Package ‘cvms’

June 28, 2023

Title Cross-Validation for Model Selection

Version 1.6.0

Description Cross-validate one or multiple regression and classification models and get relevant evaluation metrics in a tidy format. Validate the best model on a test set and compare it to a baseline evaluation. Alternatively, evaluate predictions from an external model. Currently supports regression and classification (binary and multiclass). Described in chp. 5 of Jeyaraman, B. P., Olsen, L. R., & Wambugu M. (2019, ISBN: 9781838550134).

License MIT + file LICENSE

URL https://github.com/ludvigolsen/cvms

BugReports https://github.com/ludvigolsen/cvms/issues

Depends R (>= 3.5)

Imports checkmate (>= 2.0.0),
         data.table (>= 1.12),
         dplyr (>= 0.8.5),
         ggplot2,
         groupdata2 (>= 2.0.2),
         lifecycle,
         lme4 (>= 1.1-23),
         MuMIn (>= 1.43.17),
         parameters (>= 0.15.0),
         plyr,
         pROC (>= 1.16.0),
         purrr,
         rearrr (>= 0.3.0),
         recipes (>= 0.1.13),
         rlang (>= 0.4.7),
         stats,
         stringr,
         tibble (>= 3.0.3),
         tidyr (>= 1.1.2),
         utils

Suggests AUC,
        covr (>= 3.3.1),
        e1071 (>= 1.7-2),
        furrr,
topics documented:

- ggimage (>= 0.3.3)
- ggnewscale (>= 0.4.3)
- knitr
- merDeriv (>= 0.2-4)
- nnet (>= 7.3-12)
- randomForest (>= 4.6-14)
- rmarkdown
- rsvg
- testthat (>= 2.3.2)
- xpectr (>= 0.4.1)

VignetteBuilder  knitr
RdMacros  lifecycle
Encoding  UTF-8
LazyData  true
RoxygenNote  7.2.3

R topics documented:

- baseline .................................................. 3
- baseline_binomial ....................................... 9
- baseline_gaussian ....................................... 12
- baseline_multinomial ................................... 15
- binomial_metrics ....................................... 19
- combine_predictors ..................................... 21
- compatible_formula_terms ............................... 23
- confusion_matrix ...................................... 24
- cross_validate ......................................... 27
- cross_validate_fn ...................................... 32
- cvms .................................................. 42
- evaluate ................................................ 42
- evaluate_residuals .................................... 49
- font .................................................... 51
- gaussian_metrics ...................................... 52
- model_functions ....................................... 54
- most_challenging ...................................... 55
- multiclass_probability_tibble ......................... 60
- multinomial_metrics ................................... 62
- musicians ............................................... 65
- participant.scores .................................... 65
- plot_confusion_matrix .................................. 66
- plot_metric_density .................................... 72
- precomputed.formulas .................................. 73
- predicted.musicians ................................... 74
- predict_functions ..................................... 75
- preprocess_functions ................................... 76
- process_info_binomial .................................. 77
- reconstruct_formulas ................................... 79
- select_definitions ..................................... 80
- select_metrics ......................................... 80
- simplify_formula ...................................... 81
Description

[Maturing]
Create a baseline evaluation of a test set.

In modelling, a baseline is a result that is meaningful to compare the results from our models to. For instance, in classification, we usually want our results to be better than random guessing. E.g. if we have three classes, we can expect an accuracy of 33.33%, as for every observation we have 1/3 chance of guessing the correct class. So our model should achieve a higher accuracy than 33.33% before it is more useful to us than guessing.

While this expected value is often fairly straightforward to find analytically, it only represents what we can expect on average. In reality, it’s possible to get far better results than that by guessing. baseline() (binomial, multinomial) finds the range of likely values by evaluating multiple sets of random predictions and summarizing them with a set of useful descriptors. If random guessing frequently obtains an accuracy of 40%, perhaps our model should have better performance than this, before we declare it better than guessing.

How:
When `family` is binomial: evaluates `n` sets of random predictions against the dependent variable, along with a set of all 0 predictions and a set of all 1 predictions. See also baseline_binomial().

When `family` is multinomial: creates one-vs-all (binomial) baseline evaluations for `n` sets of random predictions against the dependent variable, along with sets of “all class x,y,z,...” predictions. See also baseline_multinomial().

When `family` is gaussian: fits baseline models (y ~ 1) on `n` random subsets of `train_data` and evaluates each model on `test_data`. Also evaluates a model fitted on all rows in `train_data`. See also baseline_gaussian().

Wrapper functions:
Consider using one of the wrappers, as they are simpler to use and understand: baseline_gaussian(), baseline_multinomial(), and baseline_binomial().

