

# Package ‘daltoolbox’

October 25, 2023

**Title** Leveraging Experiment Lines to Data Analytics

**Version** 1.0.727

## 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://github.com/cefet-rj-dal/daltoolbox>

**Encoding** UTF-8

**RoxygenNote** 7.2.3

**Imports** FNN, MLmetrics, caret, class, cluster, dbscan, dplyr, e1071, elmNNRcpp, forecast, ggplot2, nnet, randomForest, reshape, tree, reticulate

**Config/reticulate** list( packages = list( list(package = ``scipy"`, list(package = ``torch"`, list(package = ``pandas"`, list(package = ``numpy"`, list(package = ``matplotlib"`, list(package = ``scikit-learn"`, list(package = ``functools"`, list(package = ``operator"`, list(package = ``sys"`) ) )

**NeedsCompilation** no

**Author** Eduardo Ogasawara [aut, ths, cre]  
(<<https://orcid.org/0000-0002-0466-0626>>),  
Antonio Castro [aut, ctb],  
Heraldo Borges [aut, ths],  
Diego Carvalho [aut, ths],  
Joel Santos [aut, ths],

Eduardo Bezerra [aut, ths],  
 Rafaelli Coutinho [aut, ths],  
 Federal Center for Technological Education of Rio de Janeiro (CEFET/RJ)  
 [cph]

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

**Repository** CRAN

**Date/Publication** 2023-10-25 21:40:02 UTC

## R topics documented:

action	4
action.dal_transform	5
adjust_class_label	5
adjust_data.frame	6
adjust_factor	6
adjust_matrix	7
adjust_ts_data	7
autoenc_encode	8
autoenc_encode_decode	9
Boston	9
categ_mapping	11
classification	11
cla_dtree	12
cla_knn	13
cla_majority	14
cla_mlp	15
cla_nb	16
cla_rf	16
cla_svm	17
cla_tune	18
cluster	19
clusterer	20
cluster_dbscan	20
cluster_kmeans	21
cluster_pam	22
clu_tune	22
dal_base	23
dal_learner	24
dal_transform	24
dal_tune	25
data_sample	25
do_fit	26
do_predict	27
dt_pca	27
evaluate	28
fit	29
fit.cla_tune	29

fit.cluster_dbscan . . . . .	30
fit_curvature_max . . . . .	30
fit_curvature_min . . . . .	31
inverse_transform . . . . .	32
k_fold . . . . .	32
minmax . . . . .	33
MSE.ts . . . . .	34
outliers . . . . .	34
plot_bar . . . . .	35
plot_boxplot . . . . .	36
plot_boxplot_class . . . . .	36
plot_density . . . . .	37
plot_density_class . . . . .	38
plot_groupedbar . . . . .	39
plot_hist . . . . .	39
plot_lollipop . . . . .	40
plot_pieplot . . . . .	41
plot_points . . . . .	42
plot_radar . . . . .	43
plot_scatter . . . . .	43
plot_series . . . . .	44
plot_stackedbar . . . . .	45
plot_ts . . . . .	45
plot_ts_pred . . . . .	46
predictor . . . . .	47
regression . . . . .	48
reg_dtree . . . . .	49
reg_knn . . . . .	49
reg_mlp . . . . .	50
reg_rf . . . . .	51
reg_svm . . . . .	52
reg_tune . . . . .	53
sample_random . . . . .	54
sample_stratified . . . . .	54
select_hyper . . . . .	55
select_hyper.cla_tune . . . . .	56
select_hyper.ts_tune . . . . .	56
set_params . . . . .	57
set_params.default . . . . .	57
sin_data . . . . .	58
sMAPE.ts . . . . .	58
smoothing . . . . .	59
smoothing_cluster . . . . .	59
smoothing_freq . . . . .	60
smoothing_inter . . . . .	61
train_test . . . . .	61
train_test_from_folds . . . . .	62
transform . . . . .	62

ts_arma	63
ts_conv1d	64
ts_data	64
ts_elm	65
ts_head	66
ts_knn	67
ts_lstm	68
ts_mlp	68
ts_norm_an	69
ts_norm_diff	70
ts_norm_ean	71
ts_norm_gminmax	72
ts_norm_swminmax	72
ts_projection	73
ts_reg	74
ts_regrsw	74
ts_rf	75
ts_sample	76
ts_svm	77
ts_tune	78
zscore	79
[.ts_data	79

## Index 81

---

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

---

### Description

Executes the action of model applied in provided data

### Usage

```
action(obj, ...)
```

### Arguments

obj	object: a <code>dal_base</code> object to apply the transformation on the input dataset.
...	optional arguments.

### Value

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**

A default function that defines the action to proxy transform method

**Usage**

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

**Arguments**

obj	object
...	optional arguments

**Value**

Transformed data

**Examples**

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

---

adjust\_class\_label    *adjust categorical mapping*

---

**Description**

vector value is adjusted to a categorical mapping

**Usage**

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

**Arguments**

x	vector to be categorized
valTrue	value to represent true
valFalse	value to represent false

**Value**

an adjusted categorical mapping

---

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

---

**Description**

dataset data is adjusted to a data.frame

**Usage**

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

**Arguments**

data	dataset
------	---------

**Value**

The date argument

**Examples**

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

---

adjust_factor	<i>adjust factors</i>
---------------	-----------------------

---

**Description**

vector value is adjusted to a factor

**Usage**

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

**Arguments**

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

**Value**

an adjusted factor

---

*adjust\_matrix*            *adjust to matrix*

---

**Description**

dataset data is adjusted to a matrix

**Usage**

```
adjust_matrix(data)
```

**Arguments**

data            dataset

**Value**

an adjusted matrix

**Examples**

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

---

*adjust\_ts\_data*            *adjust ts\_data*

---

**Description**

dataset data is adjusted to a ts\_data

**Usage**

```
adjust_ts_data(data)
```

**Arguments**

data            dataset

**Value**

an adjusted ts\_data

---

autoenc\_encode            *Autoencoder - Encode*

---

**Description**

Creates an deep learning autoencoder to encode a sequence of observations. It wraps the pytorch library.

