Package: certestats (via r-universe)

September 11, 2024

Title A Certe R Package for Statistical Modelling

Version 1.20.2

Description A Certe R Package for early-warning, applying statistical modelling (such as creating machine learning models), QC rules and distribution analysis. This package is part of the 'certedata' universe.

URL https://certe-medical-epidemiology.github.io/certestats, https://github.com/certe-medical-epidemiology/certestats

Depends R (>= 4.1.0)

Imports certestyle, AMR (>= 2.1.0), broom (>= 1.0.0), cli (>= 3.6.0), dials (>= 1.0.0), dplyr (>= 1.1.0), generics (>= 0.1.0), ggplot2 (>= 3.4.0), hardhat (>= 1.2.0), lubridate (>= 1.9.0), parsnip (>= 1.0.0), recipes (>= 1.0.0), rsample (>= 1.0.0), tibble (>= 1.0.0), tidyr (>= 1.0.0), tune (>= 1.0.0), workflows (>= 1.0.0), yaml (>= 2.2.0), yardstick (>= 1.0.0)

Suggests censored (>= 0.1.0), certeplot2, kknn, mice (>= 3.7.5), nnet, plot2, progress, ranger, rlang, rpart, survival, testthat (>= 2.0.0), xgboost

License GPL-2

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Config/testthat/edition 2

LazyData true

Repository https://certe-medical-epidemiology.r-universe.dev

RemoteUrl https://github.com/certe-medical-epidemiology/certestats

RemoteRef HEAD

RemoteSha 799ceb77293df9fa38afb069ea9af31b87927a4c

Contents

`confusion_matrix` *Confusion Matrix Metrics*

Description

Create a confusion matrix and calculate its metrics. This function is an agnostic `yardstick` wrapper: it applies all `yardstick` functions that are metrics used for confusion matrices without internal hard-coded function names.

Usage

```
confusion_matrix(data, ...)

## Default S3 method:
confusion_matrix(data, truth, estimate, na.rm = getOption("na.rm", FALSE), ...)
```

Arguments

`data`	Either a `data.frame` containing the columns specified by the `truth` and `estimate` arguments, or a `table/matrix` where the true class results should be in the columns of the table.
`...`	Not currently used.
`truth`	The column identifier for the true class results (that is a `factor`). This should be an unquoted column name although this argument is passed by expression and supports quasiquotation (you can unquote column names). For `_vec()` functions, a `factor` vector.

estimate	The column identifier for the predicted class results (that is also factor). As with truth this can be specified different ways but the primary method is to use an unquoted variable name. For _vec() functions, a factor vector.
na.rm	a logical to indicate whether empty must be removed

Default values of na.rm

This 'certestats' package supports a global default setting for na.rm in many mathematical functions. This can be set with options(na.rm = TRUE) or options(na.rm = FALSE).

For normality(), quantile() and IQR(), this also applies to the type argument. The default, type = 7, is the default of base R. Use type = 6 to comply with SPSS.

Examples

```
df <- tibble::tibble(name = c("Predict Yes", "Predict No"),
                     "Actual Yes" = c(123, 26),
                     "Actual No" = c(13, 834))
df
confusion_matrix(df)
```

---

different_means *Different Means*

Description

Functions to determine harmonic and geometric means.

Usage

```
mean_harmonic(x, ..., na.rm = getOption("na.rm", FALSE))

mean_geometric(x, ..., na.rm = getOption("na.rm", FALSE))
```

Arguments

x	numeric vector
...	arguments passed on to base::mean()
na.rm	ignore empty values

Details

The harmonic mean can be expressed as the reciprocal of the arithmetic mean of the reciprocals.

The geometric mean is defined as the nth root of the product of n numbers.

distribution_metrics *Distribution Metrics*

Description

These are simple distribution metric functions.

Usage

```
se(x, na.rm = getOption("na.rm", FALSE))

ci(x, level = 0.95, na.rm = getOption("na.rm", FALSE))

sum_of_squares(x, correct_mean = TRUE, na.rm = getOption("na.rm", FALSE))

cv(x, na.rm = getOption("na.rm", FALSE))

cqv(x, na.rm = getOption("na.rm", FALSE))

mse(actual, predicted, na.rm = getOption("na.rm", FALSE))

mape(actual, predicted, na.rm = getOption("na.rm", FALSE))

rmse(actual, predicted, na.rm = getOption("na.rm", FALSE))

mae(actual, predicted, na.rm = getOption("na.rm", FALSE))

z_score(x, na.rm = getOption("na.rm", FALSE))

midhinge(x, na.rm = getOption("na.rm", FALSE))

ewma(x, lambda, na.rm = getOption("na.rm", FALSE))

rr_ewma(x, lambda, na.rm = getOption("na.rm", FALSE))

normalise(n, n_ref, per = 1000)

normalize(n, n_ref, per = 1000)

scale_sd(x)

centre_mean(x)

percentiles(x, na.rm = getOption("na.rm", FALSE))

deciles(x, na.rm = getOption("na.rm", FALSE))
```

Arguments

x	values
na.rm	a logical to indicate whether empty must be removed from x
level	alpha level, defaults to 95%
correct_mean	with TRUE (the default) correct for the mean will be applied, by summing each square of x after the mean of x has been subtracted, so that this says something about x. With FALSE, all x^2 are simply added together, so this says something about x's location in the data.
actual	Vector of actual values
predicted	Vector of predicted values
lambda	smoothing parameter, a value between 0 and 1. A value of 0 is equal to x, a value of 1 equal to the *mean* of x. The EWMA looks back and has a delay - the rrEWMA takes the mean of a 'forward' and 'backward' EWMA.
n	number to be normalised
n_ref	reference to use for normalisation
per	normalisation factor