Usage

```r
baseline(
  test_data,
  dependent_col,
  family,
  train_data = NULL,
  n = 100,
  metrics = list(),
)```
positive = 2,
cutoff = 0.5,
random_generator_fn = runif,
random_effects = NULL,
min_training_rows = 5,
min_training_rows_left_out = 3,
REML = FALSE,
parallel = FALSE
)

Arguments

test_data data.frame.
dependent_col Name of dependent variable in the supplied test and training sets.
family Name of family. (Character)
Currently supports "gaussian", "binomial" and "multinomial".
train_data data.frame. Only used when `family` is "gaussian".
n Number of random samplings to perform. (Default is 100)
For gaussian: The number of random samplings of `train_data` to fit baseline models on.
For binomial and multinomial: The number of sets of random predictions to evaluate.
metrics list for enabling/disabling metrics.
E.g. list("RMSE" = FALSE) would remove RMSE from the regression results,
and list("Accuracy" = TRUE) would add the regular Accuracy metric to the classification results. Default values (TRUE/FALSE) will be used for the remaining available metrics.
You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why f.i. list("all" = FALSE, "RMSE" = TRUE) would return only the RMSE metric.
The list can be created with gaussian_metrics(), binomial_metrics(), or multinomial_metrics().
Also accepts the string "all".
positive Level from dependent variable to predict. Either as character (preferable) or
level index (1 or 2 - alphabetically).
E.g. if we have the levels "cat" and "dog" and we want "dog" to be the positive class, we can either provide "dog" or 2, as alphabetically, "dog" comes after "cat".
Note: For reproducibility, it's preferable to specify the name directly, as different locales may sort the levels differently.
Used when calculating confusion matrix metrics and creating ROC curves.
N.B. Only affects evaluation metrics, not the returned predictions.
N.B. Binomial only. (Character or Integer)
cutoff Threshold for predicted classes. (Numeric)
N.B. Binomial only
random_generator_fn
Function for generating random numbers when type is "multinomial". The
softmax function is applied to the generated numbers to transform them to probabilities.
The first argument must be the number of random numbers to generate, as no other arguments are supplied.
To test the effect of using different functions, see `multiclass_probability_tibble()`.

N.B. **Multinomial only**

**random_effects**
Random effects structure for the Gaussian baseline model. (Character)
E.g. with "(1|ID)", the model becomes "y ~ 1 + (1|ID)".

N.B. **Gaussian only**

**min_training_rows**
Minimum number of rows in the random subsets of `train_data`

**min_training_rows_left_out**
Minimum number of rows left out of the random subsets of `train_data`
I.e. a subset will maximally have the size:
\[
\text{max_rows_in_subset} = \text{nrow(`train_data`) - `min_training_rows_left_out`}
\]

N.B. **Gaussian only**

**REML**
Whether to use Restricted Maximum Likelihood. (Logical)

N.B. **Gaussian only**

**parallel**
Whether to run the `n` evaluations in parallel. (Logical)
Remember to register a parallel backend first. E.g. with `doParallel::registerDoParallel`.

**Details**

Packages used:

**Models:**
Gaussian: `stats::lm, lme4::lmer`

**Results: Gaussian:**
r2m: `MuMIn::r.squaredGLMM`
r2c: `MuMIn::r.squaredGLMM`
AIC: `stats::AIC`
AICc: `MuMIn::AICc`
BIC: `stats::BIC`

**Binomial and Multinomial:**
ROC and related metrics:
Binomial: `pROC::roc`
Multinomial: `pROC::multiclass.roc`

**Value**
list containing:

1. a tibble with summarized results (called `summarized_metrics`)
2. a tibble with random evaluations (`random_evaluations`)
3. a tibble with the summarized class level results (`summarized_class_level_results`) (**Multinomial only**)
Gaussian Results:

The **Summarized Results** tibble contains:
- Average RMSE, MAE, NRMSE (IQR), RRSE, RAE, RMSLE.
- See the additional metrics (disabled by default) at `?gaussian_metrics`.
- The **Measure** column indicates the statistical descriptor used on the evaluations. The row where `Measure == All_rows` is the evaluation when the baseline model is trained on all rows in `train_data`.
- The **Training Rows** column contains the aggregated number of rows used from `train_data` when fitting the baseline models.

The **Random Evaluations** tibble contains:
- The non-aggregated metrics.
- A nested tibble with the `predictions` and targets.
- A nested tibble with the `coefficients` of the baseline models.
- Number of `training rows` used when fitting the baseline model on the training set.
- A nested `Process` information object with information about the evaluation.
- Name of `dependent` variable.
- Name of `fixed` effect (bias term only).
- `Random` effects structure (if specified).

Binomial Results:

Based on the generated test set predictions, a confusion matrix and ROC curve are used to get the following:
- ROC:
  - AUC, Lower CI, and Upper CI
  - Note, that the ROC curve is only computed when AUC is enabled.
- Confusion Matrix:
  - Balanced Accuracy, Accuracy, F1, Sensitivity, Specificity, Positive Predictive Value, Negative Predictive Value, Kappa, Detection Rate, Detection Prevalence, Prevalence, and MCC (Matthews correlation coefficient).

The **Summarized Results** tibble contains:
- The **Measure** column indicates the statistical descriptor used on the evaluations. The row where `Measure == All_0` is the evaluation when all predictions are 0. The row where `Measure == All_1` is the evaluation when all predictions are 1.
- The **aggregated metrics**.

The **Random Evaluations** tibble contains:
- The non-aggregated metrics.
- A nested tibble with the `predictions` and targets.
- A list of ROC curve objects (if computed).
- A nested tibble with the `confusion matrix`. The `Pos_` columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the "positive" class. I.e. the level you wish to predict.
- A nested `Process` information object with information about the evaluation.
- Name of `dependent` variable.
Multinomial Results:

Based on the generated test set predictions, one-vs-all (binomial) evaluations are performed and aggregated to get the same metrics as in the binomial results (excluding MCC, AUC, Lower CI and Upper CI), with the addition of Overall Accuracy and multiclass MCC in the summarized results. It is possible to enable multiclass AUC as well, which has been disabled by default as it is slow to calculate when there’s a large set of classes.

Note: we also refer to the one-vs-all evaluations as the class level results.

The Summarized Results tibble contains:
Summary of the random evaluations.

How: First, the one-vs-all binomial evaluations are aggregated by repetition, then, these aggregations are summarized. Besides the metrics from the binomial evaluations (see Binomial Results above), it also includes Overall Accuracy and multiclass MCC.

The Measure column indicates the statistical descriptor used on the evaluations. The Mean, Median, SD, IQR, Max, Min, NAs, and INFs measures describe the Random Evaluations tibble, while the CL_Max, CL_Min, CL_NAs, and CL_INFs describe the Class Level results.

The rows where Measure == All_<<class name>> are the evaluations when all the observations are predicted to be in that class.

The Summarized Class Level Results tibble contains:
The (nested) summarized results for each class, with the same metrics and descriptors as the Summarized Results tibble. Use tidyr::unnest on the tibble to inspect the results.

How: The one-vs-all evaluations are summarized by class.

The rows where Measure == All_0 are the evaluations when none of the observations are predicted to be in that class, while the rows where Measure == All_1 are the evaluations when all of the observations are predicted to be in that class.

The Random Evaluations tibble contains:
The repetition results with the same metrics as the Summarized Results tibble.

How: The one-vs-all evaluations are aggregated by repetition. If a metric contains one or more NAs in the one-vs-all evaluations, it will lead to an NA result for that repetition.

Also includes:
A nested tibble with the one-vs-all binomial evaluations (Class Level Results), including nested Confusion Matrices and the Support column, which is a count of how many observations from the class is in the test set.
A nested tibble with the predictions and targets.
A list of ROC curve objects.
A nested tibble with the multiclass confusion matrix.
A nested Process information object with information about the evaluation.
Name of dependent variable.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>
See Also

Other baseline functions: `baseline_binomial()`, `baseline_gaussian()`, `baseline_multinomial()`

Examples

