-occupied units built prior to 1940
- dis: weighted distances to five Boston employment centres
- rad: index of accessibility to radial highways
- tax: full-value property-tax rate per \$10,000
- ptratio: pupil-teacher ratio by town
- black:  $1000(B_k - 0.63)^2$  where  $B_k$  is the proportion of blacks by town
- lstat: percentage of lower status of the population
- medv: Median value of owner-occupied homes in \$1000's

**Usage**

```
data(Boston)
```

**Format**

Regression Dataset.

**Source**

This dataset was obtained from the MASS library.

**References**

Creator: Harrison, D. and Rubinfeld, D.L. Hedonic prices and the demand for clean air, J. Environ. Economics & Management, vol.5, 81-102, 1978.

**Examples**

```
data(Boston)
head(Boston)
```

---

|               |                                               |
|---------------|-----------------------------------------------|
| categ_mapping | <i>Categorical mapping (one-hot encoding)</i> |
|---------------|-----------------------------------------------|

---

**Description**

Convert a factor column into dummy variables (one-hot encoding) using `model.matrix` without intercept. Each level becomes a separate binary column.

**Usage**

```
categ_mapping(attribute)
```

**Arguments**

attribute      attribute to be categorized.

**Details**

This is a light wrapper around `stats::model.matrix(~ attr - 1, data)` that drops the original column and returns only the dummy variables.

**Value**

returns a data frame with binary attributes, one for each possible category.

**Examples**

```
cm <- categ_mapping("Species")
iris_cm <- transform(cm, iris)

# can be made in a single column
species <- iris[, "Species", drop=FALSE]
iris_cm <- transform(cm, species)
```

---

|                |                                  |
|----------------|----------------------------------|
| classification | <i>Classification base class</i> |
|----------------|----------------------------------|

---

**Description**

Ancestor class for classification models providing common fields (target attribute and levels) and evaluation helpers.

**Usage**

```
classification(attribute, slevels = NULL)
```

**Arguments**

|           |                                               |
|-----------|-----------------------------------------------|
| attribute | attribute target to model building            |
| slevels   | possible values for the target classification |

**Value**

returns a classification object

**Examples**

```
#See ?cla_dtree for a classification example using a decision tree
```

---

|             |                        |
|-------------|------------------------|
| cla_bagging | <i>Bagging (ipred)</i> |
|-------------|------------------------|

---

**Description**

Bagging classifier using ipred::bagging.

**Usage**

```
cla_bagging(attribute, nbagg = 25)
```

**Arguments**

|           |                                  |
|-----------|----------------------------------|
| attribute | target attribute name            |
| nbagg     | number of bootstrap aggregations |

**Value**

returns a cla\_bagging object

**Examples**

```

if (requireNamespace("ipred", quietly = TRUE)) {
  data(iris)
  model <- cla_bagging("Species", nbagg = 25)
  model <- fit(model, iris)
  pred <- predict(model, iris)
  eval <- evaluate(model, adjust_class_label(iris$Species), pred)
  eval$metrics
}

```

cla\_boosting

*Boosting (adabag)***Description**

Boosting classifier using `adabag::boosting`.

**Usage**

```
cla_boosting(attribute, mfinal = 50)
```

**Arguments**

|                        |                               |
|------------------------|-------------------------------|
| <code>attribute</code> | target attribute name         |
| <code>mfinal</code>    | number of boosting iterations |

**Value**

returns a `cla_boosting` object

**Examples**

```

if (requireNamespace("adabag", quietly = TRUE)) {
  data(iris)
  model <- cla_boosting("Species", mfinal = 10)
  model <- fit(model, iris)
  pred <- predict(model, iris)
  eval <- evaluate(model, adjust_class_label(iris$Species), pred)
  eval$metrics
}

```

---

`cla_dtree`*Decision Tree for classification*

---

**Description**

Univariate decision tree for classification using recursive partitioning. This wrapper uses the `tree` package.

**Usage**

```
cla_dtree(attribute, slevels)
```

**Arguments**

|                        |                                                   |
|------------------------|---------------------------------------------------|
| <code>attribute</code> | attribute target to model building                |
| <code>slevels</code>   | the possible values for the target classification |

**Details**

Decision trees split the feature space by maximizing node purity (e.g., Gini/entropy), yielding a human-readable set of rules. They are fast and interpretable, and often used as base learners in ensembles.

**Value**

returns a classification object

**References**

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and Regression Trees*. Wadsworth.

**Examples**

```
data(iris)
slevels <- levels(iris$Species)
model <- cla_dtree("Species", slevels)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, iris)
train <- sr$train
test <- sr$test

model <- fit(model, train)

prediction <- predict(model, test)
predictand <- adjust_class_label(test[, "Species"])
test_eval <- evaluate(model, predictand, prediction)
test_eval$metrics
```



---

|         |                                  |
|---------|----------------------------------|
| cla_glm | <i>Logistic regression (GLM)</i> |
|---------|----------------------------------|

---

**Description**

Logistic regression classifier using `stats::glm` with binomial family.

**Usage**

```
cla_glm(attribute, positive, features = NULL, threshold = 0.5)
```

**Arguments**

|           |                                                                  |
|-----------|------------------------------------------------------------------|
| attribute | target attribute name                                            |
| positive  | positive class label                                             |
| features  | optional vector of feature names (default: all except attribute) |
| threshold | probability threshold for positive class                         |

**Value**

returns a `cla_glm` object

**Examples**

```
data(iris)
iris_bin <- iris
iris_bin$IsVersicolor <- factor(ifelse(
  iris_bin$Species == "versicolor",
  "versicolor",
  "not_versicolor"
))
model <- cla_glm("IsVersicolor", positive = "versicolor")
model <- suppressWarnings(fit(model, iris_bin))
pred <- predict(model, iris_bin)
eval <- evaluate(model, adjust_class_label(iris_bin$IsVersicolor), pred)
eval$metrics
```

---

|            |                                           |
|------------|-------------------------------------------|
| cla_glmnet | <i>LASSO logistic regression (glmnet)</i> |
|------------|-------------------------------------------|

---

**Description**

Logistic regression with L1 penalty using `glmnet::cv.glmnet`.

**Usage**

```
cla_glmnet(attribute, lambda = c("lambda.min", "lambda.1se"))
```

**Arguments**

|           |                                                    |
|-----------|----------------------------------------------------|
| attribute | target attribute name (binary)                     |
| lambda    | which lambda to use ("lambda.min" or "lambda.1se") |

**Value**

returns a cla\_glmnet object

**Examples**

```
if (requireNamespace("glmnet", quietly = TRUE)) {
  data(iris)
  iris_bin <- iris
  iris_bin$IsVersicolor <- factor(ifelse(
    iris_bin$Species == "versicolor",
    "versicolor",
    "not_versicolor"
  ))
  model <- cla_glmnet("IsVersicolor")
  model <- fit(model, iris_bin)
  pred <- predict(model, iris_bin)
  eval <- evaluate(model, adjust_class_label(iris_bin$IsVersicolor), pred)
  eval$metrics
}
```

cla\_knn

*K-Nearest Neighbors (KNN) Classification***Description**

Classification by majority vote among the k nearest neighbors. Uses `class::knn`.

**Usage**

```
cla_knn(attribute, slevels, k = 1)
```

**Arguments**

|           |                                                                           |
|-----------|---------------------------------------------------------------------------|
| attribute | attribute target to model building.                                       |
| slevels   | possible values for the target classification.                            |
| k         | a vector of integers indicating the number of neighbors to be considered. |

**Details**

KNN is a simple, non-parametric method. Choice of k trades bias/variance; distance metric is Euclidean by default.

**Value**

returns a knn object.

**References**

Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. IEEE Trans. Info. Theory.

**Examples**

```
data(iris)
slevels <- levels(iris$Species)
model <- cla_knn("Species", slevels, k=3)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, iris)
train <- sr$train
test <- sr$test

model <- fit(model, train)

prediction <- predict(model, test)
predictand <- adjust_class_label(test[, "Species"])
test_eval <- evaluate(model, predictand, prediction)
test_eval$metrics
```

---

cla\_majority

---

*Majority Baseline Classifier*


---

**Description**

Trivial classifier that always predicts the most frequent class observed in the training data.

**Usage**

```
cla_majority(attribute, slevels)
```

**Arguments**

|           |                                                              |
|-----------|--------------------------------------------------------------|
| attribute | Name of the target attribute to predict.                     |
| slevels   | Possible class labels for the target classification problem. |

**Details**

Useful as a minimum reference point. If a stronger classifier does not outperform this learner, it is often worth revisiting the predictors, the sampling strategy, or the evaluation protocol before tuning models.

**Value**

A classification object of class `cla_majority`.

**References**

Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. (2016). Data Mining: Practical Machine Learning Tools and Techniques (4th ed.). Morgan Kaufmann.

**Examples**

```
data(iris)
slevels <- levels(iris$Species)
model <- cla_majority("Species", slevels)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, iris)
train <- sr$train
test <- sr$test

model <- fit(model, train)

prediction <- predict(model, test)
predictand <- adjust_class_label(test[, "Species"])
test_eval <- evaluate(model, predictand, prediction)
test_eval$metrics
```

---

cla\_mlp

*MLP for classification*

---

**Description**

Multi-Layer Perceptron classifier using `nnet::nnet` (single hidden layer).

**Usage**

```
cla_mlp(attribute, slevels, size = NULL, decay = 0.1, maxit = 1000)
```

**Arguments**

|                        |                                                       |
|------------------------|-------------------------------------------------------|
| <code>attribute</code> | attribute target to model building                    |
| <code>slevels</code>   | possible values for the target classification         |
| <code>size</code>      | number of nodes that will be used in the hidden layer |
| <code>decay</code>     | how quickly it decreases in gradient descent          |
| <code>maxit</code>     | maximum iterations                                    |

**Details**

Uses softmax output with one-hot targets from `adjust_class_label`. `size` controls hidden units and decay applies L2 regularization. Features should be scaled.

**Value**

returns a classification object

**References**

Rumelhart, D., Hinton, G., Williams, R. (1986). Learning representations by back-propagating errors. Bishop, C. M. (1995). Neural Networks for Pattern Recognition.

**Examples**

```
data(iris)
slevels <- levels(iris$Species)
model <- cla_mlp("Species", slevels, size=3, decay=0.03)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, iris)
train <- sr$train
test <- sr$test

model <- fit(model, train)

prediction <- predict(model, test)
predictand <- adjust_class_label(test[, "Species"])
test_eval <- evaluate(model, predictand, prediction)
test_eval$metrics
```

---

cla\_multinom

*Multinomial logistic regression*


---

**Description**

Multiclass classification using `nnet::multinom`.

**Usage**

```
cla_multinom(attribute, features = NULL)
```

**Arguments**

|           |                                                                  |
|-----------|------------------------------------------------------------------|
| attribute | target attribute name                                            |
| features  | optional vector of feature names (default: all except attribute) |

**Value**

returns a cla\_multinom object

**Examples**

```
data(iris)
model <- cla_multinom("Species")
model <- fit(model, iris)
pred <- predict(model, iris)
eval <- evaluate(model, adjust_class_label(iris$Species), pred)
eval$metrics
```

---

cla\_nb

*Naive Bayes Classifier*

---

**Description**

Naive Bayes classification using e1071::naiveBayes.

**Usage**

```
cla_nb(attribute, slevels)
```

**Arguments**

|           |                                                |
|-----------|------------------------------------------------|
| attribute | attribute target to model building.            |
| slevels   | possible values for the target classification. |

**Details**

Assumes conditional independence of features given the class label, enabling fast probabilistic classification.

**Value**

returns a classification object.

**References**

Mitchell, T. (1997). Machine Learning. McGraw-Hill. (Naive Bayes)

**Examples**

```

data(iris)
slevels <- levels(iris$Species)
model <- cla_nb("Species", slevels)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, iris)
train <- sr$train
test <- sr$test

model <- fit(model, train)

prediction <- predict(model, test)
predictand <- adjust_class_label(test[, "Species"])
test_eval <- evaluate(model, predictand, prediction)
test_eval$metrics

```

cla\_rf

*Random Forest for classification***Description**

Ensemble classifier of decision trees using `randomForest::randomForest`.

**Usage**

```
cla_rf(attribute, slevels, nodesize = 5, ntree = 10, mtry = NULL)
```

**Arguments**

|                        |                                               |
|------------------------|-----------------------------------------------|
| <code>attribute</code> | attribute target to model building            |
| <code>slevels</code>   | possible values for the target classification |
| <code>nodesize</code>  | node size                                     |
| <code>ntree</code>     | number of trees                               |
| <code>mtry</code>      | number of attributes to build tree            |

**Details**

Combines many decorrelated trees to reduce variance. Key hyperparameters: `ntree`, `mtry`, `nodesize`.

**Value**

returns a classification object

## References

Breiman, L. (2001). Random Forests. *Machine Learning* 45(1):5–32. Liaw, A. and Wiener, M. (2002). Classification and Regression by randomForest. *R News*.

## Examples

```
data(iris)
slevels <- levels(iris$Species)
model <- cla_rf("Species", sleveles, ntree=5)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, iris)
train <- sr$train
test <- sr$test

model <- fit(model, train)

prediction <- predict(model, test)
predictand <- adjust_class_label(test[, "Species"])
test_eval <- evaluate(model, predictand, prediction)
test_eval$metrics
```

---

cla\_rpart

*CART (rpart)*


---

## Description

Classification tree using `rpart::rpart`.

## Usage

```
cla_rpart(attribute)
```

## Arguments

attribute      target attribute name

## Value

returns a `cla_rpart` object

## Examples

```
if (requireNamespace("rpart", quietly = TRUE)) {
  data(iris)
  model <- cla_rpart("Species")
  model <- fit(model, iris)
  pred <- predict(model, iris)
```



```

eval <- evaluate(model, adjust_class_label(iris$Species), pred)
eval$metrics
}

```

cla\_svm

*SVM for classification***Description**

Support Vector Machines (SVM) for classification using `e1071::svm`.

**Usage**

```

cla_svm(
  attribute,
  slevels,
  epsilon = 0.1,
  cost = 10,
  kernel = c("radial", "linear", "polynomial", "sigmoid")
)

```

**Arguments**

|                        |                                                                                                                   |
|------------------------|-------------------------------------------------------------------------------------------------------------------|
| <code>attribute</code> | attribute target to model building                                                                                |
| <code>slevels</code>   | possible values for the target classification                                                                     |
| <code>epsilon</code>   | parameter that controls the width of the margin around the separating hyperplane                                  |
| <code>cost</code>      | parameter that controls the trade-off between having a wide margin and correctly classifying training data points |
| <code>kernel</code>    | the type of kernel function to be used in the SVM algorithm (linear, radial, polynomial, sigmoid)                 |

**Details**

SVMs find a maximum-margin hyperplane in a transformed feature space defined by a kernel (linear, radial, polynomial, sigmoid). The cost controls the trade-off between margin width and training error; epsilon affects stopping; kernel sets the feature map.

**Value**

returns a SVM classification object

**References**

Cortes, C. and Vapnik, V. (1995). Support-Vector Networks. *Machine Learning* 20(3):273–297.  
 Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines.

**Examples**

```

data(iris)
slevels <- levels(iris$Species)
model <- cla_svm("Species", slevels, epsilon=0.0, cost=20.000)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, iris)
train <- sr$train
test <- sr$test

model <- fit(model, train)

prediction <- predict(model, test)
predictand <- adjust_class_label(test[, "Species"])
test_eval <- evaluate(model, predictand, prediction)
test_eval$metrics

```

cla\_tune

*Classification tuning (k-fold CV)***Description**

Tune hyperparameters of a base classifier via k-fold cross-validation using a chosen metric.

**Usage**

```
cla_tune(base_model, folds = 10, ranges = NULL, metric = "accuracy")
```

**Arguments**

|            |                                            |
|------------|--------------------------------------------|
| base_model | base model for tuning                      |
| folds      | number of folds for cross-validation       |
| ranges     | a list of hyperparameter ranges to explore |
| metric     | metric used to optimize                    |

**Value**

returns a cla\_tune object

**References**

Kohavi, R. (1995). A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection.

**Examples**

```
# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, iris)
train <- sr$train
test <- sr$test

# hyper parameter setup
tune <- cla_tune(cla_mlp("Species", levels(iris$Species)),
  ranges=list(size=c(3:5), decay=c(0.1)))

# hyper parameter optimization
model <- fit(tune, train)

# testing optimization
test_prediction <- predict(model, test)
test_predictand <- adjust_class_label(test[, "Species"])
test_eval <- evaluate(model, test_predictand, test_prediction)
test_eval$metrics
```

cla\_xgboost

*XGBoost***Description**

Gradient boosting classifier using xgboost.