Details

These are the explanations of the functions:

- se() calculates the standard error: sd / square root of length
- ci() calculates the confidence intervals for a mean (defaults at 95%), which returns length 2
- sum_of_squares() calculates the sum of (x - mean(x)) ^ 2
- cv() calculates the coefficient of variation: standard deviation / mean
- cqv() calculates the coefficient of quartile variation: (Q3 - Q1) / (Q3 + Q1)
- mse() calculates the mean squared error
- mape() calculates the mean absolute percentage error
- rmse() calculates the root mean squared error
- mae() calculates the mean absolute error
- z_score() calculates the number of standard deviations from the mean: (x - mean) / sd
- midhinge() calculates the mean of interquartile range: (Q1 + Q3) / 2
- ewma() calculates the EWMA (exponentially weighted moving average)
- rr_ewma() calculates the rrEWMA (reversed-recombined exponentially weighted moving average)

- `normalise()` normalises the data based on a reference: (n / reference) * unit
- `scale_sd()` normalises the data to have a standard deviation of 1, while retaining the mean
- `centre_mean()` normalises the data to have a mean of 0, while retaining the standard deviation
- `percentiles()` and `deciles()` take a numeric vector as input, and return the lowest percentiles or deciles for each value

Default values of `na.rm`

This 'certestats' package supports a global default setting for `na.rm` in many mathematical functions. This can be set with `options(na.rm = TRUE)` or `options(na.rm = FALSE)`.

For `normality()`, `quantile()` and `IQR()`, this also applies to the `type` argument. The default, `type = 7`, is the default of base R. Use `type = 6` to comply with SPSS.

See Also

For the sum of squares: https://www.thoughtco.com/sum-of-squares-formula-shortcut-3126266

Examples

```
x <- c(0, 1, 2, 3, 4, 5, 5, 5, 5, 5, 5, 5, 6)
percentiles(x)
deciles(x)

percentiles(rnorm(10))

library(dplyr, warn.conflicts = FALSE)
tib <- as_tibble(matrix(as.integer(runif(40, min = 1, max = 7)), ncol = 4),
                 .name_repair = function(...) LETTERS[1:4])
tib

# percentiles per column
tib |> mutate_all(percentiles)
```

`early_warning_biomarker` *Early Warning for Biomarkers*

Description

The `early_warning_biomarker` function is designed to detect unexpected changes in biomarker values for individual patients based on clinical chemistry data. It will flag cases where patients' biomarker results deviate from defined thresholds or exhibit significant changes within a specified time window.

Usage

```
early_warning_biomarker(
  df,
  testcode = NULL,
  ...,
  column_date = NULL,
  column_patientid = NULL,
  column_testcode = NULL,
  column_testresult = NULL,
  threshold_min = NULL,
  threshold_max = NULL,
  window_days = NULL,
  max_delta_absolute = NULL,
  max_delta_relative = NULL,
  direction = "any"
)
```

Arguments

df	A data frame containing clinical chemistry data with columns for patient ID, date, test code, and test result.
testcode	Value of the column containing test codes to filter on.
...	Filter arguments, e.g. `testcode == "eGFR"`
column_date	Name of the column to use for dates. If left blank, the first date column will be used.
column_patientid	Name of the column to use for patient IDs. If left blank, the first column resembling `"patient|patid"` will be used.
column_testcode	Name of the column containing test codes.
column_testresult	Name of the column containing test results.
threshold_min	Minimum threshold for biomarkers.
threshold_max	Maximum threshold for biomarkers.
window_days	Number of days for the time window to check for changes.
max_delta_absolute	Maximum allowable absolute change in biomarkers.
max_delta_relative	Maximum allowable relative change in biomarkers.
direction	Direction of change to check ("up", "down", or "any").

Details

This whole function, including the documentation, was written by ChatGPT 3.5 in October 2023. Only minor changes were applied manually.

This function is particularly useful in early detection of anomalous biomarker results, which can be indicative of health issues or treatment response. By providing detailed flags, it allows healthcare professionals and researchers to take timely action, conduct further investigations, or make informed clinical decisions.

The output of this function can be utilised for:

- Generating patient-specific reports for healthcare providers.
- Identifying trends and patterns in biomarker changes for research purposes.
- Enhancing patient care by enabling proactive interventions when necessary.
- Supporting data-driven clinical epidemiology studies and research.

The `format()` function allows you to format the results of the `early_warning_biomarker()` function for better readability and analysis. It organises the flag information into a structured data frame for easier inspection.

Value

A list with the following components:

- `flags`: A list of flags per patient, containing data frames for each patient with details on dates, test codes, test results, and flagging criteria. The structure of each data frame includes the following columns:
  - `patient`: Patient identifier.
  - `date`: Date of the biomarker measurement.
  - `testcode`: Code of the biomarker test.
  - `testresult`: Biomarker test result.
  - `delta_absolute`: Absolute change in biomarker values.
  - `delta_relative`: Relative change in biomarker values.
  - `threshold_min_flag`: A logical flag indicating if the threshold minimum is exceeded.
  - `threshold_max_flag`: A logical flag indicating if the threshold maximum is exceeded.
  - `delta_absolute_flag`: A logical flag indicating if the absolute change exceeds the threshold.
  - `delta_relative_flag`: A logical flag indicating if the relative change exceeds the threshold.
- `details`: A data frame containing all patient details and calculated flags, regardless of whether they meet the flagging criteria. The data frame includes the same columns as the individual patient data frames.