**Usage**

```
autoenc_encode(  
    input_size,  
    encoding_size,  
    batch_size = 32,  
    num_epochs = 1000,  
    learning_rate = 0.001  
)
```

**Arguments**

input\_size        input size  
encoding\_size    encoding size  
batch\_size       size for batch learning  
num\_epochs       number of epochs for training  
learning\_rate    learning rate

**Value**

a autoenc\_encode object.

**Examples**

#See example at <https://nbviewer.org/github/cefet-rj-dal/daltoolbox-examples>



---

autoenc\_encode\_decode *Autoencoder - Encode*

---

### Description

Creates an deep learning autoencoder to encode a sequence of observations. It wraps the pytorch library.

### Usage

```
autoenc_encode_decode(  
    input_size,  
    encoding_size,  
    batch_size = 32,  
    num_epochs = 1000,  
    learning_rate = 0.001  
)
```

### Arguments

input_size	input size
encoding_size	encoding size
batch_size	size for batch learning
num_epochs	number of epochs for training
learning_rate	learning rate

### Value

a autoenc\_encode\_decode object.

### Examples

#See example at <https://nbviewer.org/github/cefet-rj-dal/daltoolbox-examples>

---

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:  $1000(Bk - 0.63)^2$  where Bk 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</i>
---------------	----------------------------

---

**Description**

Categorical mapping provides a way to map the levels of a categorical variable to new values. Each possible value is converted to a binary attribute.

**Usage**

```
categ_mapping(attribute)
```

**Arguments**

attribute      attribute to be categorized.

**Value**

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</i>
----------------	-----------------------

---

**Description**

Ancestor class for classification problems

**Usage**

```
classification(attribute, slevels)
```

**Arguments**

attribute      attribute target to model building  
slevels          • possible values for the target classification

**Value**

classification object

**Examples**

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

---

cla_dtree	<i>Decision Tree for classification</i>
-----------	---

---

**Description**

Creates a classification object that uses the Decision Tree algorithm for classification. It wraps the tree library.

**Usage**

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

**Arguments**

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

**Value**

A classification object that uses the Decision Tree algorithm for classification.

**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_knn	<i>K Nearest Neighbor Classification</i>
---------	--

---

**Description**

Classifies using the K-Nearest Neighbor algorithm. It wraps the class library.

**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.

**Value**

A knn object.

**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 Classification*

---

**Description**

This function creates a classification object that uses the majority vote strategy to predict the target attribute. Given a target attribute, the function counts the number of occurrences of each value in the dataset and selects the one that appears most often.

**Usage**

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

**Arguments**

<code>attribute</code>	attribute target to model building.
<code>slevels</code>	Possible values for the target classification.

**Value**

Returns a classification object.

**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	<i>MLP for classification</i>
---------	-------------------------------

---

**Description**

Creates a classification object that uses the Multi-Layer Perceptron (MLP) method. It wraps the nnet library.

**Usage**

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

**Arguments**

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

**Value**

a classification object

**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_nb`*Naive Bayes Classifier*

---

**Description**

Classification using the Naive Bayes algorithm It wraps the e1071 library.

**Usage**

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

**Arguments**

`attribute`      attribute target to model building.  
`slevels`        Possible values for the target classification.

**Value**

A classification object.

**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**

Creates a classification object that uses the Random Forest method It wraps the randomForest library.



**Usage**

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

**Arguments**

attribute	attribute target to model building
slevels	possible values for the target classification
nodesize	node size
ntree	number of trees
mtry	number of attributes to build tree

**Value**

obj

**Examples**

```
data(iris)
slevels <- levels(iris$Species)
model <- cla_rf("Species", slevels, 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\_svm

*SVM for classification*

---

**Description**

Creates a classification object that uses the Support Vector Machine (SVM) method for classification. It wraps the e1071 library.

**Usage**

```
cla_svm(attribute, slevels, epsilon = 0.1, cost = 10, kernel = "radial")
```

**Arguments**

attribute	attribute target to model building
slevels	possible values for the target classification
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)

**Value**

A SVM classification object

**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 Tune*

---

**Description**

Classification Tune

**Usage**

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

**Arguments**

base_model	base model for tuning
folds	number of folds for cross-validation
metric	metric used to optimize

**Value**

a `cla_tune` object.

**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, ranges)

# 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
```

---

cluster

*Cluster*

---

**Description**

Defines a cluster method.

**Usage**

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

**Arguments**

`obj` a clusterer object.  
`...` optional arguments.

**Value**

clustered data.

**Examples**

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

---

`clusterer`*Clusterer*

---

**Description**

Ancestor class for clustering problems

**Usage**

```
clusterer()
```

**Value**

a clusterer object

**Examples**

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

---

`cluster_dbscan`*DBSCAN*

---

**Description**

Creates a clusterer object that uses the DBSCAN method It wraps the dbscan library.

**Usage**

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

**Arguments**

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

**Value**

A dbscan object.

**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_kmeans	<i>k-means</i>
----------------	----------------

---

**Description**

Creates a clusterer object that uses the k-means method It wraps the stats library.

**Usage**

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

**Arguments**

k                    The number of clusters to form.

**Value**

A k-means object.

**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\_pam                    *PAM*

---

**Description**

Creates a clusterer object that uses the Partition Around Medoids (PAM) method It wraps the cluster library.

**Usage**

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

**Arguments**

k                                The number of clusters to generate.

**Value**

A PAM object.