**Usage**

```
cla_xgboost(attribute, params = list(), nrounds = 20)
```

**Arguments**

|           |                            |
|-----------|----------------------------|
| attribute | target attribute name      |
| params    | list of xgboost parameters |
| nrounds   | number of boosting rounds  |

**Value**

returns a cla\_xgboost object

**Examples**

```
## Not run:
data(iris)
model <- cla_xgboost("Species")
model <- fit(model, iris)
pred <- predict(model, iris)
```

```
eval <- evaluate(model, adjust_class_label(iris$Species), pred)
eval$metrics

## End(Not run)
```

---

|         |                |
|---------|----------------|
| cluster | <i>Cluster</i> |
|---------|----------------|

---

**Description**

Generic for clustering methods

**Usage**

```
cluster(obj, ...)
```

**Arguments**

- obj            a clusterer object
- ...           optional arguments

**Value**

clustered data

**Examples**

```
#See ?cluster_kmeans for an example of transformation
```

---

|           |                  |
|-----------|------------------|
| clusterer | <i>Clusterer</i> |
|-----------|------------------|

---

**Description**

Base class for clustering algorithms and related evaluation utilities.

**Usage**

```
clusterer()
```

Details

The object stores shared state and defaults used by clustering methods. Current algorithms may still differ in how much they use this state, but the goal is to standardize future implementations around:

- `fit()` learning and storing model state
- `cluster()` producing labels from a fitted model
- configurable internal/external metrics and selection helpers via `cluutils()`

The base defaults are intentionally generic:

- `metric = metric_wcss()`
- `eval_internal = list(metric_wcss)`
- `eval_external = list(metric_entropy)`

These defaults provide a common contract for simple partition-based clustering, but they are not imposed on all subclasses. Concrete clustering methods can replace them when another metric is more faithful to the algorithm. For example, centroid and medoid methods often prefer separation-based indices such as silhouette, density-based methods may prefer counting noise points, mixture models naturally expose log-likelihood, and graph community detection is better described by modularity.

Therefore, users should treat the `clusterer()` defaults as a shared fallback, not as the final evaluation policy for every `cluster_*` implementation.

Value

returns a `clusterer` object

Examples

#See `?cluster_kmeans` for an example of transformation

---

|                             |                      |
|-----------------------------|----------------------|
| <code>cluster_cmeans</code> | <i>Fuzzy c-means</i> |
|-----------------------------|----------------------|

---

Description

Fuzzy c-means clustering using `e1071::cmeans`.

Usage

`cluster_cmeans(centers = 2, m = 2, iter = 100, dist = "euclidean")`

Arguments

- |                      |                                                      |
|----------------------|------------------------------------------------------|
| <code>centers</code> | number of clusters                                   |
| <code>m</code>       | fuzziness parameter ( $m > 1$ )                      |
| <code>iter</code>    | maximum number of iterations                         |
| <code>dist</code>    | distance method passed to <code>e1071::cmeans</code> |

Details

Produces soft membership for each cluster. The hard assignment is returned by `cluster()`. Membership degrees are returned in the `membership` attribute.

The base `clusterer()` uses `wcss` as a generic default, but `cluster_cmeans()` specializes that choice because fuzzy clustering carries both a hard partition view and a soft membership structure.

Default evaluation in `cluster_cmeans()` is:

- main metric: `metric_silhouette()`
- internal evaluation: `silhouette`, `withinerror`
- external evaluation: `entropy`, `purity`, `adjusted_rand_index`

The generic `wcss` fallback is not used as the main metric because fuzzy c-means is better summarized by a partition quality index plus a criterion that respects the model’s own within-cluster objective.

Value

returns a fuzzy clustering object.

References

Bezdek, J. C. (1981). Pattern Recognition with Fuzzy Objective Function Algorithms. Rousseeuw, P. J. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis.

Examples

```
data(iris)
model <- cluster_cmeans(centers = 3, m = 2)
model <- fit(model, iris[,1:4])
clu <- cluster(model, iris[,1:4])
table(clu)
```

---

|                |               |
|----------------|---------------|
| cluster_dbscan | <i>DBSCAN</i> |
|----------------|---------------|

---

Description

Density-Based Spatial Clustering of Applications with Noise using `dbscan::dbscan`.

Usage

```
cluster_dbscan(minPts = 3, eps = NULL)
```

Arguments

- |                     |                          |
|---------------------|--------------------------|
| <code>minPts</code> | minimum number of points |
| <code>eps</code>    | distance value           |

## Details

Discovers clusters as dense regions separated by sparse areas. Hyperparameters are `eps` (neighborhood radius) and `minPts` (minimum points). If `eps` is missing, it is estimated from the kNN distance curve elbow.

The base `clusterer()` uses `wcss` as a generic default, but `cluster_dbscan()` specializes that choice because DBSCAN does not force a centroid-based partition and can label observations as noise.

Default evaluation in `cluster_dbscan()` is:

- main metric: `metric_noise_points()`
- internal evaluation: `noise_points`
- external evaluation: `entropy`, `purity`, `adjusted_rand_index`

The general `wcss` fallback is not used as the main metric because it ignores the central modeling choice of DBSCAN: distinguishing dense clusters from sparse or noisy observations.

## Value

returns a `dbscan` object

## References

Ester, M., Kriegel, H.-P., Sander, J., Xu, X. (1996). A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. Zhao, Y., & Karypis, G. (2001). Criterion functions for document clustering.

## Examples

```
# setup clustering
model <- cluster_dbscan(minPts = 3)

#load dataset
data(iris)

# build model
model <- fit(model, iris[,1:4])
clu <- cluster(model, iris[,1:4])
table(clu)

# evaluate model using external metric
eval <- evaluate(model, clu, iris$Species)
eval
```

cluster\_gmm

*Gaussian mixture model clustering (GMM)***Description**

Model-based clustering using `mclust::Mclust`.

**Usage**

```
cluster_gmm(G = NULL, modelNames = NULL)
```

**Arguments**

`G` number of mixture components (clusters). If `NULL`, `Mclust` chooses.  
`modelNames` optional character vector of model names passed to `Mclust`.

**Details**

Fits a Gaussian mixture model and returns the MAP classification. The fitted model is stored in `obj$model`. Requires the `mclust` package.

The base `clusterer()` uses `wcss` as a generic default, but `cluster_gmm()` specializes that choice because GMM is a probabilistic model with an explicit likelihood function.

Default evaluation in `cluster_gmm()` is:

- main metric: `metric_loglik()`
- internal evaluation: `loglik`
- external evaluation: `entropy`, `purity`, `adjusted_rand_index`

The general `wcss` fallback is not used as the main metric because it does not represent the fit criterion optimized by the mixture model.

**Value**

returns a GMM clustering object.

**References**

Fraley, C., & Raftery, A. E. (2002). Model-based clustering. *JASA*. Banfield, J. D., & Raftery, A. E. (1993). Model-based Gaussian and non-Gaussian clustering.

**Examples**

```
if (requireNamespace("mclust", quietly = TRUE)) {
  data(iris)
  model <- cluster_gmm(G = 3)
  model <- fit(model, iris[,1:4])
  clu <- cluster(model, iris[,1:4])
  table(clu)
}
```



---

|                |                                |
|----------------|--------------------------------|
| cluster_hclust | <i>Hierarchical clustering</i> |
|----------------|--------------------------------|

---

## Description

Agglomerative hierarchical clustering using `stats::hclust`.

## Usage

```
cluster_hclust(
  k = 2,
  h = NULL,
  method = c("ward.D2", "ward.D", "single", "complete", "average", "mcquitty", "median",
    "centroid"),
  dist = c("euclidean", "maximum", "manhattan", "canberra", "binary", "minkowski"),
  scale = TRUE
)
```

## Arguments

|                     |                                                                           |
|---------------------|---------------------------------------------------------------------------|
| <code>k</code>      | number of clusters to cut the tree (default 2)                            |
| <code>h</code>      | height to cut the tree (optional; if provided, overrides <code>k</code> ) |
| <code>method</code> | linkage method passed to <code>stats::hclust</code> (default "ward.D2")   |
| <code>dist</code>   | distance method passed to <code>stats::dist</code> (default "euclidean")  |
| <code>scale</code>  | logical; whether to scale data before distance (default TRUE)             |

## Details

Computes a distance matrix (optionally after scaling) and builds a dendrogram. Clusters are obtained by cutting the tree with `k` (number of clusters) or `h` (height).

The base `clusterer()` uses `wcss` as a generic default, but `cluster_hclust()` specializes that choice because a hierarchical partition is obtained from a dendrogram and is better judged through separation/cohesion indices than through a centroid-style fallback alone.

Default evaluation in `cluster_hclust()` is:

- main metric: `metric_silhouette()`
- internal evaluation: `silhouette`, `davies_bouldin`, `calinski_harabasz`
- external evaluation: `entropy`, `purity`, `adjusted_rand_index`

The general `wcss` fallback is not kept as the main metric because the method is not fitted by directly minimizing that criterion.

## Value

returns a hierarchical clustering object.

## References

Johnson, S. C. (1967). Hierarchical clustering schemes. *Psychometrika*. Rousseeuw, P. J. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. Davies, D. L., & Bouldin, D. W. (1979). A cluster separation measure. Calinski, T., & Harabasz, J. (1974). A dendrite method for cluster analysis.

## Examples

```
data(iris)
model <- cluster_hclust(k = 3)
model <- fit(model, iris[,1:4])
clu <- cluster(model, iris[,1:4])
table(clu)
```

---

|                |                |
|----------------|----------------|
| cluster_kmeans | <i>k-means</i> |
|----------------|----------------|

---

## Description

k-means clustering using `stats::kmeans`.

## Usage

```
cluster_kmeans(k = 1)
```

## Arguments

`k` the number of clusters to form.

## Details

Partitions data into `k` clusters minimizing within-cluster sum of squares. The intrinsic quality metric returned is the total within-cluster SSE (lower is better).

The base `clusterer()` uses `wcss` as a generic default, but `cluster_kmeans()` specializes that choice because k-means is usually interpreted through compactness and separation between centroid-based groups.

Default evaluation in `cluster_kmeans()` is:

- main metric: `metric_silhouette()`
- internal evaluation: `silhouette`, `davies_bouldin`, `calinski_harabasz`
- external evaluation: `entropy`, `purity`, `adjusted_rand_index`

The general `wcss` fallback is not kept as the main metric here because it decreases mechanically as `k` grows and is therefore weaker as a standalone quality criterion for comparing partitions.

## Value

returns a k-means object.

## References

MacQueen, J. (1967). Some Methods for classification and Analysis of Multivariate Observations. Lloyd, S. (1982). Least squares quantization in PCM. Rousseeuw, P. J. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. Davies, D. L., & Bouldin, D. W. (1979). A cluster separation measure. Calinski, T., & Harabasz, J. (1974). A dendrite method for cluster analysis.

## Examples

```
# setup clustering
model <- cluster_kmeans(k=3)

#load dataset
data(iris)

# build model
model <- fit(model, iris[,1:4])
clu <- cluster(model, iris[,1:4])
table(clu)

# evaluate model using external metric
eval <- evaluate(model, clu, iris$Species)
eval
```

---

cluster\_louvain\_graph *Louvain community detection*

---

## Description

Graph community detection using `igraph::cluster_louvain`.

## Usage

```
cluster_louvain_graph(weights = NULL)
```

## Arguments

`weights` optional edge weights to pass to `cluster_louvain`

## Details

Accepts an `igraph` object and returns community memberships. Requires the `igraph` package.

The base `clusterer()` uses `wcss` and `entropy` as generic defaults, but `cluster_louvain_graph()` specializes both because graph community detection is not a point-to-centroid clustering problem.

Default evaluation in `cluster_louvain_graph()` is:

- main metric: `metric_modularity()`
- internal evaluation: `modularity`

- external evaluation: none

The general wcss fallback is not used because graph communities are judged by network connectivity structure, not by geometric dispersion. The generic external entropy default is also not kept because this class operates on an `igraph` object and has no natural reference labels unless the user supplies an external comparison separately.

## Value

returns a Louvain clustering object.

## References

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *J. Statistical Mechanics*. Newman, M. E. J. (2006). Modularity and community structure in networks.

## Examples

```
if (requireNamespace("igraph", quietly = TRUE)) {
  g <- igraph::sample_gnp(n = 20, p = 0.15)
  model <- cluster_louvain_graph()
  model <- fit(model, g)
  clu <- cluster(model, g)
  table(clu)
}
```

---

cluster\_pam

*PAM (Partitioning Around Medoids)*

---

## Description

Clustering around representative data points (medoids) using `cluster::pam`.

## Usage

```
cluster_pam(k = 1)
```

## Arguments

`k` the number of clusters to generate.

## Details

More robust to outliers than k-means. The intrinsic metric reported is the within-cluster SSE to medoids.

The base `clusterer()` uses `wcss` as a generic default, but `cluster_pam()` specializes that choice because PAM is commonly compared through partition quality indices that balance cohesion and separation.

Default evaluation in `cluster_pam()` is:

- main metric: `metric_silhouette()`
- internal evaluation: `silhouette`, `davies_bouldin`, `calinski_harabasz`
- external evaluation: `entropy`, `purity`, `adjusted_rand_index`

The generic `wcss` fallback is not used as the main metric because PAM is not optimized around centroids and is usually interpreted through medoid-based partition quality rather than only through within-cluster dispersion.

## Value

returns PAM object.

## References

Kaufman, L. and Rousseeuw, P. J. (1990). Finding Groups in Data: An Introduction to Cluster Analysis. Rousseeuw, P. J. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. Davies, D. L., & Bouldin, D. W. (1979). A cluster separation measure. Calinski, T., & Harabasz, J. (1974). A dendrite method for cluster analysis.

## Examples

```
# setup clustering
model <- cluster_pam(k = 3)

#load dataset
data(iris)

# build model
model <- fit(model, iris[,1:4])
clu <- cluster(model, iris[,1:4])
table(clu)

# evaluate model using external metric
eval <- evaluate(model, clu, iris$Species)
eval
```

cluutils

*Clustering utilities*

---

**Description**

Utility object that groups clustering metrics and model-selection helpers.

**Usage**

```
cluutils()
```

**Details**

The object organizes helpers into two semantic groups:

**Metrics**

- `metric_wcss()` computes the total within-cluster sum of squares.
- `metric_silhouette()` computes the mean silhouette score from pairwise distances.
- `metric_entropy()` computes external clustering entropy against a reference label.
- `metric_purity()` computes cluster purity against a reference label.
- `metric_davies_bouldin()` computes the Davies-Bouldin index.
- `metric_calinski_harabasz()` computes the Calinski-Harabasz score.
- `metric_adjusted_rand_index()` computes the adjusted Rand index.
- `metric_noise_points()` summarizes the number of noise points in density-based clustering.
- `metric_loglik()` and `metric_modularity()` expose model-specific quality summaries.

**Selectors**

- `selector_best()` selects the best hyperparameter value by direct optimization.
- `selector_elbow()` selects the elbow of a metric curve via maximum curvature.

Metric helpers return a standardized list with fields `metric`, `value`, `goal`, and `type`. This keeps the contract uniform even when the metrics themselves differ.

**Value**

returns a `cluutils` object exposing metric and selector helpers.

**Examples**

```
utils <- cluutils()

data(iris)
x <- iris[, 1:4]
clu <- stats::kmeans(x, centers = 3)$cluster

utils$metric_wcss(x, clu)
utils$metric_silhouette(x, clu)
utils$metric_entropy(clu, iris$Species)
utils$selector_best(c(0.31, 0.42, 0.39), goal = "maximize")
```

clu\_tune

*Clustering tuning (intrinsic metric)***Description**

Tune clustering hyperparameters by evaluating an intrinsic metric over a parameter grid and selecting the elbow (max curvature).

**Usage**

```
clu_tune(base_model, folds = 10, ranges = NULL)
```

**Arguments**

|            |                                            |
|------------|--------------------------------------------|
| base_model | base model for tuning                      |
| folds      | number of folds for cross-validation       |
| ranges     | a list of hyperparameter ranges to explore |

**Value**

returns a clu\_tune object.

**References**

Satopaa, V. et al. (2011). Finding a “Kneedle” in a Haystack.