See Also

`early_warning_cluster()`

Examples

```
data <- data.frame(date = Sys.Date() + 1:10,
                   patient = "test",
                   value = c(10,12,14,15,13,21,22,19,14,12))
```

```
check <- data |> early_warning_biomarker(window_days = 6, max_delta_absolute = 10)

check

unlist(check)
```

`early_warning_cluster` *Early Warning for Disease Clusters*

Description

Detect disease clusters with `early_warning_cluster()`. Use `has_clusters()` to return `TRUE` or `FALSE` based on its output, or employ `format()` to format the result.

Usage

```
early_warning_cluster(
  df,
  column_date = NULL,
  column_patientid = NULL,
  based_on_historic_maximum = FALSE,
  period_length_months = 12,
  minimum_cases = 5,
  minimum_days = 0,
  minimum_case_days = 2,
  minimum_case_fraction_in_period = 0.02,
  threshold_percentile = 97.5,
  remove_outliers = TRUE,
  remove_outliers_coefficient = 1.5,
  moving_average_days = 7,
  moving_average_side = "left",
  case_free_days = 14,
  ...
)

n_clusters(x)

has_clusters(x, n = 1)

has_ongoing_cluster(x, dates = Sys.Date() - 1)

has_cluster_before(x, date)

has_cluster_after(x, date)
```

Arguments

df	Data set: This must consist of **only positive results**. The minimal data set should include a date column and a patient column. Do not summarize on patient IDs; this will be handled automatically.
column_date	Name of the column to use for dates. If left blank, the first date column will be used.
column_patientid	Name of the column to use for patient IDs. If left blank, the first column resembling "patient|patid" will be used.
based_on_historic_maximum	A logical to indicate whether the cluster detection should be based on the maximum of previous years. The default is FALSE, which uses all historic data points.
period_length_months	Number of months per period.
minimum_cases	Minimum number of *cases* that a cluster requires to be considered a cluster.
minimum_days	Minimum number of *days* that a cluster requires to be considered a cluster.
minimum_case_days	Minimum number of *days with cases* that a cluster requires to be considered a cluster.
minimum_case_fraction_in_period	Minimum fraction of *cluster cases in a period* that a cluster requires to be considered a cluster.
threshold_percentile	Threshold to set.
remove_outliers	A logical to indicate whether outliers should be removed before determining the threshold.
remove_outliers_coefficient	Coefficient used for outlier determination.
moving_average_days	Number of days to set in moving_average(). Defaults to a whole week (7).
moving_average_side	Side of days to set in moving_average(). Defaults to "left" for retrospective analysis.
case_free_days	Number of days to set in get_episode().
...	not used at the moment
x	output of early_warning_cluster()
n	number of clusters, defaults to 1
dates	date(s) to test whether any of the clusters currently has this date in it, defaults to yesterday.
date	date to test whether there are any clusters since or until this date.

Details

A (disease) cluster is defined as an unusually large aggregation of disease events in time or space (ATSDR, 2008). They are common, particularly in large populations. From a statistical standpoint, it is nearly inevitable that some clusters of chronic diseases will emerge within various communities, be it schools, church groups, social circles, or neighborhoods. Initially, these clusters are often perceived as products of specific, predictable processes rather than random occurrences in a particular location, akin to a coin toss.

Whether a (suspected) cluster corresponds to an actual increase of disease in the area, needs to be assessed by an epidemiologist or biostatistician (ATSDR, 2008).

The function `has_ongoing_cluster()` returns a logical vector with the same length as `dates`, so `dates` can have any length.

See Also

`early_warning_biomarker()`

Examples

```
cases <- data.frame(date = sample(seq(as.Date("2015-01-01"),
                                      as.Date("2022-12-31"),
                                      "1 day"),
                                  size = 300),
                    patient = sample(LETTERS, size = 300, replace = TRUE))

# ------------------------------------------------------------

check <- early_warning_cluster(cases, threshold_percentile = 0.99)

has_clusters(check)
check


check2 <- early_warning_cluster(cases,
                                minimum_cases = 1,
                                threshold_percentile = 0.75)

check2
check2 |> format()

check2 |> n_clusters()
check2 |> has_clusters()
check2 |> has_clusters(n = 15)

check2 |> has_ongoing_cluster("2022-06-01")
check2 |> has_ongoing_cluster(c("2022-06-01", "2022-06-20"))
check2 |> has_cluster_before("2022-06-01")
check2 |> has_cluster_after("2022-06-01")

check2 |> unclass()
```

`esbl_tests` *Example Data Set with ESBL Test Outcomes*

Description

This data set contains phenotypic test outcomes of the presence of ESBL (Extended Spectrum Beta-Lactamase) genes, with the minimum inhibitory concentrations (MIC, in mg/L) for 17 antibiotic drugs.