**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
```

---

clu\_tune                        *Clustering Tune*

---

**Description**

Clustering Tune

**Usage**

```
clu_tune(base_model)
```

**Arguments**

base\_model      base model for tuning

**Value**

a clu\_tune object.

**Examples**

```
data(iris)

# fit model
model <- clu_tune(cluster_kmeans(k = 0))
ranges <- list(k = 1:10)
model <- fit(model, iris[,1:4], ranges)
model$k
```

---

dal\_base

*Class dal\_base*

---

**Description**

The dal\_base class is an abstract class for all dal descendants classes. It provides both fit() and action() functions

**Usage**

```
dal_base()
```

**Value**

A dal\_base object

**Examples**

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

---

`dal_learner`*DAL Learner*

---

**Description**

A ancestor class for clustering, classification, regression, and time series regression. It also provides the basis for specialized evaluation of learning performance.

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

**Usage**

```
dal_learner()
```

**Value**

a learner

**Examples**

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

---

`dal_transform`*DAL Transform*

---

**Description**

A transformation method applied to a dataset. If needed, the fit can be called to adjust the transform.

**Usage**

```
dal_transform()
```

**Value**

a `dal_transform` object.

**Examples**

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



---

`dal_tune`*DAL Tune*

---

**Description**

Ancestor class for hyper parameter optimization

**Usage**

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

**Arguments**

<code>base_model</code>	base model for tuning
<code>folds</code>	number of folds for cross-validation

**Value**

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`*Data Sample*

---

**Description**

The `data_sample` function in R is used to randomly sample data from a given data frame. It can be used to obtain a subset of data for further analysis or modeling.

Two basic specializations of `data_sample` are `sample_random` and `sample_stratified`. They provide random sampling and stratified sampling, respectively.

Data sample provides both training and testing partitioning (`train_test`) and k-fold partitioning (`k_fold`) of data.

**Usage**

```
data_sample()
```

**Value**

`obj`

**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)
```

---

do\_fit

*do fit for time series*

---

**Description**

The actual time series model fitting. This method should be override by descendants.

**Usage**

```
do_fit(obj, x, y = NULL)
```

**Arguments**

obj	object
x	input variable
y	output variable

**Value**

fitted object

---

do_predict	<i>do predict for time series</i>
------------	-----------------------------------

---

**Description**

The actual time series model prediction. This method should be override by descendants.

**Usage**

```
do_predict(obj, x)
```

**Arguments**

obj	object
x	input variable

**Value**

predicted values

---

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

---

**Description**

PCA (Principal Component Analysis) is an unsupervised dimensionality reduction technique used in data analysis and machine learning. It transforms a dataset of possibly correlated variables into a new set of uncorrelated variables called principal components.

**Usage**

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

**Arguments**

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

**Value**

obj

## 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

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
```

---

fit	<i>Fit</i>
-----	------------

---

**Description**

Fits a model.

**Usage**

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

**Arguments**

obj	object
...	optional arguments.

**Value**

obj

**Examples**

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

---

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

---

**Description**

tune hyperparameters of ml model for classification

**Usage**

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

**Arguments**

obj	object
data	dataset
ranges	hyperparameters ranges
...	optional arguments

**Value**

fitted obj

---

fit.cluster\_dbscan     *fit dbscan model*


---

**Description**

fit dbscan model

**Usage**

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

**Arguments**

obj	object
data	dataset
...	optional arguments

**Value**

fitted obj

---

fit\_curvature\_max     *maximum curvature analysis*


---

**Description**

Fitting a curvature model in a sequence of observations. It extracts the the maximum curvature computed.

**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      *minimum curvature analysis*

---

**Description**

Fitting a curvature model in a sequence of observations. It extracts the the minimum curvature computed.

**Usage**

```
fit_curvature_min()
```

**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 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)
```

---

inverse\_transform      *Inverse Transform*

---

**Description**

Reverses the transformation applied to data.

**Usage**

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

**Arguments**

obj                    a dal\_transform object.  
...                    optional arguments.

**Value**

dataset inverse transformed.

**Examples**

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

---

k\_fold                    *k-fold sampling*

---

**Description**

k-fold partition of a dataset using a sampling method

**Usage**

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

**Arguments**

obj                    object  
data                   dataset  
k                      number of folds

**Value**

k folds



**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

*min-max normalization*

---

**Description**

The minmax performs scales data between [0,1].  $\text{minmax} = (x - \min(x)) / (\max(x) - \min(x))$ .

**Usage**

```
minmax()
```

**Value**

```
obj
```

**Examples**

```
data(iris)
head(iris)

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

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

---

MSE.ts

*MSE*


---

**Description**

Compute the mean squared error (MSE) between actual values and forecasts of a time series

**Usage**

```
MSE.ts(actual, prediction)
```

**Arguments**

actual	real observations
prediction	predicted observations

**Value**

A number, which is the calculated MSE

---

outliers

*Outliers*


---

**Description**

The outliers class uses box-plot definition for outliers. An outlier is a value that is below than  $Q_1 - 1.5 \cdot IQR$  or higher than  $Q_3 + 1.5 \cdot IQR$ . The class remove outliers for numeric attributes. Users can set alpha to 3 to remove extreme values.

**Usage**

```
outliers(alpha = 1.5)
```

**Arguments**

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

**Value**

An outlier object

**Examples**

```
# code for outlier removal
out_obj <- outliers() # 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 <- iris[idx,]
iris.outliers
```

---

plot\_bar

*plot bar graph*

---

**Description**

plot bar graph

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector
alpha	level of transparency

**Value**

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_bar(data, colors="blue")
plot(grf)
```

---

plot_boxplot	<i>plot boxplot</i>
--------------	---------------------

---

**Description**

plot boxplot

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector
barwidth	width of bar

**Value**

ggplot graphic

**Examples**

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

---

plot_boxplot_class	<i>plot boxplot per class</i>
--------------------	-------------------------------

---

**Description**

plot boxplot per class

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
class_label	name of attribute for class label
label_x	x-axis label
label_y	y-axis label
colors	color vector

**Value**

ggplot graphic

**Examples**

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

---

plot_density	<i>plot density</i>
--------------	---------------------

---

**Description**

plot density

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector
bin	bin width
alpha	level of transparency

**Value**

ggplot graphic

**Examples**

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

---

plot\_density\_class     *plot density per class*

---

**Description**

plot density per class

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
class_label	name of attribute for class label
label_x	x-axis label
label_y	y-axis label
colors	color vector
bin	bin width
alpha	level of transparency

**Value**

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**

plot grouped bar

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector
alpha	level of transparency

**Value**

ggplot graphic

**Examples**

```
data <- iris |> dplyr::group_by(Species) |>
  dplyr::summarize(Sepal.Length=mean(Sepal.Length), Sepal.Width=mean(Sepal.Width))
grf <- plot_groupedbar(data, colors=c("blue", "red"))
plot(grf)
```