**Examples**

```
data(iris)

# fit model
model <- clu_tune(cluster_kmeans(k = 2), ranges = list(k = 2:10))

model <- fit(model, iris[,1:4])
model$k
```

dal\_base

*Class dal\_base***Description**

Minimal abstract base class for all DAL objects. Defines the common generics `fit()` and `action()` used by transforms and learners.

**Usage**

```
dal_base()
```

**Value**

returns a `dal_base` object

**Examples**

```
trans <- dal_base()
```

dal\_graphics

*Graphics Utilities***Description**

A collection of small plotting helpers built on `ggplot2` used across the package to quickly visualize vectors, grouped summaries, and time series.

**Details**

All functions return a `ggplot2::ggplot` object so the result can be extended with themes, scales, labels, and annotations.

Conventions adopted:

- Input data generally follows the pattern: first column is an index or category (x), remaining columns are numeric series.
- Some functions expect a long format with columns named `x`, `value`, and `variable`.
- The `colors` parameter accepts either a single color or a vector mapped to groups or variables.
- Transparency is controlled by `alpha` where provided.
- All helpers set a light `theme_bw()` baseline and place legends at the bottom by default.

**See Also**

`ggplot2`



**Examples**

```
data <- data.frame(group = c("A", "B"), value = c(2, 5))
grf <- plot_bar(data)
plot(grf)
```

---

|             |                                 |
|-------------|---------------------------------|
| dal_learner | <i>DAL Learner (base class)</i> |
|-------------|---------------------------------|

---

**Description**

Base ancestor for learning tasks (classification, regression, clustering, time series). Provides common behavior such as proxying `action()` to the model-specific operation (e.g., `predict()` for predictors, `cluster()` for clusterers) and an `evaluate()` generic.

An example of a learner is a decision tree (see `cla_dtree`).

**Usage**

```
dal_learner()
```

**Value**

returns a learner object

**Examples**

```
#See ?cla_dtree for a classification example using a decision tree
```

---

|               |                      |
|---------------|----------------------|
| dal_transform | <i>DAL Transform</i> |
|---------------|----------------------|

---

**Description**

Base class for data transformations with optional `fit()`/`inverse_transform()` support.

**Usage**

```
dal_transform()
```

**Details**

The default `transform()` calls the underlying `action.default()`; subclasses should implement `transform.className` and optionally `inverse_transform.className`.

**Value**

returns a `dal_transform` object

## Examples

```
# See ?minmax or ?zscore for examples
```

---

`dal_tune`

*DAL Tune (base for hyperparameter search)*

---

## Description

Base class for hyperparameter optimization that stores a base model, a fold count, and a parameter grid. Specializations (classification/regression/clustering) implement the evaluation logic.

## Usage

```
dal_tune(base_model, folds = 10, ranges)
```

## Arguments

|                         |                                            |
|-------------------------|--------------------------------------------|
| <code>base_model</code> | base model for tuning                      |
| <code>folds</code>      | number of folds for cross-validation       |
| <code>ranges</code>     | a list of hyperparameter ranges to explore |

## Details

Ranges are expanded via `expand.grid`, and selection is delegated to `select_hyper()` which can be overridden by subclasses to implement custom criteria.

## Value

returns a `dal_tune` object

## Examples

```
#See ?cla_tune for classification tuning
#See ?reg_tune for regression tuning
#See ?ts_tune for time series tuning
```

---

|             |                                   |
|-------------|-----------------------------------|
| data_sample | <i>Data sampling abstractions</i> |
|-------------|-----------------------------------|

---

**Description**

Base class for sampling strategies that provide train/test splitting and k-fold partitioning. Two standard implementations are `sample_random()` and `sample_stratified()`.

**Usage**

```
data_sample()
```

**Value**

returns an object of class `data_sample`

**Examples**

```
#using random sampling
sample <- sample_random()
tt <- train_test(sample, iris)

# distribution of train
table(tt$train$Species)

# preparing dataset into four folds
folds <- k_fold(sample, iris, 4)

# distribution of folds
tbl <- NULL
for (f in folds) {
  tbl <- rbind(tbl, table(f$Species))
}
head(tbl)
```

---

|          |                 |
|----------|-----------------|
| discover | <i>Discover</i> |
|----------|-----------------|

---

**Description**

Generic for pattern discovery.

**Usage**

```
discover(obj, ...)
```

**Arguments**

obj                    A pattern\_miner object.  
...                    Optional arguments passed to the concrete discovery method.

**Value**

Discovered patterns in the representation used by the mining engine.

---

|        |            |
|--------|------------|
| dt_pca | <i>PCA</i> |
|--------|------------|

---

**Description**

Principal Component Analysis (PCA) for unsupervised dimensionality reduction. Transforms correlated variables into orthogonal principal components ordered by explained variance.

**Usage**

```
dt_pca(attribute = NULL, components = NULL)
```

**Arguments**

attribute            target attribute to model building  
components          number of components for PCA

**Details**

Fits PCA on (optionally) the numeric predictors only (excluding attribute when provided), removes constant columns, and selects the number of components by an elbow rule (minimum curvature) unless components is set explicitly. New data are projected with the same centering and scaling learned during `fit()`.

**Value**

returns an object of class `dt_pca`

**References**

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components.

**Examples**

```

mypca <- dt_pca("Species")
# Automatically fitting number of components
mypca <- fit(mypca, iris)
iris.pca <- transform(mypca, iris)
head(iris.pca)
head(mypca$pca.transf)
# Manual establishment of number of components
mypca <- dt_pca("Species", 3)
mypca <- fit(mypca, datasets::iris)
iris.pca <- transform(mypca, iris)
head(iris.pca)
head(mypca$pca.transf)

```

---

evaluate

*Evaluate*


---

**Description**

Evaluate learner performance. The actual evaluate varies according to the type of learner (clustering, classification, regression, time series regression)

**Usage**

```
evaluate(obj, ...)
```

**Arguments**

|     |                    |
|-----|--------------------|
| obj | object             |
| ... | optional arguments |

**Value**

returns the evaluation

**Examples**

```

data(iris)
slevels <- levels(iris$Species)
model <- cla_dtree("Species", slevels)
model <- fit(model, iris)
prediction <- predict(model, iris)
predictand <- adjust_class_label(iris[, "Species"])
test_eval <- evaluate(model, predictand, prediction)
test_eval$metrics

```

---

|                    |                           |
|--------------------|---------------------------|
| feature_generation | <i>Feature generation</i> |
|--------------------|---------------------------|

---

**Description**

Create new features from existing columns using named expressions.

**Usage**

```
feature_generation(...)
```

**Arguments**

...                      named expressions that compute new features

**Value**

returns an object of class `feature_generation`

**Examples**

```
data(iris)
gen <- feature_generation(
  Sepal.Area = Sepal.Length * Sepal.Width,
  Petal.Area = Petal.Length * Petal.Width,
  Sepal.Ratio = Sepal.Length / Sepal.Width
)
iris_feat <- transform(gen, iris)
head(iris_feat)
```

---

|                        |                                         |
|------------------------|-----------------------------------------|
| feature_selection_corr | <i>Feature selection by correlation</i> |
|------------------------|-----------------------------------------|

---

**Description**

Remove highly correlated numeric features based on a correlation cutoff.

**Usage**

```
feature_selection_corr(cutoff = 0.9, features = NULL, keep = NULL)
```

**Arguments**

|          |                                                                                   |
|----------|-----------------------------------------------------------------------------------|
| cutoff   | correlation cutoff in [0, 1] above which one feature is removed                   |
| features | optional vector of feature names to consider (default: all numeric columns)       |
| keep     | optional vector of columns that should always be kept in <code>transform()</code> |

**Details**

Uses `caret::findCorrelation` on the correlation matrix computed from numeric columns.

**Value**

returns an object of class `feature_selection_corr`

**Examples**

```
data(iris)
fs <- feature_selection_corr(cutoff = 0.9)
fs <- fit(fs, iris)
iris_fs <- transform(fs, iris)
fs$selected
names(iris_fs)
```

---

`feature_selection_fss` *Feature selection by forward stepwise search*

---

**Description**

Selects numeric predictors using forward stepwise subset search.

**Usage**

```
feature_selection_fss(attribute, features = NULL)
```

**Arguments**

|                        |                                                                          |
|------------------------|--------------------------------------------------------------------------|
| <code>attribute</code> | target attribute name                                                    |
| <code>features</code>  | optional vector of feature names (default: all columns except attribute) |

**Details**

Uses `leaps::regsubsets` and keeps the subset with the highest adjusted R-squared. The target attribute must be numeric.

**Value**

returns an object of class `feature_selection_fss`

### Examples

```
if (requireNamespace("leaps", quietly = TRUE)) {  
  data(Boston)  
  fs <- feature_selection_fss("medv")  
  fs <- fit(fs, Boston)  
  fs$selected  
  boston_fs <- transform(fs, Boston)  
  names(boston_fs)  
}
```

---

feature\_selection\_info\_gain

*Feature selection by information gain*

---

### Description

Rank and select features using information gain with optional discretization.

### Usage

```
feature_selection_info_gain(  
  attribute,  
  features = NULL,  
  top = NULL,  
  cutoff = 0,  
  bins = 3  
)
```

### Arguments

|           |                                                                          |
|-----------|--------------------------------------------------------------------------|
| attribute | target attribute name                                                    |
| features  | optional vector of feature names (default: all columns except attribute) |
| top       | optional number of top features to keep                                  |
| cutoff    | minimum information gain to keep a feature (default: 0)                  |
| bins      | number of quantile bins for numeric features                             |

### Details

Numeric predictors are discretized by quantile bins before computing entropy-based information gain.

### Value

returns an object of class `feature_selection_info_gain`



**Examples**

```
data(iris)
fg <- feature_generation(
  IsVersicolor = ifelse(Species == "versicolor", "versicolor", "not_versicolor")
)
iris_bin <- transform(fg, iris)
iris_bin$IsVersicolor <- factor(iris_bin$IsVersicolor)
fs <- feature_selection_info_gain("IsVersicolor", top = 2)
fs <- fit(fs, iris_bin)
fs$selected
iris_fs <- transform(fs, iris_bin)
names(iris_fs)
```

---

feature\_selection\_lasso

*Feature selection by lasso*

---

**Description**

Selects predictors using L1-regularized regression.

**Usage**

```
feature_selection_lasso(attribute, features = NULL)
```

**Arguments**

|           |                                                                                  |
|-----------|----------------------------------------------------------------------------------|
| attribute | target attribute name                                                            |
| features  | optional vector of feature names (default: all numeric columns except attribute) |

**Details**

Fits a lasso path with glmnet and keeps predictors with non-zero coefficients at lambda.min. The target attribute must be numeric.

**Value**

returns an object of class feature\_selection\_lasso

**Examples**

```
if (requireNamespace("glmnet", quietly = TRUE)) {
  data(Boston)
  fs <- feature_selection_lasso("medv")
  fs <- fit(fs, Boston)
  fs$selected
  boston_fs <- transform(fs, Boston)
  names(boston_fs)
}
```

---

`feature_selection_relief`*Feature selection by RELIEF*

---

**Description**

Rank and select features using a simplified RELIEF algorithm.

**Usage**

```
feature_selection_relief(  
  attribute,  
  features = NULL,  
  top = NULL,  
  cutoff = NULL,  
  m = 50  
)
```

**Arguments**

|                        |                                                                                        |
|------------------------|----------------------------------------------------------------------------------------|
| <code>attribute</code> | target attribute name                                                                  |
| <code>features</code>  | optional vector of feature names (default: all columns except <code>attribute</code> ) |
| <code>top</code>       | optional number of top features to keep                                                |
| <code>cutoff</code>    | optional minimum RELIEF weight to keep a feature                                       |
| <code>m</code>         | number of sampled instances for RELIEF updates                                         |

**Details**

For each sampled instance, the algorithm compares nearest hit/miss neighbors and updates feature weights.

**Value**

returns an object of class `feature_selection_relief`

**Examples**

```
data(iris)  
fg <- feature_generation(  
  IsVersicolor = ifelse(Species == "versicolor", "versicolor", "not_versicolor")  
)  
iris_bin <- transform(fg, iris)  
iris_bin$IsVersicolor <- factor(iris_bin$IsVersicolor)  
fs <- feature_selection_relief("IsVersicolor", top = 2, m = 50)  
fs <- fit(fs, iris_bin)  
fs$selected  
transform(fs, iris_bin) |> names()
```

---

`feature_selection_stepwise`*Feature selection by stepwise model selection*

---

## Description

Select features using stepwise search over generalized linear models.

## Usage

```
feature_selection_stepwise(  
  attribute,  
  features = NULL,  
  direction = "forward",  
  family = stats::binomial,  
  trace = 0  
)
```

## Arguments

|                        |                                                                          |
|------------------------|--------------------------------------------------------------------------|
| <code>attribute</code> | target attribute name                                                    |
| <code>features</code>  | optional vector of feature names (default: all columns except attribute) |
| <code>direction</code> | stepwise direction: "forward", "backward", or "both"                     |
| <code>family</code>    | glm family passed to <code>stats::glm</code> (default: binomial)         |
| <code>trace</code>     | level of tracing from <code>stats::step</code>                           |

## Details

Supports forward, backward, and both directions via `stats::step`. With the default binomial family, the target should represent a binary outcome.

## Value

returns an object of class `feature_selection_stepwise`

## Examples

```
data(Boston)  
fs <- feature_selection_stepwise("medv", direction = "forward", family = stats::gaussian)  
fs <- fit(fs, Boston)  
fs$selected  
transform(fs, Boston) |> names()
```

---

|                  |            |
|------------------|------------|
| <code>fit</code> | <i>Fit</i> |
|------------------|------------|

---

**Description**

Generic to train/adjust an object using provided data and optional parameters.

**Usage**

```
fit(obj, ...)
```

**Arguments**

- `obj` object
- `...` optional arguments.

**Value**

returns a object after fitting

**Examples**

```
data(iris)
# an example is minmax normalization
trans <- minmax()
trans <- fit(trans, iris)
tiris <- action(trans, iris)
```

---

|                           |                                         |
|---------------------------|-----------------------------------------|
| <code>fit.cla_tune</code> | <i>tune hyperparameters of ml model</i> |
|---------------------------|-----------------------------------------|

---

**Description**

Tunes the hyperparameters of a machine learning model for classification

**Usage**

```
## S3 method for class 'cla_tune'
fit(obj, data, ...)
```

**Arguments**

- `obj` an object containing the model and tuning configuration
- `data` the dataset used for training and evaluation
- `...` optional arguments

**Value**

a fitted obj

---

|                    |                         |
|--------------------|-------------------------|
| fit.cluster_dbscan | <i>fit dbscan model</i> |
|--------------------|-------------------------|

---

**Description**

Fits a DBSCAN clustering model by setting the eps parameter. If eps is not provided, it is estimated based on the k-nearest neighbor distances. It wraps dbscan library

**Usage**

```
## S3 method for class 'cluster_dbscan'
fit(obj, data, ...)
```

**Arguments**

|      |                                                                                          |
|------|------------------------------------------------------------------------------------------|
| obj  | an object containing the DBSCAN model configuration, including minPts and optionally eps |
| data | the dataset to use for fitting the model                                                 |
| ...  | optional arguments                                                                       |

**Value**

returns a fitted obj with the eps parameter set

---

|                   |                                                     |
|-------------------|-----------------------------------------------------|
| fit_curvature_max | <i>Maximum curvature analysis (elbow detection)</i> |
|-------------------|-----------------------------------------------------|

---

**Description**

Computes a smoothing spline over a sequence and returns the location/value of maximum curvature, often used as an "elbow" detector.

**Usage**

```
fit_curvature_max()
```

**Value**

returns an object of class fit\_curvature\_max, which inherits from the fit\_curvature and dal\_transform classes. The object contains a list with the following elements:

- x: The position in which the maximum curvature is reached.
- y: The value where the the maximum curvature occurs.
- yfit: The value of the maximum curvature.

### Examples

```
x <- seq(from=1,to=10,by=0.5)
dat <- data.frame(x = x, value = -log(x), variable = "log")
myfit <- fit_curvature_max()
res <- transform(myfit, dat$value)
head(res)
```

---

|                   |                                                     |
|-------------------|-----------------------------------------------------|
| fit_curvature_min | <i>Minimum curvature analysis (elbow detection)</i> |
|-------------------|-----------------------------------------------------|

---

### Description

Computes a smoothing spline over a sequence and returns the location/value of minimum curvature, complementary to maximum curvature and useful in elbow detection.