Usage

```
esbl_tests
```

Format

A tibble/data.frame with 500 observations and 19 variables:

- `esbl`: Outcome of ESBL test (`TRUE` (n = 250), `FALSE` (n = 250))
- `genus`: Taxonomic genus of the bacteria (*Citrobacter* (n = 42), *Enterobacter* (n = 30), *Escherichia* (n = 301), *Klebsiella* (n = 70), *Morganella* (n = 24), *Proteus* (n = 33))
- `AMC`: MIC values of amoxicillin/clavulanic acid
- `AMP`: MIC values of ampicillin
- `TZP`: MIC values of piperacillin/tazobactam
- `CXM`: MIC values of cefuroxime
- `FOX`: MIC values of cefoxitin
- `CTX`: MIC values of cefotaxime
- `CAZ`: MIC values of ceftazidime
- `GEN`: MIC values of gentamicin
- `TOB`: MIC values of tobramycin
- `TMP`: MIC values of trimethoprim
- `SXT`: MIC values of trimethoprim/sulfamethoxazole
- `NIT`: MIC values of nitrofurantoin
- `FOS`: MIC values of fosfomycin
- `CIP`: MIC values of ciprofloxacin
- `IPM`: MIC values of imipenem
- `MEM`: MIC values of meropenem
- `COL`: MIC values of colistin

Examples

```
print(esbl_tests, n = 5)
```

impute *Impute: Filling Missing Values*

Description

Imputation is the process of replacing missing data with substituted values. This is done because of three main problems that missing data causes: missing data can introduce a substantial amount of bias, make the handling and analysis of the data more arduous, and create reductions in efficiency.

Usage

```
impute(
  .data,
  vars = everything(),
  algorithm = "mice",
  m = 10,
  method = NULL,
  FUN = median,
  info = TRUE,
  ...
)

is_imputed(.data)

get_mice(.data)
```

Arguments

`.data`	data set with missing values to impute
`vars`	variables of `.data` that must be imputed, defaults to `everything()` and supports the `tidyselect` language.
`algorithm`	algorithm to use for imputation, must be `"mice"` or `"single-point"`, see *Details*. For the latter, `FUN` must be given.
`m`	number of multiple imputations if using MICE, see `mice::mice()`. The mean of all imputations will be used as result.
`method`	method to use if using MICE, see `mice::mice()`
`FUN`	function to use for single-point imputation (directly) or for MICE to summarise the results over all `m` iterations
`info`	print info about imputation
`...`	arguments to pass on to `mice::mice()`

Details

Imputation can be done using single-point, such as the mean or the median, or using Multivariate Imputations by Chained Equations (MICE). Using MICE is a lot more reliable, but also a lot slower, than single-point imputation.

The suggested and default method is MICE. The generated MICE object will be stored as an attribute with the data, and can be retrieved with `get_mice()`, containing all specifics about the imputation. MICE is also known as *fully conditional specification* and *sequential regression multiple imputation*. It was designed for data with randomly missing values, though there is simulation evidence to suggest that with a sufficient number of auxiliary variables it can also work on data that are missing not at random.

Use `is_imputed()` to get a data.frame with `TRUE`s for all values that were imputed.

Examples

```
iris2 <- dplyr::as_tibble(iris)
iris2[2, 2] <- NA
iris2[3, 3] <- NA
iris2[4, 5] <- NA
iris
iris2

result <- iris2 |> impute()
result

iris2 |> impute(algorithm = "single-point")
iris2 |>
  impute(vars = starts_with("Sepal"),
         algorithm = "single-point")
iris2 |>
  impute(vars = where(is.double),
         algorithm = "single-point",
         FUN = median)

result |> is_imputed()
result |> get_mice()
```

`machine_learning` *Create a Machine Learning (ML) Model*

Description

These functions can be used to create a machine learning model based on different 'engines' and to generalise predicting outcomes based on such models. These functions are wrappers around `tidymodels` packages (especially `parsnip`, `recipes`, `rsample`, `tune`, and `yardstick`) created by RStudio.

Usage

```
ml_decision_trees(
  .data,
  outcome,
  predictors = everything(),
  training_fraction = 0.75,
  strata = NULL,
  max_na_fraction = 0.01,
  correlation_filter = TRUE,
  centre = TRUE,
  scale = TRUE,
  engine = "rpart",
  mode = c("classification", "regression", "unknown"),
  tree_depth = 10,
  ...
)

ml_linear_regression(
  .data,
  outcome,
  predictors = everything(),
  training_fraction = 0.75,
  strata = NULL,
  max_na_fraction = 0.01,
  correlation_filter = TRUE,
  centre = TRUE,
  scale = TRUE,
  engine = "lm",
  mode = "regression",
  ...
)

ml_logistic_regression(
  .data,
  outcome,
  predictors = everything(),
  training_fraction = 0.75,
  strata = NULL,
  max_na_fraction = 0.01,
  correlation_filter = TRUE,
  centre = TRUE,
  scale = TRUE,
  engine = "glm",
  mode = "classification",
  penalty = 0.1,
  ...
)
```

```
ml_neural_network(
  .data,
  outcome,
  predictors = everything(),
  training_fraction = 0.75,
  strata = NULL,
  max_na_fraction = 0.01,
  correlation_filter = TRUE,
  centre = TRUE,
  scale = TRUE,
  engine = "nnet",
  mode = c("classification", "regression", "unknown"),
  penalty = 0,
  epochs = 100,
  ...
)

ml_nearest_neighbour(
  .data,
  outcome,
  predictors = everything(),
  training_fraction = 0.75,
  strata = NULL,
  max_na_fraction = 0.01,
  correlation_filter = TRUE,
  centre = TRUE,
  scale = TRUE,
  engine = "kknn",
  mode = c("classification", "regression", "unknown"),
  neighbors = 5,
  weight_func = "triangular",
  ...
)

ml_random_forest(
  .data,
  outcome,
  predictors = everything(),
  training_fraction = 0.75,
  strata = NULL,
  max_na_fraction = 0.01,
  correlation_filter = TRUE,
  centre = TRUE,
  scale = TRUE,
  engine = "ranger",
  mode = c("classification", "regression", "unknown"),
  trees = 2000,
  ...
```

```
)

ml_xg_boost(
  .data,
  outcome,
  predictors = everything(),
  training_fraction = 0.75,
  strata = NULL,
  max_na_fraction = 0.01,
  correlation_filter = TRUE,
  centre = TRUE,
  scale = TRUE,
  engine = "xgboost",
  mode = c("classification", "regression", "unknown"),
  trees = 2000,
  ...
)

## S3 method for class 'certestats_ml'
confusion_matrix(data, ...)

## S3 method for class 'certestats_ml'
autoplot(object, plot_type = "roc", ...)

## S3 method for class 'certestats_ml'
predict(object, new_data, type = NULL, ...)

apply_model_to(
  object,
  new_data,
  add_certainty = TRUE,
  only_prediction = FALSE,
  correct_mistakes = TRUE,
  impute_algorithm = "mice",
  ...
)

get_metrics(object)

get_accuracy(object)

get_kappa(object)

get_recipe(object)

get_specification(object)

get_rows_testing(object)
```

```
get_rows_training(object)

get_original_data(object)

get_roc_data(object)

get_coefficients(object)

get_model_variables(object)

get_variable_weights(object)

tune_parameters(object, ..., only_params_in_model = FALSE, levels = 5, v = 10)

## S3 method for class 'certestats_tuning'
autoplot(object, type = c("marginals", "parameters", "performance"), ...)

check_testing_predictions(object)
```