```r
# Attach packages
library(cvms)
library(groupdata2) # partition()
library(dplyr) # %>% arrange()
library(tibble)

# Data is part of cvms
data <- participant.scores

# Set seed for reproducibility
set.seed(1)

# Partition data
partitions <- partition(data, p = 0.7, list_out = TRUE)
train_set <- partitions[[1]]
test_set <- partitions[[2]]

# Create baseline evaluations
# Note: usually n=100 is a good setting
# Gaussian
baseline(
  test_data = test_set, train_data = train_set,
  dependent_col = "score", random_effects = "{(1|session)}",
  n = 2, family = "gaussian"
)

# Binomial
baseline(
  test_data = test_set, dependent_col = "diagnosis",
  n = 2, family = "binomial"
)

# Multinomial
# Create some data with multiple classes
multiclass_data <- tibble(
  "target" = rep(paste0("class_", 1:5), each = 10)
) %>%
dplyr::sample_n(35)
baseline(
  test_data = multiclass_data,
  dependent_col = "target",
  n = 4, family = "multinomial"
)

# Parallelize evaluations
# Attach doParallel and register four cores
# Uncomment:
```
baseline_binomial

Create baseline evaluations for binary classification

Description

[Maturing]
Create a baseline evaluation of a test set.

In modelling, a baseline is a result that is meaningful to compare the results from our models to. For instance, in classification, we usually want our results to be better than random guessing. E.g. if we have three classes, we can expect an accuracy of 33.33%, as for every observation we have 1/3 chance of guessing the correct class. So our model should achieve a higher accuracy than 33.33% before it is more useful to us than guessing.

While this expected value is often fairly straightforward to find analytically, it only represents what we can expect on average. In reality, it’s possible to get far better results than that by guessing. baseline_binomial() finds the range of likely values by evaluating multiple sets of random predictions and summarizing them with a set of useful descriptors. Additionally, it evaluates a set of all 0 predictions and a set of all 1 predictions.

**Usage**

```r
baseline_binomial(
  test_data, 
  dependent_col, 
  n = 100, 
  metrics = list(), 
  positive = 2, 
  cutoff = 0.5, 
  parallel = FALSE
)
```

**Arguments**

- **test_data**: data.frame.
- **dependent_col**: Name of dependent variable in the supplied test and training sets.
- **n**: The number of sets of random predictions to evaluate. (Default is 100)
- **metrics**: list for enabling/disabling metrics.
  E.g. `list("F1" = FALSE)` would remove F1 from the results, and `list("Accuracy" = TRUE)` would add the regular Accuracy metric to the results. Default values (TRUE/FALSE) will be used for the remaining available metrics.
  You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why f.i. `list("all" = FALSE, "Accuracy" = TRUE)` would return only the Accuracy metric.
  The list can be created with `binomial_metrics()`.
  Also accepts the string "all".
- **positive**: Level from dependent variable to predict. Either as character (preferable) or level index (1 or 2 - alphabetically).
  E.g. if we have the levels "cat" and "dog" and we want "dog" to be the positive class, we can either provide "dog" or 2, as alphabetically, "dog" comes after "cat".
  **Note:** For reproducibility, it’s preferable to specify the name directly, as different locales may sort the levels differently.
  Used when calculating confusion matrix metrics and creating ROC curves.
  N.B. Only affects evaluation metrics, not the returned predictions.
- **cutoff**: Threshold for predicted classes. (Numeric)
- **parallel**: Whether to run the `n` evaluations in parallel. (Logical)
  Remember to register a parallel backend first. E.g. with `doParallel::registerDoParallel`. 

**Example**

```r
baseline_binomial(
  test_data, 
  dependent_col, 
  n = 100, 
  metrics = list("F1" = FALSE), 
  positive = 2, 
  cutoff = 0.5, 
  parallel = FALSE
)
```
Details

Packages used:
ROC and AUC: pROC::roc

Value

list containing:

1. a tibble with summarized results (called summarized_metrics)
2. a tibble with random evaluations (random_evaluations)

Based on the generated test set predictions, a confusion matrix and ROC curve are used to get the following:
ROC:
AUC, Lower CI, and Upper CI
Note, that the ROC curve is only computed when AUC is enabled.
Confusion Matrix:
Balanced Accuracy, Accuracy, F1, Sensitivity, Specificity, Positive Predictive Value, Negative Predictive Value, Kappa, Detection Rate, Detection Prevalence, Prevalence, and MCC (Matthews correlation coefficient).

The Summarized Results tibble contains:
The Measure column indicates the statistical descriptor used on the evaluations. The row where Measure == All_0 is the evaluation when all predictions are 0. The row where Measure == All_1 is the evaluation when all predictions are 1.
The aggregated metrics.

The Random Evaluations tibble contains:
The non-aggregated metrics.
A nested tibble with the predictions and targets.
A list of ROC curve objects (if computed).
A nested tibble with the confusion matrix. The Pos_ columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the "positive" class. i.e. the level you wish to predict.
A nested Process information object with information about the evaluation.
Name of dependent variable.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other baseline functions: baseline_gaussian(), baseline_multinomial(), baseline()
Examples

```r
# Attach packages
library(cvms)
library(groupdata2) # partition()
library(dplyr) # %>% arrange()