### Usage

```
fit_curvature_min()
```

### Value

Returns an object of class `fit_curvature_min`, which inherits from the `fit_curvature` and `dal_transform` classes. The object contains a list with the following elements:

- `x`: The position in which the minimum curvature is reached.
- `y`: The value where the the minimum curvature occurs.
- `yfit`: The value of the minimum curvature.

### Examples

```
x <- seq(from=1,to=10,by=0.5)
dat <- data.frame(x = x, value = log(x), variable = "log")
myfit <- fit_curvature_min()
res <- transform(myfit, dat$value)
head(res)
```

---

|               |                                 |
|---------------|---------------------------------|
| hierarchy_cut | <i>Hierarchy mapping by cut</i> |
|---------------|---------------------------------|

---

**Description**

Create a categorical hierarchy from a numeric attribute using cut points.

**Usage**

```
hierarchy_cut(attribute, breaks, labels = NULL, new_attribute = NULL)
```

**Arguments**

|               |                                                        |
|---------------|--------------------------------------------------------|
| attribute     | numeric attribute to discretize                        |
| breaks        | numeric breakpoints for cut                            |
| labels        | optional labels for the cut intervals                  |
| new_attribute | name of the new attribute (default: "attribute.Level") |

**Value**

returns an object of class hierarchy\_cut

**Examples**

```
data(iris)
hc <- hierarchy_cut(
  "Sepal.Length",
  breaks = c(-Inf, 5.5, 6.5, Inf),
  labels = c("baixo", "medio", "alto")
)
iris_h <- transform(hc, iris)
table(iris_h$Sepal.Length.Level)
```

---

|                       |                                   |
|-----------------------|-----------------------------------|
| imputation_predictive | <i>Predictive imputation base</i> |
|-----------------------|-----------------------------------|

---

**Description**

Base class for supervised imputers that learn one target column from a set of source columns.

**Usage**

```
imputation_predictive(target, sources = NULL, method = c("median", "mean"))
```

**Arguments**

|         |                                                                          |
|---------|--------------------------------------------------------------------------|
| target  | target column to impute                                                  |
| sources | optional vector of predictor column names                                |
| method  | initial imputation method for numeric source columns: "median" or "mean" |

**Details**

The target column is the attribute to be imputed. The source columns are the predictors used to estimate missing target values. If `sources = NULL`, all supported columns except the target are used. Missing values in source columns can be pre-imputed by a simpler method before fitting the predictive model.

**Value**

returns an object of class `imputation_predictive`

**Examples**

```
data(iris)
imp <- imputation_predictive(
  "Sepal.Length",
  sources = c("Sepal.Width", "Petal.Length", "Petal.Width", "Species")
)
class(imp)
```

---

|                   |                          |
|-------------------|--------------------------|
| imputation_simple | <i>Simple imputation</i> |
|-------------------|--------------------------|

---

**Description**

Impute missing values in mixed datasets using simple statistics.

**Usage**

```
imputation_simple(method = c("median", "mean"), cols = NULL)
```

**Arguments**

|        |                                                                            |
|--------|----------------------------------------------------------------------------|
| method | imputation method for numeric columns: "median" or "mean"                  |
| cols   | optional vector of column names to impute (default: all supported columns) |

**Details**

Numeric columns are imputed with the mean or median. Factor, character, logical, and ordered columns are imputed with the mode (most frequent observed value). This class is intended as a low-complexity baseline for preprocessing workflows. The default recommendation of median for numeric variables follows standard data preprocessing guidance because it is less sensitive to outliers than the mean, while mode imputation is the usual baseline for categorical attributes.



**Value**

returns an object of class `imputation_simple`

**References**

Han, J., Kamber, M., Pei, J. (2011). Data Mining: Concepts and Techniques.  
 Little, R. J. A., Rubin, D. B. (2019). Statistical Analysis with Missing Data.

**Examples**

```
data(iris)
iris_na <- iris
iris_na$Sepal.Length[c(2, 10, 25)] <- NA
iris_na$Species[c(3, 15)] <- NA

imp <- imputation_simple(method = "median")
imp <- fit(imp, iris_na)
iris_imp <- transform(imp, iris_na)
summary(iris_imp$Sepal.Length)
table(iris_imp$Species, useNA = "ifany")
```

---

|                 |                                         |
|-----------------|-----------------------------------------|
| imputation_tree | <i>Tree-based predictive imputation</i> |
|-----------------|-----------------------------------------|

---

**Description**

Impute one target column from a set of source columns using a decision tree.

**Usage**

```
imputation_tree(target, sources = NULL, method = c("median", "mean"))
```

**Arguments**

|         |                                                                                          |
|---------|------------------------------------------------------------------------------------------|
| target  | target column to impute                                                                  |
| sources | optional vector of predictor column names (default: all supported columns except target) |
| method  | initial imputation method for numeric source columns: "median" or "mean"                 |

**Details**

The method fits a tree with the observed values of the target column and uses the source columns as predictors. If source columns contain missing values, they are first completed with `imputation_simple()` so the tree can be trained and applied. The learned model imputes only the target column; source columns are preserved in the returned data.

**Value**

returns an object of class `imputation_tree`

**References**

Breiman, L., Friedman, J., Olshen, R., Stone, C. (1984). Classification and Regression Trees. Wadsworth.

van Buuren, S., Groothuis-Oudshoorn, K. (2011). mice: Multivariate Imputation by Chained Equations in R. Journal of Statistical Software, 45(3), 1-67.

**Examples**

```
data(iris)
iris_na <- iris
iris_na$Sepal.Length[c(2, 10, 25)] <- NA

imp <- imputation_tree("Sepal.Length")
imp <- fit(imp, iris_na)
iris_imp <- transform(imp, iris_na)
summary(iris_imp$Sepal.Length)
sum(is.na(iris_imp$Sepal.Length))
```

---

|                   |                          |
|-------------------|--------------------------|
| inverse_transform | <i>Inverse Transform</i> |
|-------------------|--------------------------|

---

**Description**

Optional inverse operation for a transformation; defaults to identity.

**Usage**

```
inverse_transform(obj, ...)
```

**Arguments**

|                  |                                      |
|------------------|--------------------------------------|
| <code>obj</code> | a <code>dal_transform</code> object. |
| <code>...</code> | optional arguments.                  |

**Value**

dataset inverse transformed.

**Examples**

```
#See ?minmax for an example of transformation
```

---

|        |                        |
|--------|------------------------|
| k_fold | <i>K-fold sampling</i> |
|--------|------------------------|

---

**Description**

Split a dataset into k folds using a sampling strategy.

**Usage**

```
k_fold(obj, data, k)
```

**Arguments**

|      |                                            |
|------|--------------------------------------------|
| obj  | an object representing the sampling method |
| data | dataset to be partitioned                  |
| k    | number of folds                            |

**Value**

returns a list of k data frames

**Examples**

```
#using random sampling
sample <- sample_random()

# preparing dataset into four folds
folds <- k_fold(sample, iris, 4)

# distribution of folds
tbl <- NULL
for (f in folds) {
  tbl <- rbind(tbl, table(f$Species))
}
head(tbl)
```

---

|        |                              |
|--------|------------------------------|
| minmax | <i>Min-max normalization</i> |
|--------|------------------------------|

---

**Description**

Linearly scales numeric columns to the [0,1] range per column.

**Usage**

```
minmax()
```

**Details**

For each numeric column  $j$ , computes  $(x - \min_j) / (\max_j - \min_j)$ . Constant columns map to 0.

$$\text{minmax} = (x - \min(x)) / (\max(x) - \min(x))$$

**Value**

returns an object of class minmax

**References**

Han, J., Kamber, M., Pei, J. (2011). Data Mining: Concepts and Techniques. (Normalization section)

**Examples**

```
data(iris)
head(iris)

trans <- minmax()
trans <- fit(trans, iris)
tiris <- transform(trans, iris)
head(tiris)

itiris <- inverse_transform(trans, tiris)
head(itiris)
```

---

|            |                              |
|------------|------------------------------|
| na_removal | <i>Missing value removal</i> |
|------------|------------------------------|

---

**Description**

Remove rows (or elements) that contain missing values.

**Usage**

```
na_removal()
```

**Details**

For data frames or matrices, removes rows with any NA. For vectors, removes NA values.

**Value**

returns an object of class na\_removal

**Examples**

```
data(iris)
iris.na <- iris
iris.na$Sepal.Length[2] <- NA
obj <- na_removal()
iris.clean <- transform(obj, iris.na)
nrow(iris.clean)
```

---

outliers\_boxplot

*Outlier removal by boxplot (IQR rule)*


---

**Description**

Removes outliers from numeric columns using Tukey's boxplot rule: values below  $Q1 - \alpha \cdot IQR$  or above  $Q3 + \alpha \cdot IQR$  are flagged as outliers.

**Usage**

```
outliers_boxplot(alpha = 1.5)
```

**Arguments**

alpha                      boxplot outlier threshold (default 1.5, but can be 3.0 to remove extreme values)

**Details**

The default  $\alpha=1.5$  corresponds to the standard boxplot whiskers;  $\alpha=3$  is used for extreme outliers.

**Value**

returns an outlier object

**References**

Tukey, J. W. (1977). Exploratory Data Analysis. Addison-Wesley.

**Examples**

```
# code for outlier removal
out_obj <- outliers_boxplot() # class for outlier analysis
out_obj <- fit(out_obj, iris) # computing boundaries
iris.clean <- transform(out_obj, iris) # returning cleaned dataset

#inspection of cleaned dataset
nrow(iris.clean)

idx <- attr(iris.clean, "idx")
table(idx)
```

```
iris.outliers_boxplot <- iris[idx,]  
iris.outliers_boxplot
```

---

|                   |                                                 |
|-------------------|-------------------------------------------------|
| outliers_gaussian | <i>Outlier removal by Gaussian 3-sigma rule</i> |
|-------------------|-------------------------------------------------|

---

### Description

Removes outliers from numeric columns using the 3-sigma rule under a Gaussian assumption: values outside  $\text{mean} \pm \alpha \cdot \text{sd}$  are flagged as outliers.

### Usage

```
outliers_gaussian(alpha = 3)
```

### Arguments

|       |                                |
|-------|--------------------------------|
| alpha | gaussian threshold (default 3) |
|-------|--------------------------------|

### Value

returns an outlier object

### References

Pukelsheim, F. (1994). The Three Sigma Rule. The American Statistician 48(2):88–91.

### Examples

```
# code for outlier removal  
out_obj <- outliers_gaussian() # class for outlier analysis  
out_obj <- fit(out_obj, iris) # computing boundaries  
iris.clean <- transform(out_obj, iris) # returning cleaned dataset  
  
#inspection of cleaned dataset  
nrow(iris.clean)  
  
idx <- attr(iris.clean, "idx")  
table(idx)  
iris.outliers_gaussian <- iris[idx,]  
iris.outliers_gaussian
```

---

pattern\_miner

*Pattern Miner*

---

### Description

Base class for frequent pattern and sequence mining.

### Usage

```
pattern_miner()
```

### Details

Pattern miners follow a lightweight Experiment Line:

- `fit()` validates the mining input and stores a schema signature
- `discover()` runs the mining algorithm on data compatible with that schema
- `evaluate()` summarizes pattern quality and filtering effects

Different miners may normalize their inputs differently (for example, item transactions versus sequence transactions), but the base contract remains the same.

### Value

A `pattern_miner` object.

### Examples

```
miner <- pattern_miner()
class(miner)
```

---

patutils

*Pattern mining utilities*

---

### Description

Utility object that groups filtering helpers and evaluation metrics for pattern mining.

### Usage

```
patutils()
```

## Details

The object groups two kinds of helpers:

- quality-filter builders such as `quality_min()` and `quality_max()`
- descriptive metrics for discovered patterns such as pattern count, mean support, mean confidence, mean lift, mean length, and retained ratio after filtering

## Value

returns a `patutils` object

## Examples

```
utils <- patutils()
utils$quality_min(confidence = 0.8, lift = 1.1)
```

---

pat\_apriori

*Apriori rules*

---

## Description

Frequent itemsets and association rules using `arules::apriori`.

## Usage

```
pat_apriori(
  target = c("rules", "frequent itemsets"),
  supp = 0,
  conf = 0,
  support_strategy = pat_support_threshold("curvature"),
  confidence_strategy = pat_confidence_threshold(),
  minlen = 2,
  maxlen = 10,
  lhs = NULL,
  rhs = NULL,
  include = NULL,
  exclude = NULL,
  quality_filter = NULL,
  rule_filter = pat_rule_filter_none(),
  control = NULL
)
```



**Arguments**

|                     |                                                                                                        |
|---------------------|--------------------------------------------------------------------------------------------------------|
| target              | mining target: "rules" or "frequent itemsets"                                                          |
| supp                | minimum support threshold. If 0, estimated during fit() using support_strategy.                        |
| conf                | minimum confidence threshold for rules. If 0, estimated during fit() using confidence_strategy.        |
| support_strategy    | support threshold strategy created with pat_support_threshold()                                        |
| confidence_strategy | confidence threshold strategy created with pat_confidence_threshold()                                  |
| minlen              | minimum pattern length                                                                                 |
| maxlen              | maximum pattern length                                                                                 |
| lhs                 | optional vector of items constrained to the left-hand side of rules                                    |
| rhs                 | optional vector of items constrained to the right-hand side of rules                                   |
| include             | optional vector of items allowed in the discovered patterns                                            |
| exclude             | optional vector of items forbidden in the discovered patterns                                          |
| quality_filter      | optional quality filter created with patutils()                                                        |
| rule_filter         | rule filter created with pat_rule_filter_none(), pat_rule_filter_interest(), or pat_rule_filter_dara() |
| control             | list of control parameters                                                                             |

**Value**

returns a pat\_apriori object

**Examples**

```
if (requireNamespace("arules", quietly = TRUE)) {
  data("AdultUCI", package = "arules")
  trans <- suppressWarnings(methods::as(as.data.frame(AdultUCI), "transactions"))
  utils <- patutils()
  pm <- pat_apriori(
    target = "rules",
    supp = 0,
    conf = 0,
    support_strategy = pat_support_threshold("curvature"),
    confidence_strategy = pat_confidence_threshold("rhs_baseline", margin = 0.1),
    minlen = 2,
    maxlen = 3,
    rhs = c("native-country=United-States"),
    quality_filter = utils$quality_min(confidence = 0.9, lift = 1.03),
    rule_filter = pat_rule_filter_interest(lift_min = 1),
    control = list(verbose = FALSE)
  )
  pm <- fit(pm, trans)
  rules <- suppressWarnings(discover(pm, trans))
  eval <- evaluate(pm, rules)
  eval$metrics
}
```

---

`pat_confidence_threshold`*Pattern confidence threshold strategy*

---

## Description

Configure how a minimum confidence threshold is estimated during `fit()`.

## Usage

```
pat_confidence_threshold(  
  method = c("rhs_baseline", "support_adaptive", "fixed"),  
  value = 0.8,  
  margin = 0.1,  
  scale = 1,  
  min = 0.5,  
  max = 0.95  
)
```

## Arguments

|                     |                                                                |
|---------------------|----------------------------------------------------------------|
| <code>method</code> | Strategy name: "fixed", "rhs_baseline", or "support_adaptive". |
| <code>value</code>  | Fixed confidence used by "fixed".                              |
| <code>margin</code> | Added to the RHS baseline confidence for "rhs_baseline".       |
| <code>scale</code>  | Multiplicative factor used by "support_adaptive".              |
| <code>min</code>    | Minimum allowed confidence.                                    |
| <code>max</code>    | Maximum allowed confidence.                                    |

## Value

A `pat_confidence_threshold` object.

## Examples

```
pat_confidence_threshold("rhs_baseline", margin = 0.1, min = 0.5)  
pat_confidence_threshold("support_adaptive", min = 0.6, max = 0.9)
```

---

|            |                         |
|------------|-------------------------|
| pat_cspade | <i>cSPADE sequences</i> |
|------------|-------------------------|

---

## Description

Sequential pattern mining using `arulesSequences::cspade`.