Arguments

`.data`	Data set to train
`outcome`	Outcome variable, also called the *response variable* or the *dependent variable*; the variable that must be predicted. The value will be evaluated in `select()` and thus supports the `tidyselect` language. In case of classification prediction, this variable will be coerced to a factor.
`predictors`	Explanatory variables, also called the *predictors* or the *independent variables*; the variables that are used to predict `outcome`. These variables will be transformed using `as.double()` (factors will be transformed to characters first). This value defaults to `everything()` and supports the `tidyselect` language.
`training_fraction`	Fraction of rows to be used for *training*, defaults to 75%. The rest will be used for *testing*. If given a number over 1, the number will be considered to be the required number of rows for *training*.
`strata`	A variable in `data` (single character or name) used to conduct stratified sampling. When not `NULL`, each resample is created within the stratification variable. Numeric `strata` are binned into quartiles.
`max_na_fraction`	Maximum fraction of `NA` values (defaults to `0.01`) of the `predictors` before they are removed from the model
`correlation_filter`	A logical to indicate whether the `predictors` should be removed that have to much correlation with each other, using `recipes::step_corr()`
`centre`	A logical to indicate whether the `predictors` should be transformed so that their mean will be `0`, using `recipes::step_center()`

scale — A logical to indicate whether the predictors should be transformed so that their standard deviation will be 1, using recipes::step_scale()

engine — R package or function name to be used for the model, will be passed on to parsnip::set_engine()

mode — Type of predicted value - defaults to "classification", but can also be "unknown" or "regression"

tree_depth — An integer for maximum depth of the tree.

... — Arguments to be passed on to the parsnip functions, see *Model Functions*.

For the tune_parameters() function, these must be dials package calls, such as dials::trees() (see Examples).

For predict(), these must be arguments passed on to parsnip::predict.model_fit()

penalty — A non-negative number representing the total amount of regularization (specific engines only).

epochs — An integer for the number of training iterations.

neighbors — A single integer for the number of neighbors to consider (often called k). For **kknn**, a value of 5 is used if neighbors is not specified.

weight_func — A *single* character for the type of kernel function used to weight distances between samples. Valid choices are: "rectangular", "triangular", "epanechnikov", "biweight", "triweight", "cos", "inv", "gaussian", "rank", or "optimal".

trees — An integer for the number of trees contained in the ensemble.

object, data — outcome of machine learning model

plot_type — the plot type, can be "roc" (default), "gain", "lift" or "pr". These functions rely on yardstick::roc_curve(), yardstick::gain_curve(), yardstick::lift_curve() and yardstick::pr_curve() to construct the curves.

new_data — A rectangular data object, such as a data frame.

type — A single character value or NULL. Possible values are "numeric", "class", "prob", "conf_int", "pred_int", "quantile", "time", "hazard", "survival", or "raw". When NULL, predict() will choose an appropriate value based on the model's mode.

add_certainty — a logical to indicate whether certainties should be added to the output data.frame

only_prediction — a logical to indicate whether predictions must be returned as vector, otherwise returns a data.frame

correct_mistakes — a logical to indicate whether missing variables and missing values should be added to new_data

impute_algorithm — the algorithm to use in impute() if correct_mistakes = TRUE. Can be "mice" (default) for the Multivariate Imputations by Chained Equations (MICE) algorithm, or "single-point" for a trained median.

only_params_in_model — a logical to indicate whether only parameters in the model should be tuned

`levels`	An integer for the number of values of each parameter to use to make the regular grid. `levels` can be a single integer or a vector of integers that is the same length as the number of parameters in `...`. `levels` can be a named integer vector, with names that match the id values of parameters.
v	The number of partitions of the data set.

Details

To predict **regression** (numeric values), the function `ml_logistic_regression()` cannot be used.

To predict **classifications** (character values), the function `ml_linear_regression()` cannot be used.

The workflow of the `ml_*()` functions is basically like this (thus saving a lot of `tidymodels` functions to type):

Use `autoplot()` on a model to plot the receiver operating characteristic (ROC) curve, the gain curve, the lift curve, or the precision-recall (PR) curve. For the ROC curve, the (overall) area under the curve (AUC) will be printed as subtitle.

The `predict()` function can be used to fit a model on a new data set. Its wrapper `apply_model_to()` works in the same way, but can also detect missing variables and missing data points within variables (and can add or fill them), and detects data type differences between the trained data and the input data.