# Data is part of cvms
data <- participant.scores

# Set seed for reproducibility
set.seed(1)

# Partition data
partitions <- partition(data, p = 0.7, list_out = TRUE)
train_set <- partitions[[1]]
test_set <- partitions[[2]]

# Create baseline evaluations
# Note: usually n=100 is a good setting
baseline_binomial(
    test_data = test_set,
    dependent_col = "diagnosis",
    n = 2
)

# Parallelize evaluations
# Attach doParallel and register four cores
# Uncomment:
# library(doParallel)
# registerDoParallel(4)

# Make sure to uncomment the parallel argument
baseline_binomial(
    test_data = test_set,
    dependent_col = "diagnosis",
    n = 4
    #, parallel = TRUE  # Uncomment
)
```

---

**baseline_gaussian**

Create baseline evaluations for regression models

**Description**

[Maturing]

Create a baseline evaluation of a test set.

In modelling, a baseline is a result that is meaningful to compare the results from our models to. In regression, we want our model to be better than a model without any predictors. If our model does not perform better than such a simple model, it’s unlikely to be useful.
baseline_gaussian() fits the intercept-only model (\(y \sim 1\)) on `n` random subsets of `train_data` and evaluates each model on `test_data`. Additionally, it evaluates a model fitted on all rows in `train_data`.

Usage

```r
code
baseline_gaussian(
  test_data,
  train_data,
  dependent_col,
  n = 100,
  metrics = list(),
  random_effects = NULL,
  min_training_rows = 5,
  min_training_rows_left_out = 3,
  REML = FALSE,
  parallel = FALSE
)
```

Arguments

test_data data.frame.

train_data data.frame.

dependent_col Name of dependent variable in the supplied test and training sets.

n The number of random samplings of `train_data` to fit baseline models on. (Default is 100)

metrics list for enabling/disabling metrics.

E.g. `list("RMSE" = FALSE)` would remove RMSE from the results, and `list("TAE" = TRUE)` would add the Total Absolute Error metric to the results. Default values (TRUE/FALSE) will be used for the remaining available metrics.

You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why f.i. `list("all" = FALSE, "RMSE" = TRUE)` would return only the RMSE metric.

The list can be created with `gaussian_metrics()`.

Also accepts the string "all".

random_effects Random effects structure for the baseline model. (Character)

E.g. with "(1|ID)", the model becomes "y ~ 1 + (1|ID)".

min_training_rows Minimum number of rows in the random subsets of `train_data`.

min_training_rows_left_out Minimum number of rows left out of the random subsets of `train_data`.

I.e. a subset will maximally have the size: `max_rows_in_subset = nrow('train_data') - min_training_rows_left_out`.

REML Whether to use Restricted Maximum Likelihood. (Logical)

parallel Whether to run the `n` evaluations in parallel. (Logical)

Remember to register a parallel backend first. E.g. with `doParallel::registerDoParallel`.  

Details

Packages used:

Models:
   `stats::lm`, `lme4::lmer`

Results:
   r2m: `MuMIn::r.squaredGLMM`
   r2c: `MuMIn::r.squaredGLMM`
   AIC: `stats::AIC`
   AICc: `MuMIn::AICc`
   BIC: `stats::BIC`

Value

list containing:

1. a tibble with summarized results (called `summarized_metrics`)
2. a tibble with random evaluations (random_evaluations)

The Summarized Results tibble contains:

Average RMSE, MAE, NRMSE(IQR), RRSE, RAE, RMSLE.
See the additional metrics (disabled by default) at `gaussian_metrics`.

The Measure column indicates the statistical descriptor used on the evaluations. The row where Measure == `All_rows` is the evaluation when the baseline model is trained on all rows in `train_data`

The Training Rows column contains the aggregated number of rows used from `train_data`, when fitting the baseline models.

The Random Evaluations tibble contains:

The non-aggregated metrics.
A nested tibble with the predictions and targets.
A nested tibble with the coefficients of the baseline models.
Number of training rows used when fitting the baseline model on the training set.
A nested Process information object with information about the evaluation.
Name of dependent variable.
Name of fixed effect (bias term only).

Random effects structure (if specified).

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other baseline functions: `baseline_binomial()`, `baseline_multinomial()`, `baseline()`
Examples

# Attach packages
library(cvms)
library(groupdata2) # partition()
library(dplyr) # %>% arrange()

# Data is part of cvms
data <- participant.scores

# Set seed for reproducibility
set.seed(1)

# Partition data
partitions <- partition(data, p = 0.7, list_out = TRUE)
train_set <- partitions[[1]]
test_set <- partitions[[2]]

# Create baseline evaluations
# Note: usually n=100 is a good setting
baseline_gaussian(
    test_data = test_set,
    train_data = train_set,
    dependent_col = "score",
    random_effects = "(1|session)",
    n = 2
)

# Parallelize evaluations

# Attach doParallel and register four cores
# Uncomment:
# library(doParallel)
# registerDoParallel(4)

# Make sure to uncomment the parallel argument
baseline_gaussian(
    test_data = test_set,
    train_data = train_set,
    dependent_col = "score",
    random_effects = "(1|session)",
    n = 4
    #, parallel = TRUE  # Uncomment
)

Description

[Maturing]
Create a baseline evaluation of a test set.
In modelling, a *baseline* is a result that is meaningful to compare the results from our models to. For instance, in classification, we usually want our results to be better than random guessing. E.g. if we have three classes, we can expect an accuracy of 33.33%, as for every observation we have 1/3 chance of guessing the correct class. So our model should achieve a higher accuracy than 33.33% before it is more useful to us than guessing.

While this expected value is often fairly straightforward to find analytically, it only represents what we can expect on average. In reality, it’s possible to get far better results than that by guessing. `baseline_multinomial()` finds the range of likely values by evaluating multiple sets of random predictions and summarizing them with a set of useful descriptors.

Technically, it creates *one-vs-all* (binomial) baseline evaluations for the `n` sets of random predictions and summarizes them. Additionally, sets of "all class x,y,z,..." predictions are evaluated.

**Usage**