## Usage

```
pat_cspade(
  support = 0,
  support_strategy = pat_support_threshold("min_count", min_count = 2),
  maxsize = NULL,
  maxlen = NULL,
  mingap = NULL,
  maxgap = NULL,
  quality_filter = NULL,
  control = list(verbose = TRUE)
)
```

## Arguments

|                               |                                                                                                            |
|-------------------------------|------------------------------------------------------------------------------------------------------------|
| <code>support</code>          | minimum support threshold. If 0, estimated during <code>fit()</code> using <code>support_strategy</code> . |
| <code>support_strategy</code> | support threshold strategy created with <code>pat_support_threshold()</code>                               |
| <code>maxsize</code>          | maximum number of items per event                                                                          |
| <code>maxlen</code>           | maximum number of events per sequence                                                                      |
| <code>mingap</code>           | minimum gap between successive events                                                                      |
| <code>maxgap</code>           | maximum gap between successive events                                                                      |
| <code>quality_filter</code>   | optional quality filter created with <code>patutils()</code>                                               |
| <code>control</code>          | list of control parameters                                                                                 |

## Value

returns a `pat_cspade` object

## Examples

```
if (requireNamespace("arulesSequences", quietly = TRUE)) {
  x <- arulesSequences::read_baskets(
    con = system.file("misc", "zaki.txt", package = "arulesSequences"),
    info = c("sequenceID", "eventID", "SIZE")
  )
  utils <- patutils()
  pm <- pat_cspade(
    support = 0,
```

```

    support_strategy = pat_support_threshold("min_count", min_count = 2),
    maxlen = 3,
    quality_filter = utils$quality_min(support = 0.5)
  )
  pm <- fit(pm, x)
  seqs <- discover(pm, x)
  eval <- evaluate(pm, seqs)
  eval$metrics
}

```

pat\_dara

*Divergent Association Ranking Analysis***Description**

Rank attribute values whose ordering diverges between the dataset and discovered rules.

**Usage**

```

pat_dara(
  data,
  rules,
  rhs_attr,
  rhs_value = NULL,
  attributes = NULL,
  rule_weight = NULL
)

```

**Arguments**

|             |                                                                                                   |
|-------------|---------------------------------------------------------------------------------------------------|
| data        | Reference data frame.                                                                             |
| rules       | Tidy rule data frame produced by <code>pat_dara_rules()</code> or <code>pat_rules_tidy()</code> . |
| rhs_attr    | Right-hand-side attribute used by the mined rules.                                                |
| rhs_value   | Optional right-hand-side value used to restrict data.                                             |
| attributes  | Optional attributes to compare. Defaults to common columns in data and rules.                     |
| rule_weight | Optional numeric column in rules used as rule frequency instead of one count per rule.            |

**Details**

DARA compares two value-frequency orderings for each explanatory attribute: the ordering in a reference dataset, optionally restricted to a right-hand-side value, and the ordering in a table of association rules. The right-hand-side attribute is required so it can be excluded from explanatory divergence. The returned counter `c` increases for values that are more prominent in rules than in the reference data and decreases for values that are less prominent in rules.

**Value**

A list of per-attribute data frames with columns value, dataset, rules, and c.

**Examples**

```
if (requireNamespace("arules", quietly = TRUE)) {
  data("AdultUCI", package = "arules")
  data <- as.data.frame(AdultUCI)
  data <- data[, c("workclass", "education", "marital-status", "occupation", "income")]
  rules <- pat_dara_rules(data, rhs = "income=small", supp = 0.05, conf = 0.6)
  ranking <- pat_dara(data, rules, rhs_attr = "income", rhs_value = "small")
  ranking[["marital-status"]]
}
```

---

pat\_dara\_rules

---

*Generate target association rules for DARA*


---

**Description**

Mine association rules for a fixed right-hand side and return them as a data frame.

**Usage**

```
pat_dara_rules(
  data,
  rhs,
  supp = 0,
  conf = 0,
  support_strategy = pat_support_threshold("curvature"),
  confidence_strategy = pat_confidence_threshold(),
  minlen = 3,
  maxlen = 4,
  quality_filter = NULL,
  rule_filter = pat_rule_filter_none(),
  remove_redundant = TRUE,
  control = list(verbose = FALSE)
)
```

**Arguments**

|                  |                                                                             |
|------------------|-----------------------------------------------------------------------------|
| data             | A data frame or transactions object.                                        |
| rhs              | Right-hand-side item, such as "income=small".                               |
| supp             | Minimum support. If 0, estimated during fit() using support_strategy.       |
| conf             | Minimum confidence. If 0, estimated during fit() using confidence_strategy. |
| support_strategy | support threshold strategy created with pat_support_threshold()             |

confidence\_strategy      confidence threshold strategy created with pat\_confidence\_threshold()  
 minlen                    Minimum rule length.  
 maxlen                    Maximum rule length.  
 quality\_filter      Optional quality filter created with patutils().  
 rule\_filter            rule filter created with pat\_rule\_filter\_none(), pat\_rule\_filter\_interest(),  
                          or pat\_rule\_filter\_dara().  
 remove\_redundant      If TRUE, remove redundant rules using arules::is.redundant.  
 control                Apriori control list.

### Value

A tidy rule data frame.

### Examples

```

if (requireNamespace("arules", quietly = TRUE)) {
  data("AdultUCI", package = "arules")
  data <- as.data.frame(AdultUCI)
  data <- data[, c("workclass", "education", "marital-status", "occupation", "income")]
  rules <- pat_dara_rules(data, rhs = "income=small", supp = 0.05, conf = 0.6)
  head(rules)
}

```

---

pat\_eclat

*ECLAT itemsets*

---

### Description

Frequent itemsets using arules::eclat.

### Usage

```

pat_eclat(
  supp = 0,
  support_strategy = pat_support_threshold("curvature"),
  minlen = 1,
  maxlen = 3,
  include = NULL,
  exclude = NULL,
  quality_filter = NULL,
  control = NULL
)

```

**Arguments**

supp                minimum support threshold. If 0, estimated during fit() using support\_strategy.  
 support\_strategy    support threshold strategy created with pat\_support\_threshold()  
 minlen             minimum itemset length  
 maxlen             maximum itemset length  
 include            optional vector of items allowed in the discovered itemsets  
 exclude            optional vector of items forbidden in the discovered itemsets  
 quality\_filter     optional quality filter created with patutils()  
 control            list of control parameters

**Value**

returns a pat\_eclat object

**Examples**

```

if (requireNamespace("arules", quietly = TRUE)) {
  data("AdultUCI", package = "arules")
  trans <- suppressWarnings(methods::as(as.data.frame(AdultUCI), "transactions"))
  utils <- patutils()
  pm <- pat_eclat(
    supp = 0,
    support_strategy = pat_support_threshold("curvature"),
    maxlen = 3,
    include = c("sex=Male", "income=small", "marital-status=Married-civ-spouse", "race=White"),
    exclude = c("income=small"),
    quality_filter = utils$quality_min(support = 0.4),
    control = list(verbose = FALSE)
  )
  pm <- fit(pm, trans)
  itemsets <- discover(pm, trans)
  eval <- evaluate(pm, itemsets)
  eval$metrics
}

```

---

|                  |                            |
|------------------|----------------------------|
| pat_filter_rules | <i>Apply a rule filter</i> |
|------------------|----------------------------|

---

**Description**

Filter association rules using a none, interest, or DARA rule filter.

**Usage**

```
pat_filter_rules(rules, filter, data = NULL, ...)
```

**Arguments**

|        |                                                                                                                                                    |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| rules  | An arules rules object or a tidy rule data frame.                                                                                                  |
| filter | A rule filter produced by <code>pat_rule_filter_none()</code> , <code>pat_rule_filter_interest()</code> , or <code>pat_rule_filter_dara()</code> . |
| data   | Optional transaction/reference data used to compute interest measures for arules rules.                                                            |
| ...    | Additional arguments reserved for future filters.                                                                                                  |

**Value**

Filtered rules in the same representation as rules.

**Examples**

```
if (requireNamespace("arules", quietly = TRUE)) {
  data("AdultUCI", package = "arules")
  data <- as.data.frame(AdultUCI)
  data <- data[, c("workclass", "education", "marital-status", "occupation", "income")]
  rules <- pat_dara_rules(data, rhs = "income=small", supp = 0.05, conf = 0.6)
  filter <- pat_rule_filter_interest(lift_min = 1)
  pat_filter_rules(rules, filter)
}
```

---

|                |                               |
|----------------|-------------------------------|
| pat_rules_tidy | <i>Tidy association rules</i> |
|----------------|-------------------------------|

---

**Description**

Convert association rules into a tabular representation suitable for downstream analysis.

**Usage**

```
pat_rules_tidy(rules, transactions = NULL, item_columns = NULL)
```

**Arguments**

|              |                                                                         |
|--------------|-------------------------------------------------------------------------|
| rules        | An arules rules object.                                                 |
| transactions | Optional transaction data used to compute additional interest measures. |
| item_columns | Optional columns to recreate from left-hand-side items.                 |

**Value**

A data frame with rule labels, quality measures, and optional item columns.



**Examples**

```

if (requireNamespace("arules", quietly = TRUE)) {
  data("AdultUCI", package = "arules")
  data <- as.data.frame(AdultUCI)
  data <- data[, c("workclass", "education", "marital-status", "occupation", "income")]
  rules <- pat_dara_rules(data, rhs = "income=small", supp = 0.05, conf = 0.6)
  head(rules)
}

```

---

pat\_rule\_filter\_dara    *DARA rule filter*

---

**Description**

Build a rule filter based on Divergent Association Ranking Analysis.

**Usage**

```

pat_rule_filter_dara(
  data,
  rhs_attr,
  rhs_value = NULL,
  attributes = NULL,
  min_score = 1,
  top = NULL,
  score = c("sum", "mean", "max"),
  use_abs = FALSE,
  rule_weight = NULL
)

```

**Arguments**

|                          |                                                               |
|--------------------------|---------------------------------------------------------------|
| <code>data</code>        | Reference data frame used by DARA.                            |
| <code>rhs_attr</code>    | Right-hand-side attribute used by the mined rules.            |
| <code>rhs_value</code>   | Optional right-hand-side value used to restrict data.         |
| <code>attributes</code>  | Optional attributes to compare. Defaults to common columns.   |
| <code>min_score</code>   | Minimum DARA score required to keep a rule.                   |
| <code>top</code>         | Optional number of highest-scored rules to keep.              |
| <code>score</code>       | Rule score aggregation: "sum", "mean", or "max".              |
| <code>use_abs</code>     | If TRUE, rank by absolute divergence counters.                |
| <code>rule_weight</code> | Optional numeric rule column used to weight rule frequencies. |

**Details**

DARA filters rules by scoring each antecedent with the divergence counters computed by `pat_dara()`. A rule with antecedent values that are more prominent in the rule set than in the exploratory data receives a higher positive score.

**Value**

A pat\_rule\_filter\_dara object.

**Examples**

```
data <- data.frame(a = factor(c("x", "x", "y")), b = factor(c("z", "w", "z")))
filter <- pat_rule_filter_dara(data, rhs_attr = "b", rhs_value = "z")
class(filter)
```

---

pat\_rule\_filter\_interest

*Interesting rule filter*

---

**Description**

Build a rule filter based on classical interest measures.

**Usage**

```
pat_rule_filter_interest(
  lift_min = 1,
  kulc_min = NULL,
  ir_max = NULL,
  support_min = NULL,
  confidence_min = NULL,
  count_min = NULL
)
```

**Arguments**

|                |                                                              |
|----------------|--------------------------------------------------------------|
| lift_min       | Minimum lift. Use NULL to ignore lift.                       |
| kulc_min       | Minimum Kulczynski measure. Use NULL to ignore Kulczynski.   |
| ir_max         | Maximum imbalance ratio. Use NULL to ignore imbalance ratio. |
| support_min    | Minimum support. Use NULL to ignore support.                 |
| confidence_min | Minimum confidence. Use NULL to ignore confidence.           |
| count_min      | Minimum rule count. Use NULL to ignore count.                |

**Details**

This filter follows the usual post-processing step for association rules: rules are discovered first, then kept only when their interest measures pass user-defined thresholds.  $lift > 1$ , high Kulczynski, and low imbalance ratio are common criteria discussed in association-rule mining texts such as Han, Kamber, and Pei.

**Value**

A pat\_rule\_filter\_interest object.

**Examples**

```
filter <- pat_rule_filter_interest(lift_min = 1, kulc_min = 0.6, ir_max = 0.9)
class(filter)
```

---

pat\_rule\_filter\_none    *No rule filtering*

---

**Description**

Build an explicit rule filter that leaves discovered rules unchanged.

**Usage**

```
pat_rule_filter_none()
```

**Details**

Use this filter when the miner should only discover candidate rules and skip post-mining interestingness filtering. It is the default rule filter for association-rule miners.

**Value**

A pat\_rule\_filter\_none object.

**Examples**

```
filter <- pat_rule_filter_none()
class(filter)
```

---

pat\_support\_threshold    *Pattern support threshold strategy*

---

**Description**

Configure how a minimum support threshold is estimated during fit().

**Usage**

```
pat_support_threshold(
  method = c("curvature", "min_count", "sqrt", "quantile", "density"),
  min_count = 10,
  q = 0.25,
  scale = 1,
  min = 0.001,
  max = 0.5
)
```

Arguments

|           |                                                                            |
|-----------|----------------------------------------------------------------------------|
| method    | Strategy name: "curvature", "min_count", "sqrt", "quantile", or "density". |
| min_count | Minimum absolute transaction count for "min_count".                        |
| q         | Quantile of item frequencies for "quantile".                               |
| scale     | Multiplicative factor used by "quantile" and "density".                    |
| min       | Minimum allowed support.                                                   |
| max       | Maximum allowed support.                                                   |

Value

A pat\_support\_threshold object.

Examples

```
pat_support_threshold("curvature")
pat_support_threshold("min_count", min_count = 20)
pat_support_threshold("quantile", q = 0.25, scale = 0.5)
```

---

|          |                       |
|----------|-----------------------|
| plot_bar | <i>Plot Bar Graph</i> |
|----------|-----------------------|

---

Description

Draw a simple bar chart from a two-column data frame: first column as categories (x), second as values (y).

Usage

```
plot_bar(data, label_x = "", label_y = "", colors = NULL, alpha = 1)
```

Arguments

|         |                                                                                    |
|---------|------------------------------------------------------------------------------------|
| data    | Two-column data frame: category in the first column, numeric values in the second. |
| label_x | Label for the x-axis.                                                              |
| label_y | Label for the y-axis.                                                              |
| colors  | Optional fill color.                                                               |
| alpha   | Bar transparency between 0 and 1.                                                  |

Details

If colors is provided, a constant fill is used; otherwise ggplot2’s default palette applies. alpha controls bar transparency. The first column is coerced to factor when needed, which makes the function convenient for quick summaries of grouped values.

**Value**

A ggplot2::ggplot object.

**References**

Wickham, H. (2016). ggplot2: Elegant Graphics for Data Analysis. Springer.

**Examples**

```
# summarizing iris dataset
data <- iris |>
  dplyr::group_by(Species) |>
  dplyr::summarize(Sepal.Length = mean(Sepal.Length))
head(data)

# plotting data
grf <- plot_bar(data, colors = "blue")
plot(grf)
```

---

|              |                     |
|--------------|---------------------|
| plot_boxplot | <i>Plot boxplot</i> |
|--------------|---------------------|

---

**Description**

Boxplots for each numeric column of a data.frame.

**Usage**

```
plot_boxplot(data, label_x = "", label_y = "", colors = NULL, barwidth = 0.25)
```

**Arguments**

|          |                                             |
|----------|---------------------------------------------|
| data     | data.frame with one or more numeric columns |
| label_x  | x-axis label                                |
| label_y  | y-axis label                                |
| colors   | optional fill color for boxes               |
| barwidth | width of the box (numeric)                  |

**Details**

The data is melted to long format and a box is drawn per original column. If colors is provided, a constant fill is applied to all boxes. Use barwidth to control box width.

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
grf <- plot_boxplot(iris, colors="white")
plot(grf)
```

---

|                    |                          |
|--------------------|--------------------------|
| plot_boxplot_class | <i>Boxplot per class</i> |
|--------------------|--------------------------|

---

**Description**

Boxplots of a numeric column grouped by a class label.

**Usage**

```
plot_boxplot_class(
  data,
  class_label,
  label_x = "",
  label_y = "",
  colors = NULL
)
```

**Arguments**

|             |                                                          |
|-------------|----------------------------------------------------------|
| data        | data.frame with a grouping column and one numeric column |
| class_label | name of the grouping (class) column                      |
| label_x     | x-axis label                                             |
| label_y     | y-axis label                                             |
| colors      | optional fill color for the boxes                        |

**Details**

Expects a data.frame with the grouping column named in `class_label` and one numeric column. The function melts to long format and draws per-group distributions.

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
grf <- plot_boxplot_class(iris |> dplyr::select(Sepal.Width, Species),
  class_label = "Species", colors=c("red", "green", "blue"))
plot(grf)
```

---

|                  |                         |
|------------------|-------------------------|
| plot_correlation | <i>Plot correlation</i> |
|------------------|-------------------------|

---

## Description

Correlation heatmap with optional labels and triangle filtering.