Use the `get_model_variables()` function to return a zero-row data.frame with the variables that were used for training, even before the recipe steps.

Use the `get_variable_weights()` function to determine the (rough) estimated weights of each variable in the model. This is not as reliable as retrieving coefficients, but it does work for any

model. The weights are determined by running the model over all the highest and lowest values of each variable in the trained data. The function returns a data set with 1 row, of which the values sum up to 1.

Use the `tune_parameters()` function to analyse tune parameters of any `ml_*()` function. Without any parameters manually defined, it will try to tune all parameters of the underlying ML model. The tuning will be based on a V-fold cross-validation, of which the number of partitions can be set with `v`. The number of `levels` will be used to split the range of the parameters. For example, a range of 1-10 with `levels = 2` will lead to `[1, 10]`, while `levels = 5` will lead to `[1, 3, 5, 7, 9]`. The resulting data.frame will be sorted from best to worst. These results can also be plotted using `autoplot()`.

The `check_testing_predictions()` function combines the data used for testing from the original data with its predictions, so the original data can be reviewed per prediction.

Value

A machine learning model of class `certestats_ml` / `_rpart` / `model_fit`.

Attributes

The `ml_*()` functions return the following attributes:

- `properties`: a list with model properties: the ML function, engine package, training size, testing size, strata size, mode, and the different ML function-specific properties (such as `tree_depth` in `ml_decision_trees()`)
- `recipe`: a recipe as generated with `recipes::prep()`, to be used for training and testing
- `data_original`: a data.frame containing the original data, possibly without invalid strata
- `data_structure`: a data.frame containing the original data structure (only trained variables) with zero rows
- `data_means`: a data.frame containing the means of the original data (only trained variables)
- `data_training`: a data.frame containing the training data of `data_original`
- `data_testing`: a data.frame containing the testing data of `data_original`
- `rows_training`: an integer vector of rows used for training in `data_original`
- `rows_testing`: an integer vector of rows used for training in `data_original`
- `predictions`: a data.frame containing predicted values based on the testing data
- `metrics`: a data.frame with model metrics as returned by `yardstick::metrics()`
- `correlation_filter`: a logical indicating whether `recipes::step_corr()` has been applied
- `centre`: a logical indicating whether `recipes::step_center()` has been applied
- `scale`: a logical indicating whether `recipes::step_scale()` has been applied

Model Functions

These are the called functions from the `parsnip` package. Arguments set in `...` will be passed on to these `parsnip` functions:

- ml_decision_trees: parsnip::decision_tree()
- ml_linear_regression: parsnip::linear_reg()
- ml_logistic_regression: parsnip::logistic_reg()
- ml_neural_network: parsnip::mlp()
- ml_nearest_neighbour: parsnip::nearest_neighbor()
- ml_random_forest: parsnip::rand_forest()
- ml_xg_boost: parsnip::xgb_train()

Examples

```
# 'esbl_tests' is an included data set, see ?esbl_tests
print(esbl_tests, n = 5)

# predict ESBL test outcome based on MICs using 2 different models
model1 <- esbl_tests |> ml_xg_boost(esbl, where(is.double))
model2 <- esbl_tests |> ml_random_forest(esbl, where(is.double))

model1 |> get_metrics()
model2 |> get_metrics()

model1 |> confusion_matrix()


# Applying A Model ----------------------------------------------------

# simply use base R `predict()` to apply a model:
model1 |> predict(esbl_tests)

# but apply_model_to() contains more info and can apply corrections:
model1 |> apply_model_to(esbl_tests)
model1 |> apply_model_to(esbl_tests[, 1:15])
esbl_tests2 <- esbl_tests
esbl_tests2[2, "CIP"] <- NA
esbl_tests2[5, "AMC"] <- NA
# with XGBoost, nothing will be changed (it can correct for missings):
model1 |> apply_model_to(esbl_tests2)
# with random forest (or others), missings will be imputed:
model2 |> apply_model_to(esbl_tests2)


# Tuning A Model ------------------------------------------------------

# tune the parameters of a model (will take some time)
tuning <- model2 |>
  tune_parameters(v = 5, levels = 3)
autoplot(tuning)

# tuning analysis by specifying (some) parameters
iris |>
  ml_random_forest(Species) |>
```

```
  tune_parameters(mtry = dials::mtry(range = c(1, 3)),
                  trees = dials::trees())


# Practical Example #1 ------------------------------------------------

# this is what iris data set looks like:
head(iris)
# create a model to predict the species:
iris_model <- iris |> ml_xg_boost(Species)
iris_model_rf <- iris |> ml_random_forest(Species)
# is it a bit reliable?
get_metrics(iris_model)

# now try to predict species from an arbitrary data set:
to_predict <- data.frame(Sepal.Length = 5,
                         Sepal.Width = 3,
                         Petal.Length = 1.5,
                         Petal.Width = 0.5)
to_predict

# should be 'setosa' in the 'predicted' column:
iris_model |> apply_model_to(to_predict)

# which variables are generally important (only trained variables)?
iris_model |> get_variable_weights()

# how would the model do without the important 'Sepal.Length' column?
to_predict <- to_predict[, c("Sepal.Width", "Petal.Width", "Petal.Length")]
to_predict
iris_model |> apply_model_to(to_predict)

# now compare that with a random forest model that requires imputation:
iris_model_rf |> apply_model_to(to_predict)


# Practical Example #2 ------------------------------------------------

# this example shows plotting methods for a model

# train model to predict genus based on MICs:
genus <- esbl_tests |> ml_neural_network(genus, everything())
genus |> get_metrics()
genus |> autoplot()
genus |> autoplot(plot_type = "gain")
genus |> autoplot(plot_type = "pr")
```

`math_functions` *Mathematical Functions With Global* `na.rm`

Description

These functions call their original base R namesake, but with a global settable `na.rm` argument.