```r
baseline_multinomial(
  test_data,  # data.frame.
  dependent_col,  # Name of dependent variable in the supplied test and training sets.
  n = 100,  # The number of sets of random predictions to evaluate. (Default is 100)
  metrics = list(),  # list for enabling/disabling metrics.
  random_generator_fn = runif,  # Function for generating random numbers. The softmax function is applied to
                                 # the generated numbers to transform them to probabilities.
  parallel = FALSE  # Whether to run the `n` evaluations in parallel. (Logical)
)
```

**Arguments**

- `test_data` *data.frame.*
- `dependent_col` Name of dependent variable in the supplied test and training sets.
- `n` The number of sets of random predictions to evaluate. (Default is 100)
- `metrics` list for enabling/disabling metrics. E.g. `list("F1" = FALSE)` would remove F1 from the results, and `list("Accuracy" = TRUE)` would add the regular Accuracy metric to the results. Default values (TRUE/FALSE) will be used for the remaining available metrics.
  
  You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why f.i. `list("all" = FALSE, "Accuracy" = TRUE)` would return only the Accuracy metric.
  
  The list can be created with `multinomial_metrics()`. Also accepts the string "all".
- `random_generator_fn` Function for generating random numbers. The softmax function is applied to the generated numbers to transform them to probabilities.
  
  The first argument must be the number of random numbers to generate, as no other arguments are supplied.
  
  To test the effect of using different functions, see `multiclass_probability_tibble()`.
- `parallel` Whether to run the "n" evaluations in parallel. (Logical)

**Details**

Packages used:

- Multiclass ROC curve and AUC: `pROC::multiclass.roc`
**Value**

list containing:

1. a tibble with summarized results (called `summarized_metrics`)
2. a tibble with random evaluations (random_evaluations)
3. a tibble with the summarized class level results (summarized_class_level_results)

---------------------------------------------

**Macro metrics:**
Based on the generated predictions, one-vs-all (binomial) evaluations are performed and aggregated to get the following macro metrics:
Balanced Accuracy, F1, Sensitivity, Specificity, Positive Predictive Value, Negative Predictive Value, Kappa, Detection Rate, Detection Prevalence, and Prevalence.
In general, the metrics mentioned in `binomial_metrics()` can be enabled as macro metrics (excluding MCC, AUC, Lower CI, Upper CI, and the AIC/AICc/BIC metrics). These metrics also has a weighted average version.

**N.B.** we also refer to the one-vs-all evaluations as the class level results.

**Multiclass metrics:**
In addition, the Overall Accuracy and multiclass MCC metrics are computed. Multiclass AUC can be enabled but is slow to calculate with many classes.

---------------------------------------------

The **Summarized Results** tibble contains:
Summary of the random evaluations.

**How:** The one-vs-all binomial evaluations are aggregated by repetition and summarized. Besides the metrics from the binomial evaluations, it also includes Overall Accuracy and multiclass MCC.
The Measure column indicates the statistical descriptor used on the evaluations. The Mean, Median, SD, IQR, Max, Min, NAs, and INFs measures describe the Random Evaluations tibble, while the CL_Max, CL_Min, CL_NAs, and CL_INFs describe the Class Level results.
The rows where Measure == All_<<class name>> are the evaluations when all the observations are predicted to be in that class.

---------------------------------------------

The **Summarized Class Level Results** tibble contains:
The (nested) summarized results for each class, with the same metrics and descriptors as the Summarized Results tibble. Use `tidyr::unnest` on the tibble to inspect the results.

**How:** The one-vs-all evaluations are summarized by class.
The rows where Measure == All_0 are the evaluations when none of the observations are predicted to be in that class, while the rows where Measure == All_1 are the evaluations when all of the observations are predicted to be in that class.

---------------------------------------------

The **Random Evaluations** tibble contains:
The repetition results with the same metrics as the Summarized Results tibble.

**How:** The one-vs-all evaluations are aggregated by repetition. If a metric contains one or more NAs in the one-vs-all evaluations, it will lead to an NA result for that repetition.
Also includes:
A nested tibble with the one-vs-all binomial evaluations (Class Level Results), including nested Confusion Matrices and the Support column, which is a count of how many observations from the class is in the test set.

A nested tibble with the predictions and targets.

A list of ROC curve objects.

A nested tibble with the multiclass confusion matrix.

A nested Process information object with information about the evaluation.

Name of dependent variable.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other baseline functions: baseline_binomial(), baseline_gaussian(), baseline()

Examples

```r
# Attach packages
library(cvms)
library(groupdata2) # partition()
library(dplyr) # %>% arrange()
library(tibble)

# Data is part of cvms
data <- participant.scores

# Set seed for reproducibility
set.seed(1)

# Partition data
partitions <- partition(data, p = 0.7, list_out = TRUE)
train_set <- partitions[[1]]
test_set <- partitions[[2]]

# Create baseline evaluations
# Note: usually n=100 is a good setting

# Create some data with multiple classes
multiclass_data <- tibble(
  "target" = rep(paste0("class_", 1:5), each = 10)
) %>%
dplyr::sample_n(35)

baseline_multinomial(
  test_data = multiclass_data,
  dependent_col = "target",
  n = 4
)

# Parallelize evaluations

# Attach doParallel and register four cores
# Uncomment:
```
```r
# library(doParallel)
# registerDoParallel(4)

# Make sure to uncomment the parallel argument
(mb <- baseline_multinomial(
    test_data = multiclass_data,
    dependent_col = "target",
    n = 6
    #, parallel = TRUE  # Uncomment
))

# Inspect the summarized class level results
# for class_2
mb$summarized_class_level_results %>%
dplyr::filter(Class == "class_2") %>%
tidy::unnest(Results)

# Multinomial with custom random generator function
# that creates very "certain" predictions
# (once softmax is applied)
rcertain <- function(n) {
    (runif(n, min = 1, max = 100)^1.4) / 100
}