## Usage

```
plot_correlation(  
  df,  
  vars = NULL,  
  method = c("pearson", "spearman", "kendall"),  
  use = "pairwise.complete.obs",  
  triangle = c("full", "upper", "lower"),  
  reorder = c("none", "hclust", "alphabetical"),  
  digits = 2,  
  label_size = 3,  
  tile_color = "white",  
  show_diag = TRUE,  
  title = NULL  
)
```

## Arguments

|            |                                                          |
|------------|----------------------------------------------------------|
| df         | data.frame with numeric columns                          |
| vars       | optional vector of column names to include               |
| method     | correlation method: "pearson", "spearman", or "kendall"  |
| use        | handling of missing values for <code>stats::cor</code>   |
| triangle   | which triangle to show: "full", "upper", or "lower"      |
| reorder    | reordering strategy: "none", "hclust", or "alphabetical" |
| digits     | number of digits for labels                              |
| label_size | size of label text                                       |
| tile_color | border color for tiles                                   |
| show_diag  | whether to show the diagonal                             |
| title      | optional plot title                                      |

## Details

Computes a correlation matrix from numeric columns (or `vars`) and renders a `ggplot2` heatmap with values annotated. Supports reordering by hierarchical clustering or alphabetically.

## Value

returns a `ggplot2::ggplot` graphic

**Examples**

```
data(iris)
grf <- plot_correlation(iris[,1:4])
plot(grf)
```

---

|                 |                        |
|-----------------|------------------------|
| plot_dendrogram | <i>Plot dendrogram</i> |
|-----------------|------------------------|

---

**Description**

Dendrogram plot for an hclust or dendrogram object using ggplot2.

**Usage**

```
plot_dendrogram(hc, labels = TRUE, label_size = 3, title = NULL)
```

**Arguments**

|            |                                         |
|------------|-----------------------------------------|
| hc         | an object of class hclust or dendrogram |
| labels     | logical; whether to draw leaf labels    |
| label_size | label text size                         |
| title      | optional plot title                     |

**Details**

Converts a dendrogram into line segments and renders it with ggplot2.

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
data(iris)
hc <- hclust(dist(scale(iris[,1:4])), method = "ward.D2")
grf <- plot_dendrogram(hc)
plot(grf)
```



---

|              |                     |
|--------------|---------------------|
| plot_density | <i>Plot density</i> |
|--------------|---------------------|

---

## Description

Kernel density plot for one or multiple numeric columns.

## Usage

```
plot_density(  
  data,  
  label_x = "",  
  label_y = "",  
  colors = NULL,  
  bin = NULL,  
  alpha = 0.25  
)
```

## Arguments

|         |                                                          |
|---------|----------------------------------------------------------|
| data    | data.frame with one or more numeric columns              |
| label_x | x-axis label                                             |
| label_y | y-axis label                                             |
| colors  | optional fill color (single column) or vector for groups |
| bin     | optional bin width passed to geom_density                |
| alpha   | fill transparency (0–1)                                  |

## Details

If data has multiple numeric columns, densities are overlaid and filled by column (group). When a single column is provided, colors (if set) is used as a constant fill. The bin argument is passed to `geom_density(binwidth=...)`.

## Value

returns a `ggplot2::ggplot` graphic

## Examples

```
grf <- plot_density(iris |> dplyr::select(Sepal.Width), colors="blue")  
plot(grf)
```

---

|                    |                               |
|--------------------|-------------------------------|
| plot_density_class | <i>Plot density per class</i> |
|--------------------|-------------------------------|

---

**Description**

Kernel density plot grouped by a class label.

**Usage**

```
plot_density_class(  
  data,  
  class_label,  
  label_x = "",  
  label_y = "",  
  colors = NULL,  
  bin = NULL,  
  alpha = 0.5  
)
```

**Arguments**

|             |                                                  |
|-------------|--------------------------------------------------|
| data        | data.frame with class label and a numeric column |
| class_label | name of the grouping (class) column              |
| label_x     | x-axis label                                     |
| label_y     | y-axis label                                     |
| colors      | optional vector of fills per class               |
| bin         | optional bin width passed to geom_density        |
| alpha       | fill transparency (0–1)                          |

**Details**

Expects data with a grouping column named in `class_label` and one numeric column. Each group is filled with a distinct color (if provided).

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
grf <- plot_density_class(iris |> dplyr::select(Sepal.Width, Species),  
  class = "Species", colors=c("red", "green", "blue"))  
plot(grf)
```

---

|                 |                         |
|-----------------|-------------------------|
| plot_groupedbar | <i>Plot grouped bar</i> |
|-----------------|-------------------------|

---

## Description

Grouped (side-by-side) bar chart for multiple series per category.

## Usage

```
plot_groupedbar(data, label_x = "", label_y = "", colors = NULL, alpha = 1)
```

## Arguments

|         |                                                                          |
|---------|--------------------------------------------------------------------------|
| data    | data.frame with category in first column and series in remaining columns |
| label_x | x-axis label                                                             |
| label_y | y-axis label                                                             |
| colors  | optional vector of fill colors, one per series                           |
| alpha   | bar transparency (0–1)                                                   |

## Details

Expects a data.frame where the first column is the category (x) and the remaining columns are numeric series. Bars are grouped by series. Provide colors with length equal to the number of series to set fills.

## Value

returns a ggplot2::ggplot graphic

## Examples

```
#summarizing iris dataset
data <- iris |> dplyr::group_by(Species) |>
dplyr::summarize(Sepal.Length=mean(Sepal.Length), Sepal.Width=mean(Sepal.Width))
head(data)

#plotting data
grf <- plot_groupedbar(data, colors=c("blue", "red"))
plot(grf)
```

---

|                        |                       |
|------------------------|-----------------------|
| <code>plot_hist</code> | <i>Plot histogram</i> |
|------------------------|-----------------------|

---

**Description**

Histogram for a numeric column using ggplot2.

**Usage**

```
plot_hist(data, label_x = "", label_y = "", color = "white", alpha = 0.25)
```

**Arguments**

|                      |                                                                       |
|----------------------|-----------------------------------------------------------------------|
| <code>data</code>    | data.frame with one numeric column (first column is used if multiple) |
| <code>label_x</code> | x-axis label                                                          |
| <code>label_y</code> | y-axis label                                                          |
| <code>color</code>   | fill color                                                            |
| <code>alpha</code>   | transparency level (0–1)                                              |

**Details**

If multiple columns are provided, only the first is used. Breaks are computed via `graphics::hist` to mirror base R binning. `color` controls the fill; `alpha` the transparency.

**Value**

returns a `ggplot2::ggplot` graphic

**Examples**

```
grf <- plot_hist(iris |> dplyr::select(Sepal.Width), color=c("blue"))
plot(grf)
```

---

|                            |                      |
|----------------------------|----------------------|
| <code>plot_lollipop</code> | <i>Plot lollipop</i> |
|----------------------------|----------------------|

---

**Description**

Lollipop chart (stick + circle + value label) per category.

**Usage**

```
plot_lollipop(
  data,
  label_x = "",
  label_y = "",
  colors = NULL,
  color_text = "black",
  size_text = 3,
  size_ball = 8,
  alpha_ball = 0.2,
  min_value = 0,
  max_value_gap = 1
)
```

**Arguments**

|               |                                             |
|---------------|---------------------------------------------|
| data          | data.frame with category and numeric values |
| label_x       | x-axis label                                |
| label_y       | y-axis label                                |
| colors        | stick/circle color                          |
| color_text    | color of the text inside the circle         |
| size_text     | text size                                   |
| size_ball     | circle size                                 |
| alpha_ball    | circle transparency (0–1)                   |
| min_value     | minimum baseline for the stick              |
| max_value_gap | gap from value to stick end                 |

**Details**

Expects a data.frame with category in the first column and numeric values in subsequent columns.  
Circles are drawn at values, with vertical segments extending from min\_value to value - max\_value\_gap.

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
#summarizing iris dataset
data <- iris |> dplyr::group_by(Species) |>
dplyr::summarize(Sepal.Length=mean(Sepal.Length))
head(data)

#plotting data
grf <- plot_lollipop(data, colors="blue", max_value_gap=0.2)
plot(grf)
```

---

|           |                            |
|-----------|----------------------------|
| plot_pair | <i>Plot scatter matrix</i> |
|-----------|----------------------------|

---

**Description**

Scatter matrix using GGally::ggpairs with optional class coloring.

**Usage**

```
plot_pair(data, cnames, title = NULL, clabel = NULL, colors = NULL)
```

**Arguments**

|        |                                       |
|--------|---------------------------------------|
| data   | data.frame                            |
| cnames | column names to include               |
| title  | optional title                        |
| clabel | optional class label column name      |
| colors | optional vector of colors for classes |

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
data(iris)
grf <- plot_pair(iris, cnames = colnames(iris)[1:4], title = "Iris")
print(grf)
```

---

|               |                                     |
|---------------|-------------------------------------|
| plot_pair_adv | <i>Plot advanced scatter matrix</i> |
|---------------|-------------------------------------|

---

**Description**

Scatter matrix with class coloring and manual palette application.

**Usage**

```
plot_pair_adv(data, cnames, title = NULL, clabel = NULL, colors = NULL)
```

**Arguments**

|        |                                       |
|--------|---------------------------------------|
| data   | data.frame                            |
| cnames | column names to include               |
| title  | optional title                        |
| clabel | optional class label column name      |
| colors | optional vector of colors for classes |

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
data(iris)
grf <- plot_pair_adv(iris, cnames = colnames(iris)[1:4], title = "Iris")
print(grf)
```

---

|               |                                  |
|---------------|----------------------------------|
| plot_parallel | <i>Plot parallel coordinates</i> |
|---------------|----------------------------------|

---

**Description**

Parallel coordinates plot using GGally::ggparcoord.

**Usage**

```
plot_parallel(data, columns, group, colors = NULL, title = NULL)
```

**Arguments**

|         |                                               |
|---------|-----------------------------------------------|
| data    | data.frame                                    |
| columns | numeric columns to include (indices or names) |
| group   | grouping column (index or name)               |
| colors  | optional vector of colors for groups          |
| title   | optional title                                |

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
data(iris)
grf <- plot_parallel(iris, columns = 1:4, group = 5)
plot(grf)
```

plot\_pieplot

*Plot pie***Description**

Pie chart from a two-column data.frame (category, value) using polar coordinates.

**Usage**

```
plot_pieplot(
  data,
  label_x = "",
  label_y = "",
  colors = NULL,
  textcolor = "white",
  bordercolor = "black"
)
```

**Arguments**

|             |                                                 |
|-------------|-------------------------------------------------|
| data        | two-column data.frame with category and value   |
| label_x     | x-axis label (unused in pie, kept for symmetry) |
| label_y     | y-axis label (unused in pie)                    |
| colors      | vector of slice fills                           |
| textcolor   | label text color                                |
| bordercolor | slice border color                              |

**Details**

Slices are sized by the second (numeric) column. Text and border colors can be customized.

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
#summarizing iris dataset
data <- iris |> dplyr::group_by(Species) |>
dplyr::summarize(Sepal.Length=mean(Sepal.Length))
head(data)

#ploting data
grf <- plot_pieplot(data, colors=c("red", "green", "blue"))
plot(grf)
```



---

|            |                                 |
|------------|---------------------------------|
| plot_pixel | <i>Plot pixel visualization</i> |
|------------|---------------------------------|

---

## Description

Pixel-oriented visualization of a numeric matrix or data.frame.

## Usage

```
plot_pixel(  
  data,  
  colors = NULL,  
  title = NULL,  
  label_x = "sample",  
  label_y = "Attributes"  
)
```

## Arguments

|         |                                                 |
|---------|-------------------------------------------------|
| data    | numeric matrix or data.frame                    |
| colors  | optional vector of colors for the fill gradient |
| title   | optional plot title                             |
| label_x | x-axis label                                    |
| label_y | y-axis label                                    |

## Details

Renders a heatmap-like plot where each cell is a pixel. Useful for multivariate inspection.

## Value

returns a ggplot2::ggplot graphic

## Examples

```
data(iris)  
grf <- plot_pixel(as.matrix(iris[,1:4]), title = "Iris")  
plot(grf)
```

---

|             |                    |
|-------------|--------------------|
| plot_points | <i>Plot points</i> |
|-------------|--------------------|

---

## Description

Dot chart for multiple series across categories (points only).

## Usage

```
plot_points(data, label_x = "", label_y = "", colors = NULL)
```

## Arguments

|         |                                                        |
|---------|--------------------------------------------------------|
| data    | data.frame with category + one or more numeric columns |
| label_x | x-axis label                                           |
| label_y | y-axis label                                           |
| colors  | optional color vector for series                       |

## Details

Expects a data.frame with category in the first column and one or more numeric series. Points are colored by series (legend shows original column names). Supply colors to override the palette.

## Value

returns a ggplot2::ggplot graphic

## Examples

```
x <- seq(0, 10, 0.25)
data <- data.frame(x, sin=sin(x), cosine=cos(x)+5)
head(data)

grf <- plot_points(data, colors=c("red", "green"))
plot(grf)
```

---

`plot_radar`*Plot radar*

---

## Description

Radar (spider) chart for a single profile of variables using radial axes.

## Usage

```
plot_radar(data, label_x = "", label_y = "", colors = NULL)
```

## Arguments

|                      |                                                                   |
|----------------------|-------------------------------------------------------------------|
| <code>data</code>    | two-column data.frame: variable name and value                    |
| <code>label_x</code> | x-axis label (unused; variable names are shown around the circle) |
| <code>label_y</code> | y-axis label                                                      |
| <code>colors</code>  | line/fill color for the polygon                                   |

## Details

Expects a two-column data.frame with variable names in the first column and numeric values in the second. The graphic is built as an n-sided polygon, where n is the number of variables, so at least three variables are required. The function already sets the drawing limits for the full polygon; adding `ylim()` or other Cartesian clipping after the fact can hide part of the radar.

## Value

returns a `ggplot2::ggplot` graphic

## Examples

```
data <- data.frame(name = "Petal.Length", value = mean(iris$Petal.Length))
data <- rbind(data, data.frame(name = "Petal.Width", value = mean(iris$Petal.Width)))
data <- rbind(data, data.frame(name = "Sepal.Length", value = mean(iris$Sepal.Length)))
data <- rbind(data, data.frame(name = "Sepal.Width", value = mean(iris$Sepal.Width)))

grf <- plot_radar(data, colors = "red")
plot(grf)
```

---

|              |               |
|--------------|---------------|
| plot_scatter | Scatter graph |
|--------------|---------------|

---

**Description**

Scatter plot from a long data.frame with columns named x, value, and variable.

**Usage**

```
plot_scatter(data, label_x = "", label_y = "", colors = NULL)
```

**Arguments**

|         |                                                                            |
|---------|----------------------------------------------------------------------------|
| data    | long data.frame with columns x, value, variable                            |
| label_x | x-axis label                                                               |
| label_y | y-axis label                                                               |
| colors  | optional color(s); for numeric variable, supply a gradient as c(low, high) |

**Details**

Colors are mapped to variable. If variable is numeric, a gradient color scale is used when colors is provided.

**Value**

return a ggplot2::ggplot graphic

**Examples**

```
grf <- plot_scatter(iris |> dplyr::select(x = Sepal.Length,
value = Sepal.Width, variable = Species),
label_x = "Sepal.Length", label_y = "Sepal.Width",
colors=c("red", "green", "blue"))
plot(grf)
```

---

|             |             |
|-------------|-------------|
| plot_series | Plot series |
|-------------|-------------|

---

**Description**

Line plot for one or more series over a common x index.

**Usage**

```
plot_series(data, label_x = "", label_y = "", colors = NULL)
```

**Arguments**

|         |                                                                       |
|---------|-----------------------------------------------------------------------|
| data    | data.frame with x in the first column and series in remaining columns |
| label_x | x-axis label                                                          |
| label_y | y-axis label                                                          |
| colors  | optional vector of colors for series                                  |

**Details**

Expects a data.frame where the first column is the x index and remaining columns are numeric series. Points and lines are drawn per series; supply colors to override the palette.

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
x <- seq(0, 10, 0.25)
data <- data.frame(x, sin=sin(x))
head(data)

grf <- plot_series(data, colors=c("red"))
plot(grf)
```

---

|                 |                         |
|-----------------|-------------------------|
| plot_stackedbar | <i>Plot stacked bar</i> |
|-----------------|-------------------------|

---

**Description**

Stacked bar chart for multiple series per category.