Usage

```
any(..., na.rm = getOption("na.rm", FALSE))

all(..., na.rm = getOption("na.rm", FALSE))

mean(x, ..., na.rm = getOption("na.rm", FALSE))

sum(..., na.rm = getOption("na.rm", FALSE))

prod(..., na.rm = getOption("na.rm", FALSE))

min(..., na.rm = getOption("na.rm", FALSE))

max(..., na.rm = getOption("na.rm", FALSE))

pmin(..., na.rm = getOption("na.rm", FALSE))

pmax(..., na.rm = getOption("na.rm", FALSE))

range(..., na.rm = getOption("na.rm", FALSE))

sd(x, na.rm = getOption("na.rm", FALSE))

var(x, ..., na.rm = getOption("na.rm", FALSE))

median(x, na.rm = getOption("na.rm", FALSE), ...)

fivenum(x, na.rm = getOption("na.rm", FALSE))

quantile(
  x,
  ...,
  na.rm = getOption("na.rm", FALSE),
  type = getOption("quantile.type", 7)
)

IQR(
  x,
  ...,
  na.rm = getOption("na.rm", FALSE),
  type = getOption("quantile.type", 7)
)
```

Arguments

...	zero or more logical vectors. Other objects of zero length are ignored, and the rest are coerced to logical ignoring any class.
na.rm	logical. If true NA values are removed before the result is computed.
x	an R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.
type	an integer between 1 and 9 selecting one of the nine quantile algorithms detailed below to be used.

Default values of na.rm

This 'certestats' package supports a global default setting for na.rm in many mathematical functions. This can be set with options(na.rm = TRUE) or options(na.rm = FALSE).

For normality(), quantile() and IQR(), this also applies to the type argument. The default, type = 7, is the default of base R. Use type = 6 to comply with SPSS.

Source

base::any()

base::all()

base::mean()

base::sum()

base::prod()

base::min()

base::max()

base::pmin()

base::pmax()

base::range()

stats::sd()

stats::var()

stats::median()

stats::fivenum()

stats::quantile()

stats::IQR()

`moving_average` *Moving Average*

Description

Functions to calculate a moving average. These are useful to get rid of (strong) peakiness.

Usage

```
moving_average(x, w, side = "centre", na.rm = getOption("na.rm", FALSE))

moving_sum(x, w, side = "centre", na.rm = getOption("na.rm", FALSE))

moving_Q1(x, w, side = "centre", na.rm = getOption("na.rm", FALSE))

moving_Q3(x, w, side = "centre", na.rm = getOption("na.rm", FALSE))

moving_fn(x, w, fun, side = "centre", ...)
```

Arguments

`x`	vector of values
`w`	window length; total number of observations to include. This should preferably be an odd number, so that the same number of values to the left and right of `x` are included.
`side`	default is `"centre"`, can also be `"left"` or `"right"`. This can be used to take a moving average (or sum, or ...) of e.g. the last 7 days.
`na.rm`	a logical to indicate whether empty must be removed from `x`
`fun`	function to apply
`...`	arguments passed on to `fun`

Details

Each function can be used over a moving period with `moving_fn()`. For example, for a moving median: `moving_fn(x, 7, fun = median)`. Or a moving maximum: `moving_fn(x, 5, fun = max)`.

The moving average is determined by averaging `floor(w / 2)` values before and after each element of `x` and all elements in between.

Default values of `na.rm`

This 'certestats' package supports a global default setting for `na.rm` in many mathematical functions. This can be set with `options(na.rm = TRUE)` or `options(na.rm = FALSE)`.

For `normality()`, `quantile()` and `IQR()`, this also applies to the `type` argument. The default, `type = 7`, is the default of base R. Use `type = 6` to comply with SPSS.

`normality` *Normality Analysis*

Description

Check normality of a vector of values.

Usage

```
normality(
  x,
  na.rm = getOption("na.rm", FALSE),
  type = getOption("quantile.type", 7)
)
```

Arguments

`x`	vector of values
`na.rm`	Remove empty values
`type`	an integer between 1 and 9 selecting one of the nine quantile algorithms detailed below to be used.

Default values of `na.rm`

This 'certestats' package supports a global default setting for `na.rm` in many mathematical functions. This can be set with `options(na.rm = TRUE)` or `options(na.rm = FALSE)`.

For `normality()`, `quantile()` and `IQR()`, this also applies to the `type` argument. The default, `type = 7`, is the default of base R. Use `type = 6` to comply with SPSS.

Examples

```
x <- runif(1000)
normality(x)

x <- rnorm(1000)
normality(x)

x <- rexp(1000, rate = 3)
normality(x)
```

qc_rules *Quality Control (QC) Rules*

Description

These rules are used for quality control (QC). Default values are set for **Nelson's QC rules**, but they also support Westgard, AIAG, Montgomery and Healthcare QC rules.