---

plot_hist	<i>plot histogram</i>
-----------	-----------------------

---

**Description**

plot histogram

**Usage**

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

**Arguments**

<code>data</code>	data.frame contain x, value, and variable
<code>label_x</code>	x-axis label
<code>label_y</code>	y-axis label
<code>color</code>	color vector
<code>alpha</code>	transparency level

**Value**

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**

plot lollipop

**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**

<code>data</code>	data.frame contain x, value, and variable
<code>label_x</code>	x-axis label
<code>label_y</code>	y-axis label
<code>colors</code>	color vector
<code>color_text</code>	color of text inside ball



size_text	size of text inside ball
size_ball	size of ball
alpha_ball	transparency of ball
min_value	minimum value
max_value_gap	maximum value gap

**Value**

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_lollipop(data, colors="blue", max_value_gap=0.2)
plot(grf)
```

---

plot_pieplot	<i>plot pie</i>
--------------	-----------------

---

**Description**

plot pie

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector
textcolor	text color
bordercolor	border color

**Value**

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_pieplot(data, colors=c("red", "green", "blue"))
plot(grf)
```

---

plot\_points

*plot points*

---

**Description**

plot points

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector

**Value**

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	<i>plot radar</i>
------------	-------------------

---

**Description**

plot radar

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector

**Value**

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") + ggplot2::ylim(0, NA)
plot(grf)
```

---

plot_scatter	<i>scatter graph</i>
--------------	----------------------

---

**Description**

scatter graph

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector

**Value**

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**

plot series

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector

**Value**

plot

**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**

plot stacked bar

**Usage**

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

**Arguments**

data	data.frame contain x, value, and variable
label_x	x-axis label
label_y	y-axis label
colors	color vector
alpha	level of transparency

**Value**

ggplot graphic

**Examples**

```
data <- iris |> dplyr::group_by(Species) |>
  dplyr::summarize(Sepal.Length=mean(Sepal.Length), Sepal.Width=mean(Sepal.Width))
grf <- plot_stackedbar(data, colors=c("blue", "red"))
plot(grf)
```

---

plot_ts	<i>Plot a time series chart</i>
---------	---------------------------------

---

**Description**

The function receives six variables as a parameter, which are obj and y, yadj, main and xlabel. The graph is plotted with 3 lines: the original series (in black), the adjusted series (in blue) and the predicted series (in green)

**Usage**

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

**Arguments**

x	input variable
y	output variable
label_x	x-axis label
label_y	y-axis label
color	color for time series

**Value**

ggplot graphic

**Examples**

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

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

---

plot\_ts\_pred

*Plot a time series chart*

---

**Description**

The function receives six variables as a parameter, which are obj and y, yadj, main and xlabel. The graph is plotted with 3 lines: the original series (in black), the adjusted series (in blue) and the predicted series (in green)

**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
y	time series
yadj	adjustment of time series
ypred	prediction of the time series
label_x	x-axis title
label_y	y-axis title
color	color for the time series
color_adjust	color for the adjusted values
color_prediction	color for the predictions

**Value**

ggplot graphic

**Examples**

```
data(sin_data)
ts <- ts_data(sin_data$y, 0)
ts_head(ts, 3)

samp <- ts_sample(ts, test_size= 5)
io_train <- ts_projection(samp$train)
io_test <- ts_projection(samp$test)

model <- ts_arima()
model <- fit(model, x=io_train$input, y=io_train$output)
adjust <- predict(model, io_train$input)

prediction <- predict(model, x=io_test$input, steps_ahead=5)
prediction <- as.vector(prediction)

yvalues <- c(io_train$output, io_test$output)
grf <- plot_ts_pred(y=yvalues, yadj=adjust, ypre=prediction)
plot(grf)
```

---

predictor

*DAL Predict*

---

**Description**

Ancestor class for regression and classification. It provides basis for fit and predict methods. Besides, action method proxies to predict.

An example of learner is a decision tree (cla\_dtree)

**Usage**

```
predictor()
```

**Value**

a predictor object

**Examples**

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

---

regression

*Regression*

---

**Description**

Ancestor class for regression problems

**Usage**

```
regression(attribute)
```

**Arguments**

attribute      attribute target to model building

**Value**

regression object

**Examples**

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



---

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

---

**Description**

Creates a regression object that uses the Decision Tree method for regression. It wraps the tree library.

**Usage**

```
reg_dtree(attribute)
```

**Arguments**

attribute      attribute target to model building.

**Value**

A decision tree regression object

**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	<i>knn regression</i>
---------	-----------------------

---

**Description**

Creates a regression object that uses the K-Nearest Neighbors (knn) method for regression

**Usage**

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

**Arguments**

attribute	attribute target to model building
k	number of k neighbors

**Value**

A knn regression object

**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\_mlp

*MLP for regression*


---

**Description**

Creates a regression object that uses the Multi-Layer Perceptron (MLP) method. It wraps the nnet library.

**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

**Value**

obj

**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**

Creates a regression object that uses the Random Forest method. It wraps the randomForest library.

**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

**Value**

obj

**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

Creates a regression object that uses the Support Vector Machine (SVM) method for regression. It wraps the e1071 library.

### Usage

```
reg_svm(attribute, epsilon = 0.1, cost = 10, kernel = "radial")
```

### 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)

### Value

A SVM regression object

### 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

*Regression Tune*

---

## Description

Regression Tune

## Usage

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

## Arguments

base_model	base model for tuning
folds	number of folds for cross-validation

## Value

a reg\_tune object.

## 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"))
ranges <- list(size=c(3), decay=c(0.1,0.5))

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

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

---

`sample_random`*Sample Random*

---

**Description**

The `sample_random` function in R is used to generate a random sample of specified size from a given data set.

**Usage**

```
sample_random()
```

**Value**

obj

**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_stratified`*sample\_stratified*

---

**Description**

The `sample_stratified` function in R is used to generate a stratified random sample from a given dataset. Stratified sampling is a statistical method that is used when the population is divided into non-overlapping subgroups or strata, and a sample is selected from each stratum to represent the entire population. In stratified sampling, the sample is selected in such a way that it is representative of the entire population and the variability within each stratum is minimized.

**Usage**