**Usage**

```
plot_stackedbar(data, label_x = "", label_y = "", colors = NULL, alpha = 1)
```

**Arguments**

|         |                                                                          |
|---------|--------------------------------------------------------------------------|
| data    | data.frame with category in first column and series in remaining columns |
| label_x | x-axis label                                                             |
| label_y | y-axis label                                                             |
| colors  | optional vector of fill colors, one per series                           |
| alpha   | bar transparency (0–1)                                                   |

**Details**

Expects a data.frame with category in the first column and series in remaining columns. Bars are stacked within each category. Provide colors (one per series) to control fills.

**Value**

returns a `ggplot2::ggplot` graphic

**Examples**

```
#summarizing iris dataset
data <- iris |> dplyr::group_by(Species) |>
dplyr::summarize(Sepal.Length=mean(Sepal.Length), Sepal.Width=mean(Sepal.Width))

#plotting data
grf <- plot_stackedbar(data, colors=c("blue", "red"))
plot(grf)
```

---

plot\_ts

*Plot time series chart*


---

**Description**

Simple time series plot with points and a line.

**Usage**

```
plot_ts(x = NULL, y, label_x = "", label_y = "", color = "black")
```

**Arguments**

|         |                                                        |
|---------|--------------------------------------------------------|
| x       | time index (numeric vector) or NULL to use 1:length(y) |
| y       | numeric series                                         |
| label_x | x-axis label                                           |
| label_y | y-axis label                                           |
| color   | color for the series                                   |

**Details**

If x is NULL, an integer index 1:n is used. The color applies to both points and line.

**Value**

returns a `ggplot2::ggplot` graphic

**Examples**

```
x <- seq(0, 10, 0.25)
y <- sin(x)

grf <- plot_ts(x = x, y = y, color=c("red"))
plot(grf)
```

---

plot\_ts\_pred

---

*Plot time series with predictions*

---

**Description**

Plot original series plus dashed lines for in-sample adjustment and optional out-of-sample predictions.

**Usage**

```
plot_ts_pred(  
  x = NULL,  
  y,  
  yadj,  
  ypred = NULL,  
  label_x = "",  
  label_y = "",  
  color = "black",  
  color_adjust = "blue",  
  color_prediction = "green"  
)
```

**Arguments**

|                  |                                                        |
|------------------|--------------------------------------------------------|
| x                | time index (numeric vector) or NULL to use 1:length(y) |
| y                | numeric time series                                    |
| yadj             | fitted/adjusted values for the training window         |
| ypred            | optional predicted values after the training window    |
| label_x          | x-axis title                                           |
| label_y          | y-axis title                                           |
| color            | color for the original series                          |
| color_adjust     | color for the adjusted values (dashed)                 |
| color_prediction | color for the predictions (dashed)                     |

**Details**

yadj length defines the training segment; ypred (if provided) is appended after yadj.

**Value**

returns a ggplot2::ggplot graphic

**Examples**

```
x <- base::seq(0, 10, 0.25)
yvalues <- sin(x) + rnorm(41,0,0.1)
adjust <- sin(x[1:35])
prediction <- sin(x[36:41])
grf <- plot_ts_pred(y=yvalues, yadj=adjust, ypred=prediction)
plot(grf)
```

---

|           |                                                       |
|-----------|-------------------------------------------------------|
| predictor | <i>Predictor (base for classification/regression)</i> |
|-----------|-------------------------------------------------------|

---

**Description**

Ancestor class for supervised predictors (classification and regression). Provides a default `fit()` to record feature names and proxies `action()` to `predict()`.  
An example predictor is a decision tree classifier (`cla_dtree`).

**Usage**

```
predictor()
```

**Value**

returns a predictor object

**Examples**

```
#See ?cla_dtree for a classification example using a decision tree
```

---

|                       |                                   |
|-----------------------|-----------------------------------|
| predictor_prepare_fit | <i>Prepare predictor fit data</i> |
|-----------------------|-----------------------------------|

---

**Description**

Normalizes supervised fit input and records predictor columns by removing the target attribute from the data columns.

**Usage**

```
predictor_prepare_fit(obj, data)
```

**Arguments**

- obj                predictor-like object containing attribute
- data              training data



**Value**

returns a list with the updated object and normalized data

---

|            |                              |
|------------|------------------------------|
| regression | <i>Regression base class</i> |
|------------|------------------------------|

---

**Description**

Ancestor class for regression models. Stores the target attribute and provides common evaluation metrics.

**Usage**

```
regression(attribute)
```

**Arguments**

attribute      attribute target to model building

**Value**

returns a regression object

**Examples**

```
#See ?reg_dtree for a regression example using a decision tree
```

---

|           |                                     |
|-----------|-------------------------------------|
| reg_dtree | <i>Decision Tree for regression</i> |
|-----------|-------------------------------------|

---

**Description**

Regression tree using recursive partitioning via the `tree` package.

**Usage**

```
reg_dtree(attribute)
```

**Arguments**

attribute      attribute target to model building.

**Details**

Splits are chosen to reduce squared error within nodes; result is an interpretable set of piecewise constants.

**Value**

returns a decision tree regression object

**References**

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). Classification and Regression Trees. Wadsworth.

**Examples**

```
data(Boston)
model <- reg_dtree("medv")

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, Boston)
train <- sr$train
test <- sr$test

model <- fit(model, train)

test_prediction <- predict(model, test)
test_predictand <- test[, "medv"]
test_eval <- evaluate(model, test_predictand, test_prediction)
test_eval$metrics
```

---

reg\_knn

---

*K-Nearest Neighbors (KNN) Regression*


---

**Description**

KNN regression using `FNN::knn.reg`, predicting by averaging the targets of the  $k$  nearest neighbors.

**Usage**

```
reg_knn(attribute, k)
```

**Arguments**

|           |                                    |
|-----------|------------------------------------|
| attribute | attribute target to model building |
| k         | number of $k$ neighbors            |

**Details**

Non-parametric approach suitable for local smoothing. Sensitive to feature scaling; consider normalization beforehand.

**Value**

returns a knn regression object

**References**

Altman, N. (1992). An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression.

**Examples**

```
data(Boston)
model <- reg_knn("medv", k=3)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, Boston)
train <- sr$train
test <- sr$test

model <- fit(model, train)

test_prediction <- predict(model, test)
test_predictand <- test[, "medv"]
test_eval <- evaluate(model, test_predictand, test_prediction)
test_eval$metrics
```

---

reg\_lm

*Linear regression (lm)*


---

**Description**

Linear regression using stats::lm.

**Usage**

```
reg_lm(formula = NULL, attribute = NULL, features = NULL)
```

**Arguments**

|           |                                                              |
|-----------|--------------------------------------------------------------|
| formula   | optional regression formula (e.g., $y \sim x_1 + x_2$ ).     |
| attribute | target attribute name (used when formula is NULL)            |
| features  | optional vector of feature names (used when formula is NULL) |

**Value**

returns a reg\_lm object

## Examples

```
if (requireNamespace("MASS", quietly = TRUE)) {
  data(Boston, package = "MASS")

  # Simple linear regression
  model_simple <- reg_lm(formula = medv ~ lstat)
  model_simple <- fit(model_simple, Boston)
  pred_simple <- predict(model_simple, Boston)
  eval_simple <- evaluate(model_simple, Boston$medv, pred_simple)
  eval_simple$metrics

  # Polynomial regression (degree 2)
  model_poly <- reg_lm(formula = medv ~ poly(lstat, 2, raw = TRUE))
  model_poly <- fit(model_poly, Boston)
  pred_poly <- predict(model_poly, Boston)
  eval_poly <- evaluate(model_poly, Boston$medv, pred_poly)
  eval_poly$metrics

  # Multiple regression
  model_multi <- reg_lm(formula = medv ~ lstat + rm + ptratio)
  model_multi <- fit(model_multi, Boston)
  pred_multi <- predict(model_multi, Boston)
  eval_multi <- evaluate(model_multi, Boston$medv, pred_multi)
  eval_multi$metrics
}
```

---

reg\_mlp

MLP for regression

---

## Description

Multi-Layer Perceptron regression using `nnet::nnet` (single hidden layer).

## Usage

```
reg_mlp(attribute, size = NULL, decay = 0.05, maxit = 1000)
```

## Arguments

|           |                                           |
|-----------|-------------------------------------------|
| attribute | attribute target to model building        |
| size      | number of neurons in hidden layers        |
| decay     | decay learning rate                       |
| maxit     | number of maximum iterations for training |

## Details

Feedforward neural network with size hidden units and L2 regularization controlled by decay. Data should be scaled for stable training.

**Value**

returns a object of class reg\_mlp

**References**

Bishop, C. M. (1995). Neural Networks for Pattern Recognition. Oxford University Press.

**Examples**

```
data(Boston)
model <- reg_mlp("medv", size=5, decay=0.54)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, Boston)
train <- sr$train
test <- sr$test

model <- fit(model, train)

test_prediction <- predict(model, test)
test_predictand <- test[, "medv"]
test_eval <- evaluate(model, test_predictand, test_prediction)
test_eval$metrics
```

---

reg\_rf

*Random Forest for regression*

---

**Description**

Regression via Random Forests, an ensemble of decision trees trained on bootstrap samples with random feature subsetting at each split. This wrapper uses the randomForest package API.

**Usage**

```
reg_rf(attribute, nodesize = 1, ntree = 10, mtry = NULL)
```

**Arguments**

|           |                                    |
|-----------|------------------------------------|
| attribute | attribute target to model building |
| nodesize  | node size                          |
| ntree     | number of trees                    |
| mtry      | number of attributes to build tree |

**Details**

Random Forests reduce variance and are robust to overfitting on tabular data. Key hyperparameters are the number of trees (ntree), the number of variables tried at each split (mtry), and the minimum node size (nodesize).

**Value**

returns an object of class `reg_rfobj`

**References**

Breiman, L. (2001). Random Forests. *Machine Learning* 45(1):5–32. Liaw, A. and Wiener, M. (2002). Classification and Regression by randomForest. *R News*.

**Examples**

```
data(Boston)
model <- reg_rf("medv", ntree=10)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, Boston)
train <- sr$train
test <- sr$test

model <- fit(model, train)

test_prediction <- predict(model, test)
test_predictand <- test[, "medv"]
test_eval <- evaluate(model, test_predictand, test_prediction)
test_eval$metrics
```

---

reg\_svm

*SVM for regression*


---

**Description**

Support Vector Regression (SVR) using `e1071::svm`.

**Usage**

```
reg_svm(
  attribute,
  epsilon = 0.1,
  cost = 10,
  kernel = c("radial", "linear", "polynomial", "sigmoid")
)
```

**Arguments**

|           |                                                                                                                   |
|-----------|-------------------------------------------------------------------------------------------------------------------|
| attribute | attribute target to model building                                                                                |
| epsilon   | parameter that controls the width of the margin around the separating hyperplane                                  |
| cost      | parameter that controls the trade-off between having a wide margin and correctly classifying training data points |

**kernel** the type of kernel function to be used in the SVM algorithm (linear, radial, polynomial, sigmoid)

### Details

SVR optimizes a margin with an epsilon-insensitive loss around the regression function. The cost controls regularization strength; epsilon sets the width of the insensitive tube; and kernel defines the feature map (linear, radial, polynomial, sigmoid).

### Value

returns a SVM regression object

### References

Drucker, H., Burges, C., Kaufman, L., Smola, A., Vapnik, V. (1997). Support Vector Regression Machines. Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines.

### Examples

```
data(Boston)
model <- reg_svm("medv", epsilon=0.2, cost=40.000)

# preparing dataset for random sampling
sr <- sample_random()
sr <- train_test(sr, Boston)
train <- sr$train
test <- sr$test

model <- fit(model, train)

test_prediction <- predict(model, test)
test_predictand <- test[, "medv"]
test_eval <- evaluate(model, test_predictand, test_prediction)
test_eval$metrics
```

---

|          |                                      |
|----------|--------------------------------------|
| reg_tune | <i>Regression tuning (k-fold CV)</i> |
|----------|--------------------------------------|

---

### Description

Tune hyperparameters of a base regressor via k-fold cross-validation minimizing an error metric (MSE).

### Usage

```
reg_tune(base_model, folds = 10, ranges = NULL)
```

**Arguments**

- base\_model      base model for tuning
- folds            number of folds for cross-validation
- ranges           a list of hyperparameter ranges to explore

**Value**

returns a reg\_tune object.

**References**

Kohavi, R. (1995). A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection.

**Examples**

```
# preparing dataset for random sampling
data(Boston)
sr <- sample_random()
sr <- train_test(sr, Boston)
train <- sr$train
test <- sr$test

# hyper parameter setup
tune <- reg_tune(reg_mlp("medv"), folds=3, ranges = list(size=c(3), decay=c(0.1,0.5)))

# hyper parameter optimization
model <- fit(tune, train)

test_prediction <- predict(model, test)
test_predictand <- test[, "medv"]
test_eval <- evaluate(model, test_predictand, test_prediction)
test_eval$metrics
```

---

|                |                                           |
|----------------|-------------------------------------------|
| sample_balance | <i>Class balancing (up/down sampling)</i> |
|----------------|-------------------------------------------|

---

**Description**

Balance class distribution using up-sampling or down-sampling.

**Usage**

```
sample_balance(attribute, method = c("down", "up"))
```

**Arguments**

- attribute      target class attribute name
- method          balancing method: "down" or "up"



**Value**

returns an object of class sample\_balance

**Examples**

```
data(iris)
iris_imb <- iris[iris$Species != "setosa", ]
sb <- sample_balance("Species", method = "down")
iris_bal <- transform(sb, iris_imb)
table(iris_bal$Species)
```

---

sample\_groups

*Group sampling*

---

**Description**

Sample entire groups defined by a categorical attribute. In sampling theory, this design is known as cluster sampling (also called one-stage cluster sampling or sampling by groups). The groups are assumed to be pre-defined in the data; this function does not infer groups with clustering algorithms such as k-means.

**Usage**

```
sample_groups(attribute, n_groups)
```

**Arguments**

|           |                               |
|-----------|-------------------------------|
| attribute | group-defining attribute name |
| n_groups  | number of groups to sample    |

**Value**

returns an object of class sample\_groups

**Examples**

```
data(iris)
sc <- sample_groups("Species", n_groups = 2)
iris_sc <- transform(sc, iris)
table(iris_sc$Species)
```

---

|               |                        |
|---------------|------------------------|
| sample_random | <i>Random Sampling</i> |
|---------------|------------------------|

---

**Description**

Train/test split and k-fold partitioning by simple random sampling.

**Usage**

```
sample_random()
```

**Details**

This is the simplest partitioning strategy in daltoolbox. It is a good starting point when the dataset is reasonably balanced and the user wants to understand the train/test workflow before moving to stratified or more specialized sampling schemes.

**Value**

An object of class sample\_random.

**Examples**

```
# using random sampling
sample <- sample_random()
tt <- train_test(sample, iris)

# distribution of train
table(tt$train$Species)

# preparing dataset into four folds
folds <- k_fold(sample, iris, 4)

# distribution of folds
tbl <- NULL
for (f in folds) {
  tbl <- rbind(tbl, table(f$Species))
}
head(tbl)
```

---

|               |                        |
|---------------|------------------------|
| sample_simple | <i>Simple sampling</i> |
|---------------|------------------------|

---

**Description**

Sample rows or elements with or without replacement.

**Usage**

```
sample_simple(size, replace = FALSE, prob = NULL)
```

**Arguments**

|         |                                           |
|---------|-------------------------------------------|
| size    | number of samples to draw                 |
| replace | logical; sample with replacement if TRUE  |
| prob    | optional vector of sampling probabilities |

**Value**

returns an object of class sample\_simple

**Examples**

```
data(iris)
srswor <- sample_simple(size = 10, replace = FALSE)
srswr <- sample_simple(size = 10, replace = TRUE)
sample_wor <- transform(srswor, iris$Sepal.Length)
sample_wr <- transform(srswr, iris$Sepal.Length)
sample_wor
sample_wr
```

---

|                   |                            |
|-------------------|----------------------------|
| sample_stratified | <i>Stratified Sampling</i> |
|-------------------|----------------------------|

---

**Description**

Train/test split and k-fold partitioning that preserve the target class proportions.