# Make sure to uncomment the parallel argument
baseline_multinomial(
    test_data = multiclass_data,
    dependent_col = "target",
    n = 6,
    random_generator_fn = rcertain
    #, parallel = TRUE  # Uncomment
)
```

---

### binomial_metrics

Select metrics for binomial evaluation

#### Description

**[Experimental]**

Enable/disable metrics for binomial evaluation. Can be supplied to the `metrics` argument in many of the `cvms` functions.

Note: Some functions may have slightly different defaults than the ones supplied here.

#### Usage

```r
binomial_metrics(  
    all = NULL,  
    balanced_accuracy = NULL,  
    accuracy = NULL,  
    f1 = NULL,  
    sensitivity = NULL,  
)```
specificity = NULL,
pos_pred_value = NULL,
neg_pred_value = NULL,
auc = NULL,
lower_ci = NULL,
upper_ci = NULL,
kappa = NULL,
mcc = NULL,
detection_rate = NULL,
detection_prevalence = NULL,
prevalence = NULL,
false_neg_rate = NULL,
false_pos_rate = NULL,
false_discovery_rate = NULL,
false_omission_rate = NULL,
threat_score = NULL,
aic = NULL,
aicc = NULL,
bic = NULL
)

Arguments

all        Enable/disable all arguments at once. (Logical)

Specifying other metrics will overwrite this, why you can use (all = FALSE, accuracy = TRUE) to get only the Accuracy metric.

balanced_accuracy  Balanced Accuracy (Default: TRUE)
accuracy           Accuracy (Default: FALSE)
f1                 F1 (Default: TRUE)
sensitivity        Sensitivity (Default: TRUE)
specificity         Specificity (Default: TRUE)
pos_pred_value     Pos Pred Value (Default: TRUE)
neg_pred_value     Neg Pred Value (Default: TRUE)
auc                AUC (Default: TRUE)
lower_ci           Lower CI (Default: TRUE)
upper_ci           Upper CI (Default: TRUE)
kappa             Kappa (Default: TRUE)
mcc                MCC (Default: TRUE)
detection_rate     Detection Rate (Default: TRUE)
detection_prevalence Detection Prevalence (Default: TRUE)
prevalence         Prevalence (Default: TRUE)
false_neg_rate     False Neg Rate (Default: FALSE)
false_pos_rate     False Pos Rate (Default: FALSE)
false_discovery_rate False Discovery Rate (Default: FALSE)
**combine_predictors**

Generate model formulas by combining predictors

**false omission rate**
False Omission Rate (Default: FALSE)

**threat score**
Threat Score (Default: FALSE)

**aic**
AIC. (Default: FALSE)

**aicc**
AICc. (Default: FALSE)

**bic**
BIC. (Default: FALSE)

**Author(s)**
Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

**See Also**
Other evaluation functions: `confusion_matrix()` , `evaluate_residuals()` , `evaluate()` , `gaussian_metrics()` , `multinomial_metrics()`

**Examples**

```r
# Attach packages
library(cvms)

# Enable only Balanced Accuracy
binomial_metrics(all = FALSE, balanced_accuracy = TRUE)

# Enable all but Balanced Accuracy
binomial_metrics(all = TRUE, balanced_accuracy = FALSE)

# Disable Balanced Accuracy
binomial_metrics(balanced_accuracy = FALSE)
```

**Description**

[Maturing]
Create model formulas with every combination of your fixed effects, along with the dependent variable and random effects. 259,358 formulas have been precomputed with two- and three-way interactions for up to 8 fixed effects, with up to 5 included effects per formula. Uses the + and * operators, so lower order interactions are automatically included.

**Usage**

```r
combine_predictors(
  dependent,
  fixed_effects,
  random_effects = NULL,
  max_fixed_effects = 5,
  max_interaction_size = 3,
  max_effect_frequency = NULL
)
```
**combine_predictors**

**Arguments**

- **dependent**
  Name of dependent variable. (Character)

- **fixed_effects**
  List of fixed effects. (Character)
  Max. limit of 8 effects **when interactions are included!**
  A fixed effect name cannot contain: white spaces, "," or "+".
  Effects in sublists will be interchanged. This can be useful, when we have multiple versions of a predictor (e.g. x1 and log(x1)) that we do not wish to have in the same formula.
  Example of interchangeable effects:
  ```
  list(list("x1", "log_x1"), "x2", "x3")
  ```

- **random_effects**
  The random effects structure. (Character)
  Is appended to the model formulas.

- **max_fixed_effects**
  Maximum number of fixed effects in a model formula. (Integer)
  Max. limit of 5 **when interactions are included!**

- **max_interaction_size**
  Maximum number of effects in an interaction. (Integer)
  Max. limit of 3.
  Use this to limit the n-way interactions allowed. 0 or 1 excludes interactions all together.
  A model formula can contain multiple interactions.

- **max_effect_frequency**
  Maximum number of times an effect is included in a formula string.

**Value**

List of model formulas.

E.g.:
```
c("y ~ x1 + (1|z)", "y ~ x2 + (1|z)", "y ~ x1 + x2 + (1|z)", "y ~ x1 * x2 + (1|z)").
```

**Author(s)**

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

**Examples**

```r
# Attach packages
library(cvms)

# Create effect names
dependent <- "y"
fixed_effects <- c("a", "b", "c")
random_effects <- "(1|e)"

# Create model formulas
combine_predictors(
    dependent, fixed_effects, random_effects)
```

# Create effect names with interchangeable effects in sublists
fixed_effects <- list("a", list("b", "log_b"), "c")

# Create model formulas
combine_predictors(
  dependent, fixed_effects,
  random_effects
)

## compatible.formula.terms

Compatible formula terms

### Description
162,660 pairs of compatible terms for building model formulas with up to 15 fixed effects.

### Format
A data.frame with 162,660 rows and 5 variables:

- **left** term, fixed effect or interaction, with fixed effects separated by "*"
- **right** term, fixed effect or interaction, with fixed effects separated by "*"
- **max_interaction_size** maximum interaction size in the two terms, up to 3
- **num_effects** number of unique fixed effects in the two terms, up to 5
- **min_num_fixed_effects** minimum number of fixed effects required to use a formula with the two terms, i.e. the index in the alphabet of the last of the alphabetically ordered effects (letters) in the two terms, so 4 if left == "A" and right == "D"

### Details
A term is either a fixed effect or an interaction between fixed effects (up to three-way), where the effects are separated by the "*" operator.