```
sample_stratified(attribute)
```

**Arguments**

attribute      attribute target to model building

**Value**

obj

**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 hyper parameters*

---

**Description**

Selection hyper parameters from a k-fold cross-validation execution

**Usage**

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

**Arguments**

obj                      object  
hyperparameters              data set with hyper parameters and quality measure from execution

**Value**

index of selected hyper parameter

---

select\_hyper.cla\_tune *selection of hyperparameters*

---

**Description**

selection of hyperparameters (maximizing classification metric)

**Usage**

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

**Arguments**

obj                    object  
hyperparameters        hyperparameters dataset

**Value**

optimized key number of hyperparameters

---

select\_hyper.ts\_tune *selection of hyperparameters (time series)*

---

**Description**

selection of hyperparameters (minimizing error)

**Usage**

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

**Arguments**

obj                    object  
hyperparameters        hyperparameters dataset

**Value**

optimized key number of hyperparameters



---

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

---

**Description**

set\_params function assigns all parameters to the attributes presented in the object. It returns the object with the parameters set.

**Usage**

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

**Arguments**

obj	object of class dal_base
params	parameters to set obj

**Value**

obj with parameters set

**Examples**

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

---

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

---

**Description**

This function receives the obj and params variables as parameters. It returns the obj as it is.

**Usage**

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

**Arguments**

obj	object
params	parameters

**Value**

obj

sin\_data                      *Time series example dataset*

---

**Description**

Synthetic dataset of sine function.

- x: correspond time from 0 to 10.
- y: dependent variable for time series modeling.

**Usage**

```
data(sin_data)
```

**Format**

```
data.frame.
```

**Source**

This dataset was generated for examples.

**Examples**

```
data(sin_data)
head(sin_data)
```

---

sMAPE.ts                      *sMAPE*

---

**Description**

Compute the symmetric mean absolute percent error (sMAPE)

**Usage**

```
sMAPE.ts(actual, prediction)
```

**Arguments**

actual	real observations
prediction	predicted observations

**Value**

The sMAPE between the actual and prediction vectors

---

`smoothing`*Smoothing*

---

**Description**

Smoothing is a statistical technique used to reduce the noise in a signal or a dataset by removing the high-frequency components. The smoothing level is associated with the number of bins used. There are alternative methods to establish the smoothing: equal interval, equal frequency, and clustering.

**Usage**

```
smoothing(n)
```

**Arguments**

`n`                    number of bins

**Value**

`obj`

**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

entro <- evaluate(obj, as.factor(names(sl.bi)), iris$Species)
entro$entropy
```

---

`smoothing_cluster`*Smoothing by cluster*

---

**Description**

Uses clustering method to perform data smoothing. The input vector is divided into clusters using the k-means algorithm. The mean of each cluster is then calculated and used as the smoothed value for all observations within that cluster.

**Usage**

```
smoothing_cluster(n)
```

**Arguments**

n                    number of bins

**Value**

obj

**Examples**

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

entro <- evaluate(obj, as.factor(names(sl.bi)), iris$Species)
entro$entropy
```

---

smoothing\_freq                    *Smoothing by Freq*

---

**Description**

The `'smoothing_freq'` function is used to smooth a given time series data by aggregating observations within a fixed frequency.

**Usage**

```
smoothing_freq(n)
```

**Arguments**

n                    number of bins

**Value**

obj

**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

entro <- evaluate(obj, as.factor(names(sl.bi)), iris$Species)
entro$entropy
```

---

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

---

**Description**

The "smoothing by interval" function is used to apply a smoothing technique to a vector or time series data using a moving window approach.

**Usage**

```
smoothing_inter(n)
```

**Arguments**

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

**Value**

obj

**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

entro <- evaluate(obj, as.factor(names(sl.bi)), iris$Species)
entro$entropy
```

---

train_test	<i>training and test</i>
------------	--------------------------

---

**Description**

training and test partition of a dataset using a sampling method

**Usage**

```
train_test(obj, data, ...)
```

**Arguments**

obj	object
data	dataset
...	optional arguments.

**Value**

train and test sets

**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**

k-fold training and test partition object

**Usage**

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

**Arguments**

<code>folds</code>	data partitioned into folds
<code>k</code>	k-fold for test set, all reminder for training set

**Value**

train and test folds

---

<code>transform</code>	<i>Transform</i>
------------------------	------------------

---

**Description**

Defines a transformation method.

**Usage**

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

**Arguments**

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

**Value**

transformed data.

**Examples**

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

---

ts_arima	ARIMA
----------	-------

---

**Description**

Creates a time series prediction object that uses the AutoRegressive Integrated Moving Average (ARIMA). It wraps the forecast library.

**Usage**

```
ts_arima()
```

**Value**

a ts\_arima object.

**Examples**

```
data(sin_data)
ts <- ts_data(sin_data$y, 0)
ts_head(ts, 3)

samp <- ts_sample(ts, test_size = 5)
io_train <- ts_projection(samp$train)
io_test <- ts_projection(samp$test)

model <- ts_arima()
model <- fit(model, x=io_train$input, y=io_train$output)

prediction <- predict(model, x=io_test$input[1,], steps_ahead=5)
prediction <- as.vector(prediction)
output <- as.vector(io_test$output)

ev_test <- evaluate(model, output, prediction)
ev_test
```

---

 ts\_conv1d

*Conv1D*


---

### Description

Creates a time series prediction object that uses the Conv1D. It wraps the pytorch library.

### Usage

```
ts_conv1d(preprocess = NA, input_size = NA, epochs = 10000L)
```

### Arguments

preprocess	normalization
input_size	input size for machine learning model
epochs	maximum number of epochs

### Value

a ts\_conv1d object.

### Examples