Usage

```
qc_rule1(x, m = mean(x), s = sd(x))

qc_rule2(x, m = mean(x), threshold = 9)

qc_rule3(x, threshold = 6)

qc_rule4(x, m = mean(x), threshold = 14, direction_mean = FALSE)

qc_rule5(x, m = mean(x), s = sd(x), threshold = 3)

qc_rule6(x, m = mean(x), s = sd(x), threshold = 5)

qc_rule7(x, m = mean(x), s = sd(x), threshold = 15)

qc_rule8(x, m = mean(x), s = sd(x), threshold = 8)

qc_rule_text(rule, threshold)

qc_test(x, m = mean(x), s = sd(x), guideline = "Nelson")
```

Arguments

x	vector with values
m	mean
s	standard deviation
threshold	minimal number of sequential values before rule is triggered (defaults to Nelson's)
direction_mean	a logical to indicate whether *n* observations in a row must be tested for alternating in direction of the mean
rule	number of the rule
guideline	guideline of QC rules set, must be "Nelson", "Westgard", "AIAG", "Montgomery", or "Healthcare"

Rules list

Rule	Rule explanation:	Nelson	Westgard	AIAG	Montg.	HC
#1	One point is more than 3 standard deviations from the mean	1	1	1	1	1
#2	n (or more) points in a row are on the same side of the mean	9	9	7	8	8
#3	n (or more) points in a row are continually incr. or decr.	6	-	6	6	6
#4	n (or more) points in a row alternate in direction, incr. then decr.	14	-	14	14	-
#5	n - 1 out of n points in a row are >2 sd from the mean	3	3	3	3	3
#6	n - 1 out of n points in a row are >1 sd from the mean	5	5	5	5	-
#7	>=n points in a row are within 1 sd of the mean	15	-	15	15	15
#8	>=n points in a row outside 1 sd of the mean, in both directions	8	-	8	8	-

Montg.: Montgomery, *HC*: Healthcare

Source

Nelson LS. **The Shewhart Control Chart—Tests for Special Causes**. Journal of Quality Technology. Informa UK Limited; 1984 Oct;16(4):237–9. doi:10.1080/00224065.1984.11978921.

Examples

```
x <- qc_test(rnorm(250))
x

# turn into data.frame, e.g. for export
head(as.data.frame(x))

## Not run:

library(plot2)
plot2(x, subtitle = "Workflow 'example123'")


## End(Not run)
```

`regression` *Fast Regression Models*

Description

Functions to do fast regression modelling. The functions return a tibble with statistics. Use `plot()` for an extensive model visualisation.

Usage

```
regression(x, ...)

## Default S3 method:
regression(x, y = NULL, type = "lm", family = stats::gaussian, ...)

## S3 method for class 'data.frame'
regression(x, var1, var2 = NULL, type = "lm", family = stats::gaussian, ...)

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

## S3 method for class 'certestats_reg'
autoplot(object, ...)
```

Arguments

x	vector of values, or a data.frame
...	arguments for lm() or glm()
y	vector of values, optional
type	type of function to use, can be "lm" or "glm"
family	only used for glm()
var1, var2	column to use of x, the var2 argument is optional
object	data to plot

Examples

```
runif(10) |> regression()

data.frame(x = 1:50, y = runif(50)) |>
  regression(x, y)

mrsa_from_blood_years <- c(0, 1, 0, 0, 2, 0, 1, 3, 1, 2, 3, 1, 2)
mrsa_from_blood_years |> plot()

mrsa_from_blood_years |> regression()

mrsa_from_blood_years |> regression() |> plot()
```

---

remove_outliers *Remove Outliers*

Description

This simple function uses boxplot.stats() to determine outliers and removes them from the vector.

Usage

```
remove_outliers(x, coef = 1.5)
```

Arguments

x	a vector of values
coef	a multiple of the IQR that is allowed at maximum to keep values within the accepted range

Examples

```
remove_outliers(c(1,2,1,2,1,2,8))

remove_outliers(c(1,2,1,2,1,2))
```

---

row_function *Apply Function per Row*

Description

This can be used to e.g. add a maximum of certain rows.

Usage

```
row_function(fn, ..., data = NULL)
```

Arguments

fn	function to apply, such as `max()`
...	tidyverse selector helpers, passed on to `select()`
data	data set, will be determined with `pick()` if left blank

Examples

```
if (require("dplyr")) {
  iris |>
    mutate(max = row_function(max, where(is.numeric)),
           sepal_mean = row_function(mean, starts_with("Sepal"))) |>
    head()
}
```

`weighted_mean` *Weighted Mean*

Description

Functions to calculate a weighted mean or any other metric.

Usage

```
weighted_mean(x, w, na.rm = getOption("na.rm", FALSE))

weighted_median(x, w, na.rm = getOption("na.rm", FALSE))

weighted_Q1(x, w, na.rm = getOption("na.rm", FALSE))

weighted_Q3(x, w, na.rm = getOption("na.rm", FALSE))

weighted_fn(x, w, fun, ...)
```

Arguments

`x`	vector of values
`w`	weights, length 1 or length of `x`
`na.rm`	a logical to indicate whether empty must be removed from `x`
`fun`	function to apply
`...`	arguments passed on to `fun`

Default values of `na.rm`

This 'certestats' package supports a global default setting for `na.rm` in many mathematical functions. This can be set with `options(na.rm = TRUE)` or `options(na.rm = FALSE)`.

For `normality()`, `quantile()` and `IQR()`, this also applies to the `type` argument. The default, `type = 7`, is the default of base R. Use `type = 6` to comply with SPSS.

Examples

```
x <- c(1:10)
y <- c(1:10)

mean(x)
weighted_mean(x, y)

# essentially equal to:
mean(rep(x, y))

x <- c(0:1000)
y <- c(0:1000)
```

```
mean(x)
weighted_mean(x, y)
weighted_median(x, y)
weighted_Q1(x, y)
weighted_Q3(x, y)
weighted_fn(x, y, quantile, c(0.01, 0.99))
```

Index