Two terms are compatible if they are not redundant, meaning that both add a fixed effect to the formula. E.g. as the interaction "x1 * x2 * x3" expands to "x1 + x2 + x3 + x1 * x2 + x1 * x3 + x2 * x3 + x1 * x2 * x3", the higher order interaction makes these "sub terms" redundant. Note: All terms are compatible with NA.

Effects are represented by the first fifteen capital letters.

Used to generate the model formulas for combine_predictors.

### Author(s)
Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>
confusion_matrix

Create a confusion matrix

Description

Create a confusion matrix from targets and predictions. Calculates associated metrics.

Multiclass results are based on one-vs-all evaluations. Both regular averaging and weighted averaging are available. Also calculates the Overall Accuracy.

Note: In most cases you should use evaluate() instead. It has additional metrics and works in magrittr pipes (e.g. %>%) and with dplyr::group_by(). confusion_matrix() is more lightweight and may be preferred in programming when you don’t need the extra stuff in evaluate().

Usage

confusion_matrix(
  targets, predictions,
  metrics = list(),
  positive = 2,
  c_levels = NULL,
  do_one_vs_all = TRUE,
  parallel = FALSE
)

Arguments

targets vector with true classes. Either numeric or character.
predictions vector with predicted classes. Either numeric or character.
metrics list for enabling/disabling metrics.
E.g. list("Accuracy" = TRUE) would add the regular accuracy metric, while list("F1" = FALSE) would remove the F1 metric. Default values (TRUE/FALSE) will be used for the remaining available metrics.
You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why for instance list("all" = FALSE, "Accuracy" = TRUE) would return only the Accuracy metric.
The list can be created with binomial_metrics() or multinomial_metrics(). Also accepts the string "all".
positive Level from `targets` to predict. Either as character (preferable) or level index (1 or 2 - alphabetically). (Two-class only)
E.g. if we have the levels "cat" and "dog" and we want "dog" to be the positive class, we can either provide "dog" or 2, as alphabetically, "dog" comes after "cat".
Note: For reproducibility, it’s preferable to specify the name directly, as different locales may sort the levels differently.
c_levels  vector with categorical levels in the targets. Should have same type as `targets`. If NULL, they are inferred from `targets`. N.B. the levels are sorted alphabetically. When `positive` is numeric (i.e. an index), it therefore still refers to the index of the alphabetically sorted levels.

do_one_vs_all  Whether to perform one-vs-all evaluations when working with more than 2 classes (multiclass). If you are only interested in the confusion matrix, this allows you to skip most of the metric calculations.

parallel  Whether to perform the one-vs-all evaluations in parallel. (Logical) N.B. This only makes sense when you have a lot of classes or a very large dataset. Remember to register a parallel backend first. E.g. with doParallel::registerDoParallel.

Details

The following formulas are used for calculating the metrics:

\[
\text{Sensitivity} = \frac{TP}{TP + FN}
\]

\[
\text{Specificity} = \frac{TN}{TN + FP}
\]

\[
\text{Pos Pred Value} = \frac{TP}{TP + FP}
\]

\[
\text{Neg Pred Value} = \frac{TN}{TN + FN}
\]

\[
\text{Balanced Accuracy} = \frac{(\text{Sensitivity} + \text{Specificity})}{2}
\]

\[
\text{Accuracy} = \frac{(TP + TN)}{(TP + TN + FP + FN)}
\]

\[
\text{Overall Accuracy} = \frac{\text{Correct}}{(\text{Correct} + \text{Incorrect})}
\]

\[
F1 = 2 \times \frac{\text{Pos Pred Value} \times \text{Sensitivity}}{(\text{Pos Pred Value} + \text{Sensitivity})}
\]

\[
\text{MCC} = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}}
\]

Note for MCC: Formula is for the binary case. When the denominator is 0, we set it to 1 to avoid NaN. See the metrics vignette for the multiclass version.

\[
\text{Detection Rate} = \frac{TP}{(TP + FN + TN + FP)}
\]

\[
\text{Detection Prevalence} = \frac{(TP + FP)}{(TP + FN + TN + FP)}
\]

\[
\text{Threat Score} = \frac{TP}{(TP + FN + FP)}
\]

\[
\text{False Neg Rate} = 1 - \text{Sensitivity}
\]

\[
\text{False Pos Rate} = 1 - \text{Specificity}
\]

\[
\text{False Discovery Rate} = 1 - \text{Pos Pred Value}
\]

\[
\text{False Omission Rate} = 1 - \text{Neg Pred Value}
\]

For Kappa the counts (TP, TN, FP, FN) are normalized to percentages (summing to 1). Then the following is calculated:

\[
p_{\text{observed}} = TP + TN
\]

\[
p_{\text{expected}} = (TN + FP) \times (TN + FN) + (FN + TP) \times (FP + TP)
\]

\[
\text{Kappa} = \frac{(p_{\text{observed}} - p_{\text{expected}})}{(1 - p_{\text{expected}})}
\]
Value

tibble with:

- Nested confusion matrix (tidied version)
- Nested confusion matrix (table)

The Positive Class.

Multiclass only: Nested Class Level Results with the two-class metrics, the nested confusion matrices, and the Support metric, which is a count of the class in the target column and is used for the weighted average metrics.