**Usage**

```
sample_stratified(attribute)
```

**Arguments**

|           |                                                                           |
|-----------|---------------------------------------------------------------------------|
| attribute | Name of the target attribute whose class proportions should be preserved. |
|-----------|---------------------------------------------------------------------------|

**Details**

Use this sampler when the response distribution matters, especially in imbalanced classification problems. Compared with simple random sampling, it reduces the chance that one split will over-represent or underrepresent a class by accident.

**Value**

An object of class `sample_stratified`.

**Examples**

```
# using stratified sampling
sample <- sample_stratified("Species")
tt <- train_test(sample, iris)

# distribution of train
table(tt$train$Species)

# preparing dataset into four folds
folds <- k_fold(sample, iris, 4)

# distribution of folds
tbl <- NULL
for (f in folds) {
  tbl <- rbind(tbl, table(f$Species))
}
head(tbl)
```

---

select\_hyper

*Selection of hyperparameters*


---

**Description**

Generic to select the best hyperparameters from cross-validation results; subclasses can override.

**Usage**

```
select_hyper(obj, hyperparameters)
```

**Arguments**

`obj` the object or model used for hyperparameter selection.  
`hyperparameters` data set with hyper parameters and quality measure from execution

**Value**

returns the index of selected hyper parameter

---

|                       |                                     |
|-----------------------|-------------------------------------|
| select_hyper.cla_tune | <i>selection of hyperparameters</i> |
|-----------------------|-------------------------------------|

---

**Description**

Selects the optimal hyperparameter by maximizing the average classification metric. It wraps dplyr library.

**Usage**

```
## S3 method for class 'cla_tune'  
select_hyper(obj, hyperparameters)
```

**Arguments**

- obj                    an object representing the model or tuning process
- hyperparameters      a dataframe with columns key (hyperparameter configuration) and metric (classification metric)

**Value**

returns a optimized key number of hyperparameters

---

|            |                          |
|------------|--------------------------|
| set_params | <i>Assign parameters</i> |
|------------|--------------------------|

---

**Description**

Assign a named list of parameters to matching fields in the object (best-effort).

**Usage**

```
set_params(obj, params)
```

**Arguments**

- obj                    object of class dal\_base
- params                parameters to set obj

**Value**

returns an object with parameters set

**Examples**

```
obj <- set_params(dal_base(), list(x = 0))
```

---

|                    |                                  |
|--------------------|----------------------------------|
| set_params.default | <i>Default Assign parameters</i> |
|--------------------|----------------------------------|

---

### Description

Default method for set\_params (returns object unchanged).

### Usage

```
## Default S3 method:
set_params(obj, params)
```

### Arguments

|        |            |
|--------|------------|
| obj    | object     |
| params | parameters |

### Value

returns the object unchanged

---

|           |                                         |
|-----------|-----------------------------------------|
| smoothing | <i>Smoothing (binning/quantization)</i> |
|-----------|-----------------------------------------|

---

### Description

Family of smoothing methods that reduce noise by replacing values with the mean of a bin. Supported strategies include equal-interval bins, equal-frequency (quantile) bins, k-means quantization, and class-aware clustering.

### Usage

```
smoothing(n)
```

### Arguments

|   |                |
|---|----------------|
| n | number of bins |
|---|----------------|

### Details

The smoothing level is controlled by n (number of bins/levels). The base helper tune() chooses n by locating the elbow (maximum curvature) of the MSE curve across candidates. Concrete sub-classes may override that criterion when supervision is required. After fit(), values are mapped to bin means via transform().

**Value**

returns an object of class `smoothing`

**Examples**

```
data(iris)
obj <- smoothing_inter(n = 2)
obj <- fit(obj, iris$Sepal.Length)
sl.bi <- transform(obj, iris$Sepal.Length)
table(sl.bi)
obj$interval

bins <- cut(iris$Sepal.Length, unique(obj$interval.adj), FALSE, include.lowest = TRUE)
entro <- evaluate(obj, bins, iris$Species)
entro$entropy
```

---

|                                |                                            |
|--------------------------------|--------------------------------------------|
| <code>smoothing_cluster</code> | <i>Smoothing by class-aware clustering</i> |
|--------------------------------|--------------------------------------------|

---

**Description**

Discretize a numeric attribute into `n` bins by clustering the attribute together with a one-hot representation of the class label, then projecting the clusters back to ordered cut points on the numeric axis.

**Usage**

```
smoothing_cluster(class_label, n)
```

**Arguments**

|                          |                             |
|--------------------------|-----------------------------|
| <code>class_label</code> | name of the class attribute |
| <code>n</code>           | number of bins              |

**Value**

returns an object of class `smoothing_cluster`

**References**

Han, J., Kamber, M., Pei, J. (2011). Data Mining: Concepts and Techniques. (Discretization)

**Examples**

```

data(iris)
cluster_data <- iris[, c("Sepal.Length", "Species")]
obj <- smoothing_cluster("Species", n = 2)
obj <- fit(obj, cluster_data)
sl.bi <- transform(obj, iris$Sepal.Length)
table(sl.bi)
obj$interval

bins <- cut(iris$Sepal.Length, unique(obj$interval.adj), FALSE, include.lowest = TRUE)
entro <- evaluate(obj, bins, iris$Species)
entro$entropy

```

---

|                |                                     |
|----------------|-------------------------------------|
| smoothing_freq | <i>Smoothing by equal frequency</i> |
|----------------|-------------------------------------|

---

**Description**

Discretize a numeric vector into  $n$  bins with approximately equal frequency (quantile cuts), and replace each value by the mean of its bin.

**Usage**

```
smoothing_freq(n)
```

**Arguments**

|                |                |
|----------------|----------------|
| <code>n</code> | number of bins |
|----------------|----------------|

**Value**

returns an object of class `smoothing_freq`

**References**

Han, J., Kamber, M., Pei, J. (2011). Data Mining: Concepts and Techniques. (Discretization)

**Examples**

```

data(iris)
obj <- smoothing_freq(n = 2)
obj <- fit(obj, iris$Sepal.Length)
sl.bi <- transform(obj, iris$Sepal.Length)
table(sl.bi)
obj$interval

bins <- cut(iris$Sepal.Length, unique(obj$interval.adj), FALSE, include.lowest = TRUE)
entro <- evaluate(obj, bins, iris$Species)
entro$entropy

```



---

|                 |                                    |
|-----------------|------------------------------------|
| smoothing_inter | <i>Smoothing by equal interval</i> |
|-----------------|------------------------------------|

---

**Description**

Discretize a numeric vector into  $n$  equal-width intervals (robust bounds via boxplot whiskers) and replace each value by the bin mean.

**Usage**

```
smoothing_inter(n)
```

**Arguments**

|     |                |
|-----|----------------|
| $n$ | number of bins |
|-----|----------------|

**Value**

returns an object of class `smoothing_inter`

**References**

Han, J., Kamber, M., Pei, J. (2011). Data Mining: Concepts and Techniques. (Discretization)

**Examples**

```
data(iris)
obj <- smoothing_inter(n = 2)
obj <- fit(obj, iris$Sepal.Length)
sl.bi <- transform(obj, iris$Sepal.Length)
table(sl.bi)
obj$interval

bins <- cut(iris$Sepal.Length, unique(obj$interval.adj), FALSE, include.lowest = TRUE)
entro <- evaluate(obj, bins, iris$Species)
entro$entropy
```

---

|                        |                                            |
|------------------------|--------------------------------------------|
| smoothing_quantization | <i>Smoothing by quantization (k-means)</i> |
|------------------------|--------------------------------------------|

---

**Description**

Quantize a numeric vector into  $n$  levels using  $k$ -means on the values and replace each value by its cluster mean (vector quantization).

**Usage**

smoothing\_quantization(n)

**Arguments**

n                      number of bins

**Value**

returns an object of class smoothing\_quantization

**References**

MacQueen, J. (1967). Some Methods for classification and Analysis of Multivariate Observations.

**Examples**

```
data(iris)
obj <- smoothing_quantization(n = 2)
obj <- fit(obj, iris$Sepal.Length)
sl.bi <- transform(obj, iris$Sepal.Length)
table(sl.bi)
obj$interval

bins <- cut(iris$Sepal.Length, unique(obj$interval.adj), FALSE, include.lowest = TRUE)
entro <- evaluate(obj, bins, iris$Species)
entro$entropy
```

---

|            |                             |
|------------|-----------------------------|
| train_test | <i>Train-Test Partition</i> |
|------------|-----------------------------|

---

**Description**

Partition a dataset into training and test sets using a sampling strategy.

**Usage**

train\_test(obj, data, perc = 0.8, ...)

**Arguments**

obj                      an object of a class that supports the train\_test method

data                     dataset to be partitioned

perc                     a numeric value between 0 and 1 specifying the proportion of data to be used for training

...                      additional optional arguments passed to specific methods.

**Value**

returns an list with two elements:

- train: A data frame containing the training set
- test: A data frame containing the test set

**Examples**

```
#using random sampling
sample <- sample_random()
tt <- train_test(sample, iris)

# distribution of train
table(tt$train$Species)
```

---

train\_test\_from\_folds *k-Fold Training and Test Partition Object*

---

**Description**

Build one train/test split from a list of folds produced by cross-validation.

**Usage**

```
train_test_from_folds(folds, k)
```

**Arguments**

|       |                                        |
|-------|----------------------------------------|
| folds | Data already partitioned into folds.   |
| k     | Fold index to be used as the test set. |

**Details**

The k-th fold is returned as the test set and all remaining folds are concatenated into the training set. This helper makes manual cross-validation loops easier to explain and reproduce.

**Value**

A list with two elements:

- train: data frame containing all folds except the k-th.
- test: data frame corresponding to the k-th fold.

**Examples**

```
# Create k-fold partitions of a dataset (e.g., iris)
folds <- k_fold(sample_random(), iris, k = 5)

# Use the first fold as the test set and combine the remaining folds
# for the training set
train_test_split <- train_test_from_folds(folds, k = 1)

# Display the training set
head(train_test_split$train)

# Display the test set
head(train_test_split$test)
```

---

|           |                  |
|-----------|------------------|
| transform | <i>Transform</i> |
|-----------|------------------|

---

**Description**

Generic to apply a transformation to data.

**Usage**

```
transform(obj, ...)
```

**Arguments**

|     |                         |
|-----|-------------------------|
| obj | a dal_transform object. |
| ... | optional arguments.     |

**Value**

returns a transformed data.

**Examples**

```
#See ?minmax for an example of transformation
```

---

|        |                              |
|--------|------------------------------|
| zscore | <i>Z-score normalization</i> |
|--------|------------------------------|

---

**Description**

Standardize numeric columns to zero mean and unit variance, optionally rescaled to a target mean (nmean) and sd (nsd).

**Usage**

```
zscore(nmean = 0, nsd = 1)
```

**Arguments**

|       |                                            |
|-------|--------------------------------------------|
| nmean | new mean for normalized data               |
| nsd   | new standard deviation for normalized data |

**Details**

For each numeric column  $j$ , computes  $((x - \text{mean}_j)/\text{sd}_j) * \text{nsd} + \text{nmean}$ . Constant columns become nmean.

$$zscore = (x - \text{mean}(x))/\text{sd}(x)$$

**Value**

returns the z-score transformation object

**References**

Han, J., Kamber, M., Pei, J. (2011). Data Mining: Concepts and Techniques. (Standardization)

**Examples**

```
data(iris)
head(iris)

trans <- zscore()
trans <- fit(trans, iris)
tiris <- transform(trans, iris)
head(tiris)

itiris <- inverse_transform(trans, tiris)
head(itiris)
```

# Index

- \* **datasets**
  - Boston, [12](#)
- \* **graphics**
  - `dal_graphics`, [40](#)
- \* **visualization**
  - `dal_graphics`, [40](#)
- `action`, [5](#)
- `action.dal_transform`, [6](#)
- `adjust_class_label`, [6](#)
- `adjust_data.frame`, [7](#)
- `adjust_factor`, [7](#)
- `adjust_matrix`, [8](#)
- `aggregation`, [8](#)
- `autoenc_base_e`, [9](#)
- `autoenc_base_ed`, [10](#)
- 
- `bal_oversampling`, [11](#)
- `bal_subsampling`, [11](#)
- Boston, [12](#)
- 
- `categ_mapping`, [13](#)
- `cla_bagging`, [14](#)
- `cla_boosting`, [15](#)
- `cla_dtree`, [16](#)
- `cla_glm`, [17](#)
- `cla_glmnet`, [17](#)
- `cla_knn`, [18](#)
- `cla_majority`, [19](#)
- `cla_mlp`, [20](#)
- `cla_multinom`, [21](#)
- `cla_nb`, [22](#)
- `cla_rf`, [23](#)
- `cla_rpart`, [24](#)
- `cla_svm`, [25](#)
- `cla_tune`, [26](#)
- `cla_xgboost`, [27](#)
- `classification`, [14](#)
- `clu_tune`, [39](#)
- `cluster`, [28](#)
- 
- `cluster_cmeans`, [29](#)
- `cluster_dbscan`, [30](#)
- `cluster_gmm`, [32](#)
- `cluster_hclust`, [33](#)
- `cluster_kmeans`, [34](#)
- `cluster_louvain_graph`, [35](#)
- `cluster_pam`, [36](#)
- `clusterer`, [28](#)
- `cluutils`, [38](#)
- 
- `dal_base`, [40](#)
- `dal_graphics`, [40](#)
- `dal_learner`, [41](#)
- `dal_transform`, [41](#)
- `dal_tune`, [42](#)
- `data_sample`, [43](#)
- `discover`, [43](#)
- `dt_pca`, [44](#)
- 
- `evaluate`, [45](#)
- 
- `feature_generation`, [46](#)
- `feature_selection_corr`, [46](#)
- `feature_selection_fss`, [47](#)
- `feature_selection_info_gain`, [48](#)
- `feature_selection_lasso`, [49](#)
- `feature_selection_relief`, [50](#)
- `feature_selection_stepwise`, [51](#)
- `fit`, [52](#)
- `fit.cla_tune`, [52](#)
- `fit.cluster_dbscan`, [53](#)
- `fit_curvature_max`, [53](#)
- `fit_curvature_min`, [54](#)
- 
- `hierarchy_cut`, [55](#)
- 
- `imputation_predictive`, [55](#)
- `imputation_simple`, [56](#)
- `imputation_tree`, [57](#)
- `inverse_transform`, [58](#)

k\_fold, 59

minmax, 59

na\_removal, 60

outliers\_boxplot, 61

outliers\_gaussian, 62

pat\_apriori, 64

pat\_confidence\_threshold, 66

pat\_cspade, 67

pat\_dara, 68

pat\_dara\_rules, 69

pat\_eclat, 70

pat\_filter\_rules, 71

pat\_rule\_filter\_dara, 73

pat\_rule\_filter\_interest, 74

pat\_rule\_filter\_none, 75

pat\_rules\_tidy, 72

pat\_support\_threshold, 75

pattern\_miner, 63

patutils, 63

plot\_bar, 76

plot\_boxplot, 77

plot\_boxplot\_class, 78

plot\_correlation, 79

plot\_dendrogram, 80

plot\_density, 81

plot\_density\_class, 82

plot\_groupedbar, 83

plot\_hist, 84

plot\_lollipop, 84

plot\_pair, 86

plot\_pair\_adv, 86

plot\_parallel, 87

plot\_pieplot, 88

plot\_pixel, 89

plot\_points, 90

plot\_radar, 91

plot\_scatter, 92

plot\_series, 92

plot\_stackedbar, 93

plot\_ts, 94

plot\_ts\_pred, 95

predictor, 96

predictor\_prepare\_fit, 96

reg\_dtrees, 97

reg\_knn, 98

reg\_lm, 99

reg\_mlp, 100

reg\_rf, 101

reg\_svm, 102

reg\_tune, 103

regression, 97

sample\_balance, 104

sample\_groups, 105

sample\_random, 106

sample\_simple, 107

sample\_stratified, 107

select\_hyper, 108

select\_hyper.cla\_tune, 109

set\_params, 109

set\_params.default, 110

smoothing, 110

smoothing\_cluster, 111

smoothing\_freq, 112

smoothing\_inter, 113

smoothing\_quantization, 113

train\_test, 114

train\_test\_from\_folds, 115

transform, 116

zscore, 117