```
#Use the same example of ts_mlp changing the constructor to:
model <- ts_conv1d(ts_norm_gminmax(), input_size=4, epochs = 10000L)
```

---

 ts\_data

*ts\_data*


---

### Description

Time series data structure used in DAL Toolbox. It receives a vector (representing a time series) or a matrix y (representing a sliding windows). Internal ts\_data is matrix of sliding windows with size sw. If sw equals to zero, it store a time series as a single matrix column.

### Usage

```
ts_data(y, sw = 1)
```

### Arguments

y	output variable
sw	integer: sliding window size.



**Value**

a ts\_data object.

**Examples**

```
data(sin_data)
head(sin_data)

data <- ts_data(sin_data$y)
ts_head(data)

data10 <- ts_data(sin_data$y, 10)
ts_head(data10)
```

---

ts\_elm

*ELM*


---

**Description**

Creates a time series prediction object that uses the Extreme Learning Machine (ELM). It wraps the elmNNRcpp library.

**Usage**

```
ts_elm(preprocess = NA, input_size = NA, nhid = NA, actfun = "purelin")
```

**Arguments**

preprocess	normalization
input_size	input size for machine learning model
nhid	ensemble size
actfun	defines the type to use, possible values: 'sig', 'radbas', 'tribas', 'relu', 'purelin' (default).

**Value**

a ts\_elm object.

**Examples**

```
data(sin_data)
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)

samp <- ts_sample(ts, test_size = 5)
io_train <- ts_projection(samp$train)
io_test <- ts_projection(samp$test)
```

```

model <- ts_elm(ts_norm_gminmax(), input_size=4, nhid=3, actfun="purelin")
model <- fit(model, x=io_train$input, y=io_train$output)

prediction <- predict(model, x=io_test$input[1,], steps_ahead=5)
prediction <- as.vector(prediction)
output <- as.vector(io_test$output)

ev_test <- evaluate(model, output, prediction)
ev_test

```

---

ts_head	<i>ts_head</i>
---------	----------------

---

## Description

Returns the first n observations from a ts\_data

## Usage

```
ts_head(x, n = 6L, ...)
```

## Arguments

x	ts_data
n	number of rows to return
...	optional arguments

## Value

The first n observations of a ts\_data

## Examples

```

data(sin_data)
data10 <- ts_data(sin_data$y, 10)
ts_head(data10)

```

---

ts_knn	<i>knn time series prediction</i>
--------	-----------------------------------

---

## Description

Creates a prediction object that uses the K-Nearest Neighbors (knn) method for time series regression

## Usage

```
ts_knn(preprocess = NA, input_size = NA, k = NA)
```

## Arguments

preprocess	normalization
input_size	input size for machine learning model
k	number of k neighbors

## Value

a ts\_knn object.

## Examples

```
data(sin_data)
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)

samp <- ts_sample(ts, test_size = 5)
io_train <- ts_projection(samp$train)
io_test <- ts_projection(samp$test)

model <- ts_knn(ts_norm_gminmax(), input_size=4, k=3)
model <- fit(model, x=io_train$input, y=io_train$output)

prediction <- predict(model, x=io_test$input[1,], steps_ahead=5)
prediction <- as.vector(prediction)
output <- as.vector(io_test$output)

ev_test <- evaluate(model, output, prediction)
ev_test
```

---

ts_lstm	<i>LSTM</i>
---------	-------------

---

**Description**

Creates a time series prediction object that uses the LSTM. It wraps the pytorch library.

**Usage**

```
ts_lstm(preprocess = NA, input_size = NA, epochs = 10000L)
```

**Arguments**

preprocess	normalization
input_size	input size for machine learning model
epochs	maximum number of epochs

**Value**

a ts\_lstm object.

**Examples**

```
#Use the same example of ts_mlp changing the constructor to:
model <- ts_lstm(ts_norm_gminmax(), input_size=4, epochs = 10000L)
```

---

ts_mlp	<i>MLP</i>
--------	------------

---

**Description**

Creates a time series prediction object that uses the Multilayer Perceptron (MLP). It wraps the nnet library.

**Usage**

```
ts_mlp(preprocess = NA, input_size = NA, size = NA, decay = 0.01, maxit = 1000)
```

**Arguments**

preprocess	normalization
input_size	input size for machine learning model
size	number of neurons inside hidden layer
decay	decay parameter for MLP
maxit	maximum number of iterations

**Value**

a ts\_mlp object.

**Examples**

```
data(sin_data)
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)

samp <- ts_sample(ts, test_size = 5)
io_train <- ts_projection(samp$train)
io_test <- ts_projection(samp$test)

model <- ts_mlp(ts_norm_gminmax(), input_size=4, size=4, decay=0)
model <- fit(model, x=io_train$input, y=io_train$output)

prediction <- predict(model, x=io_test$input[1,], steps_ahead=5)
prediction <- as.vector(prediction)
output <- as.vector(io_test$output)

ev_test <- evaluate(model, output, prediction)
ev_test
```

---

ts\_norm\_an

*Time Series Adaptive Normalization*

---

**Description**

Transform data to a common scale while taking into account the changes in the statistical properties of the data over time.

**Usage**

```
ts_norm_an(remove_outliers = TRUE, nw = 0)
```

**Arguments**

remove\_outliers  
logical: if TRUE (default) outliers will be removed.

nw  
integer: window size.

**Value**

a ts\_norm\_an object.

**Examples**

```
# time series to normalize
data(sin_data)

# convert to sliding windows
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)
summary(ts[,10])

# normalization
preproc <- ts_norm_an()
preproc <- fit(preproc, ts)
tst <- transform(preproc, ts)
ts_head(tst, 3)
summary(tst[,10])
```

---

ts\_norm\_diff

*Time Series Diff*

---

**Description**

It receives as parameter the variable `remove_outliers`. This function calculates the difference between the values of a time series

**Usage**

```
ts_norm_diff(remove_outliers = TRUE)
```

**Arguments**

`remove_outliers`  
logical: if TRUE (default) outliers will be removed.

**Value**

a `ts_norm_diff` object.

**Examples**

```
# time series to normalize
data(sin_data)

# convert to sliding windows
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)
summary(ts[,10])

# normalization
preproc <- ts_norm_diff()
```

```
preproc <- fit(preproc, ts)
tst <- transform(preproc, ts)
ts_head(tst, 3)
summary(tst[,9])
```

---

ts_norm_ean	<i>Time Series Adaptive Normalization (Exponential Moving Average - EMA)</i>
-------------	--

---

### Description

It takes 2 parameters: remove\_outliers and nw

### Usage

```
ts_norm_ean(remove_outliers = TRUE, nw = 0)
```

### Arguments

remove_outliers	logical: if TRUE (default) outliers will be removed.
nw	windows size

### Value

a ts\_norm\_ean object.

### Examples

```
# time series to normalize
data(sin_data)

# convert to sliding windows
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)
summary(ts[,10])

# normalization
preproc <- ts_norm_ean()
preproc <- fit(preproc, ts)
tst <- transform(preproc, ts)
ts_head(tst, 3)
summary(tst[,10])
```

---

ts\_norm\_gminmax      *Time Series Global Min-Max*

---

**Description**

Rescales data, so the minimum value is mapped to 0 and the maximum value is mapped to 1.

**Usage**

```
ts_norm_gminmax(remove_outliers = TRUE)
```

**Arguments**

remove\_outliers  
logical: if TRUE (default) outliers will be removed.

**Value**

a ts\_norm\_gminmax object.

**Examples**

```
# time series to normalize
data(sin_data)

# convert to sliding windows
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)
summary(ts[,10])

# normalization
preproc <- ts_norm_gminmax()
preproc <- fit(preproc, ts)
tst <- transform(preproc, ts)
ts_head(tst, 3)
summary(tst[,10])
```

---

ts\_norm\_swminmax      *Time Series Sliding Window Min-Max*

---

**Description**

It takes as parameter the variable remove\_outliers. The ts\_norm\_swminmax function creates an object for normalizing a time series based on the "sliding window min-max scaling" method

**Usage**

```
ts_norm_swminmax(remove_outliers = TRUE)
```



**Arguments**

remove\_outliers  
logical: if TRUE (default) outliers will be removed.

**Value**

a ts\_norm\_swminmax object.

**Examples**

```
# time series to normalize
data(sin_data)

# convert to sliding windows
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)
summary(ts[,10])

# normalization
preproc <- ts_norm_swminmax()
preproc <- fit(preproc, ts)
tst <- transform(preproc, ts)
ts_head(tst, 3)
summary(tst[,10])
```

---

ts\_projection

*Time Series Projection*

---

**Description**

Separates the ts\_data into input and output.

**Usage**

```
ts_projection(ts)
```

**Arguments**

ts                   matrix or data.frame containing the time series.

**Value**

a ts\_projection object.

**Examples**

```
#setting up a ts_data
data(sin_data)
ts <- ts_data(sin_data$y, 10)

io <- ts_projection(ts)

#input data
ts_head(io$input)

#output data
ts_head(io$output)
```

---

ts_reg	<i>TSReg</i>
--------	--------------

---

**Description**

Time Series Regression directly from time series Ancestral class for non-sliding windows implementation.

**Usage**

```
ts_reg()
```

**Value**

A ts\_reg object

**Examples**

```
#See ?ts_arma for an example using Auto-regressive Integrated Moving Average
```

---

ts_regsw	<i>TSRegSW</i>
----------	----------------

---

**Description**

Time Series Regression from Sliding Windows. Ancestral class for Machine Learning Implementation.

**Usage**

```
ts_regsw(preprocess = NA, input_size = NA)
```

**Arguments**

preprocess      normalization  
input\_size      input size for machine learning model

**Value**

A ts\_regs object

**Examples**

#See ?ts\_elm for an example using Extreme Learning Machine

---

ts_rf	<i>Random Forest</i>
-------	----------------------

---

**Description**

Creates a time series prediction object that uses the Random Forest. It wraps the randomForest library.

**Usage**

```
ts_rf(preprocess = NA, input_size = NA, nodesize = 1, ntree = 10, mtry = NULL)
```

**Arguments**

preprocess      normalization  
input\_size      input size for machine learning model  
nodesize        node size  
ntree            number of trees  
mtry            number of attributes to build tree

**Value**

a ts\_rf object.

**Examples**

```
data(sin_data)
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)
```

```
samp <- ts_sample(ts, test_size = 5)
io_train <- ts_projection(samp$train)
io_test <- ts_projection(samp$test)
```

```
model <- ts_rf(ts_norm_gminmax(), input_size=4, nodesize=3, ntree=50)
```

```

model <- fit(model, x=io_train$input, y=io_train$output)

prediction <- predict(model, x=io_test$input[1,], steps_ahead=5)
prediction <- as.vector(prediction)
output <- as.vector(io_test$output)

ev_test <- evaluate(model, output, prediction)
ev_test

```

---

ts\_sample

*Time Series Sample*


---

### Description

Separates the `ts_data` into training and test. It separates the test size from the last observations minus an offset. The offset is important to allow replication under different recent origins. The data for test uses the number of rows of a `ts_data` minus the test size and offset.

### Usage

```
ts_sample(ts, test_size = 1, offset = 0)
```

### Arguments

<code>ts</code>	time series data.
<code>test_size</code>	integer: size of test data (default = 1).
<code>offset</code>	integer: starting point (default = 0).

### Value

A list with the two samples

### Examples

```

#setting up a ts_data
data(sin_data)
ts <- ts_data(sin_data$y, 10)

#separating into train and test
test_size <- 3
samp <- ts_sample(ts, test_size)

#first five rows from training data
ts_head(samp$train, 5)

#last five rows from training data
ts_head(samp$train[-c(1:(nrow(samp$train)-5)),])

#testing data
ts_head(samp$test)

```

---

ts_svm	SVM
--------	-----

---

### Description

Creates a time series prediction object that uses the Support Vector Machine (SVM). It wraps the e1071 library.

### Usage

```
ts_svm(
  preprocess = NA,
  input_size = NA,
  kernel = "radial",
  epsilon = 0,
  cost = 10
)
```

### Arguments

preprocess	normalization
input_size	input size for machine learning model
kernel	SVM kernel (linear, radial, polynomial, sigmoid)
epsilon	error threshold
cost	cost

### Value

a ts\_svm object.

### Examples

```
data(sin_data)
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)

samp <- ts_sample(ts, test_size = 5)
io_train <- ts_projection(samp$train)
io_test <- ts_projection(samp$test)

model <- ts_svm(ts_norm_gminmax(), input_size=4)
model <- fit(model, x=io_train$input, y=io_train$output)

prediction <- predict(model, x=io_test$input[1,], steps_ahead=5)
prediction <- as.vector(prediction)
output <- as.vector(io_test$output)

ev_test <- evaluate(model, output, prediction)
ev_test
```

---

`ts_tune`*Time Series Tune*

---

**Description**

Time Series Tune

**Usage**

```
ts_tune(input_size, base_model, folds = 10)
```

**Arguments**

<code>input_size</code>	input size for machine learning model
<code>base_model</code>	base model for tuning
<code>folds</code>	number of folds for cross-validation

**Value**a `ts_tune` object.**Examples**

```
data(sin_data)
ts <- ts_data(sin_data$y, 10)
ts_head(ts, 3)

samp <- ts_sample(ts, test_size = 5)
io_train <- ts_projection(samp$train)
io_test <- ts_projection(samp$test)

tune <- ts_tune(input_size=c(3:5), base_model = ts_elm(ts_norm_gminmax()))
ranges <- list(nhid = 1:5, actfun=c('purelin'))

# Generic model tuning
model <- fit(tune, x=io_train$input, y=io_train$output, ranges)

prediction <- predict(model, x=io_test$input[1,], steps_ahead=5)
prediction <- as.vector(prediction)
output <- as.vector(io_test$output)

ev_test <- evaluate(model, output, prediction)
ev_test
```

---

zscore	<i>z-score normalization</i>
--------	------------------------------

---

**Description**

Scale data using z-score normalization.  $zscore = (x - \text{mean}(x))/sd(x)$ .

**Usage**

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

**Arguments**

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

**Value**

z-score transformation object

**Examples**

```
data(iris)
head(iris)

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

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

---

[.ts_data	<i>Extract a subset of a time series stored in an object</i>
-----------	--

---

**Description**

Receives as parameters the variables x, i, j ...

**Usage**

```
## S3 method for class 'ts_data'
x[i, j, ...]
```

**Arguments**

x	input variable
i	row i
j	column j
...	optional arguments

**Value**

A new ts\_data object

**Examples**

```
data(sin_data)
data10 <- ts_data(sin_data$y, 10)
ts_head(data10)
#single line
data10[12,]

#range of lines
data10[12:13,]

#single column
data10[,1]

#range of columns
data10[,1:2]

#range of rows and columns
data10[12:13,1:2]

#single line and a range of columns
#'data10[12,1:2]

#range of lines and a single column
data10[12:13,1]

#single observation
data10[12,1]
```



# Index

## \* datasets

- Boston, 9
- sin\_data, 58
- [.ts\_data, 79

action, 4

action.dal\_transform, 5

adjust\_class\_label, 5

adjust\_data.frame, 6

adjust\_factor, 6

adjust\_matrix, 7

adjust\_ts\_data, 7

autoenc\_encode, 8

autoenc\_encode\_decode, 9

Boston, 9

categ\_mapping, 11

cla\_dtree, 12

cla\_knn, 13

cla\_majority, 14

cla\_mlp, 15

cla\_nb, 16

cla\_rf, 16

cla\_svm, 17

cla\_tune, 18

classification, 11

clu\_tune, 22

cluster, 19

cluster\_dbscan, 20

cluster\_kmeans, 21

cluster\_pam, 22

clusterer, 20

dal\_base, 23

dal\_learner, 24

dal\_transform, 24

dal\_tune, 25

data\_sample, 25

do\_fit, 26

do\_predict, 27

dt\_pca, 27

evaluate, 28

fit, 29

fit.cla\_tune, 29

fit.cluster\_dbscan, 30

fit\_curvature\_max, 30

fit\_curvature\_min, 31

inverse\_transform, 32

k\_fold, 32

minmax, 33

MSE.ts, 34

outliers, 34

plot\_bar, 35

plot\_boxplot, 36

plot\_boxplot\_class, 36

plot\_density, 37

plot\_density\_class, 38

plot\_groupedbar, 39

plot\_hist, 39

plot\_lollipop, 40

plot\_pieplot, 41

plot\_points, 42

plot\_radar, 43

plot\_scatter, 43

plot\_series, 44

plot\_stackedbar, 45

plot\_ts, 45

plot\_ts\_pred, 46

predictor, 47

reg\_dtree, 49

reg\_knn, 49

reg\_mlp, 50

reg\_rf, [51](#)  
reg\_svm, [52](#)  
reg\_tune, [53](#)  
regression, [48](#)

sample\_random, [54](#)  
sample\_stratified, [54](#)  
select\_hyper, [55](#)  
select\_hyper.cla\_tune, [56](#)  
select\_hyper.ts\_tune, [56](#)  
set\_params, [57](#)  
set\_params.default, [57](#)  
sin\_data, [58](#)  
sMAPE.ts, [58](#)  
smoothing, [59](#)  
smoothing\_cluster, [59](#)  
smoothing\_freq, [60](#)  
smoothing\_inter, [61](#)

train\_test, [61](#)  
train\_test\_from\_folds, [62](#)  
transform, [62](#)  
ts\_arima, [63](#)  
ts\_conv1d, [64](#)  
ts\_data, [64](#)  
ts\_elm, [65](#)  
ts\_head, [66](#)  
ts\_knn, [67](#)  
ts\_lstm, [68](#)  
ts\_mlp, [68](#)  
ts\_norm\_an, [69](#)  
ts\_norm\_diff, [70](#)  
ts\_norm\_ean, [71](#)  
ts\_norm\_gminmax, [72](#)  
ts\_norm\_swminmax, [72](#)  
ts\_projection, [73](#)  
ts\_reg, [74](#)  
ts\_regsw, [74](#)  
ts\_rf, [75](#)  
ts\_sample, [76](#)  
ts\_svm, [77](#)  
ts\_tune, [78](#)

zscore, [79](#)