The following metrics are available (see `metrics`):

Two classes or more:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balanced Accuracy</td>
<td>&quot;Balanced Accuracy&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Accuracy</td>
<td>&quot;Accuracy&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>F1</td>
<td>&quot;F1&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>&quot;Sensitivity&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Specificity</td>
<td>&quot;Specificity&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Positive Predictive Value</td>
<td>&quot;Pos Pred Value&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Negative Predictive Value</td>
<td>&quot;Neg Pred Value&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Kappa</td>
<td>&quot;Kappa&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Matthews Correlation Coefficient</td>
<td>&quot;MCC&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Detection Rate</td>
<td>&quot;Detection Rate&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Detection Prevalence</td>
<td>&quot;Detection Prevalence&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Prevalence</td>
<td>&quot;Prevalence&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>False Negative Rate</td>
<td>&quot;False Neg Rate&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>False Positive Rate</td>
<td>&quot;False Pos Rate&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>False Discovery Rate</td>
<td>&quot;False Discovery Rate&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>False Omission Rate</td>
<td>&quot;False Omission Rate&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>Threat Score</td>
<td>&quot;Threat Score&quot;</td>
<td>Disabled</td>
</tr>
</tbody>
</table>

The Name column refers to the name used in the package. This is the name in the output and when enabling/disabling in `metrics`.

Three classes or more:

The metrics mentioned above (excluding MCC) has a weighted average version (disabled by default; weighted by the Support).

In order to enable a weighted metric, prefix the metric name with "Weighted " when specifying `metrics`.

E.g. `metrics = list("Weighted Accuracy" = TRUE)`.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall Accuracy</td>
<td>&quot;Overall Accuracy&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Weighted *</td>
<td>&quot;Weighted *&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>Multiclass MCC</td>
<td>&quot;MCC&quot;</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>
cross_validate

Cross-validate regression models for model selection

tibble

Description

[Stable]

Cross-validate one or multiple linear or logistic regression models at once. Perform repeated cross-validation. Returns results in a tibble for easy comparison, reporting and further analysis.

See cross_validate_fn() for use with custom model functions.

See Also

Other evaluation functions: binomial_metrics(), evaluate_residuals(), evaluate(), gaussian_metrics(), multinomial_metrics()
Usage

cross_validate(
  data,
  formulas,
  family,
  fold_cols = ".folds",
  control = NULL,
  REML = FALSE,
  cutoff = 0.5,
  positive = 2,
  metrics = list(),
  preprocessing = NULL,
  rm_nc = FALSE,
  parallel = FALSE,
  verbose = FALSE,
  link = deprecated(),
  models = deprecated(),
  model_verbose = deprecated()
)

Arguments

data data.frame. Must include one or more grouping factors for identifying folds - as made with groupdata2::fold().
formulas Model formulas as strings. (Character) E.g. c("y~x", "y~z"). Can contain random effects. E.g. c("y~x+(1|r)", "y~z+(1|r)").
family Name of the family. (Character) Currently supports "gaussian" for linear regression with lm() / lme4::lmer() and "binomial" for binary classification with glm() / lme4::glmer(). See cross_validate_fn() for use with other model functions.
fold_cols Name(s) of grouping factor(s) for identifying folds. (Character) Include names of multiple grouping factors for repeated cross-validation.
control Construct control structures for mixed model fitting (with lme4::lmer() or lme4::glmer()). See lme4::lmerControl and lme4::glmerControl. N.B. Ignored if fitting lm() or glm() models.
REML Restricted Maximum Likelihood. (Logical)
cutoff Threshold for predicted classes. (Numeric) N.B. Binomial models only
positive Level from dependent variable to predict. Either as character (preferable) or level index (1 or 2 - alphabetically). E.g. if we have the levels "cat" and "dog" and we want "dog" to be the positive class, we can either provide "dog" or 2, as alphabetically, "dog" comes after "cat".

Note: For reproducibility, it’s preferable to specify the name directly, as different locales may sort the levels differently.
**cross_validate**

Used when calculating confusion matrix metrics and creating ROC curves.
The **Process** column in the output can be used to verify this setting.
N.B. Only affects evaluation metrics, not the model training or returned predictions.

### N.B. Binomial models only.

**metrics**

List for enabling/disabling metrics.
E.g. `list("RMSE" = FALSE)` would remove RMSE from the results, and `list("Accuracy" = TRUE)` would add the regular Accuracy metric to the classification results. Default values (TRUE/FALSE) will be used for the remaining available metrics.
You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why `list("all" = FALSE, "RMSE" = TRUE)` would return only the RMSE metric.
The list can be created with `gaussian_metrics()` or `binomial_metrics()`.
Also accepts the string "all".

**preprocessing**

Name of preprocessing to apply.
Available preprocessings are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;standardize&quot;</td>
<td>Centers and scales the numeric predictors.</td>
</tr>
<tr>
<td>&quot;range&quot;</td>
<td>Normalizes the numeric predictors to the 0-1 range. Values outside the min/max range in the test fold are truncated to 0/1.</td>
</tr>
<tr>
<td>&quot;scale&quot;</td>
<td>Scales the numeric predictors to have a standard deviation of one.</td>
</tr>
<tr>
<td>&quot;center&quot;</td>
<td>Centers the numeric predictors to have a mean of zero.</td>
</tr>
</tbody>
</table>

The preprocessing parameters (mean, SD, etc.) are extracted from the training folds and applied to both the training folds and the test fold. They are returned in the **Preprocess** column for inspection.
N.B. The preprocessings should not affect the results to a noticeable degree, although "range" might due to the truncation.

**rm_nc**

Remove non-converged models from output. (Logical)

**parallel**

Whether to cross-validate the list of models in parallel. (Logical)
Remember to register a parallel backend first. E.g. with `doParallel::registerDoParallel()`.

**verbose**

Whether to message process information like the number of model instances to fit and which model function was applied. (Logical)

**link, models, model_verbose**

Deprecated.

### Details

Packages used:

**Models:**

Gaussian: `stats::lm, lme4::lmer`
Binomial: `stats::glm, lme4::glmer`

**Results:**

*Shared:*

AIC: `stats::AIC`
AICC: `MuMin::AICC`
BIC: `stats::BIC`
**Gaussian:**

- `r2m`: `MuMIn::r.squaredGLMM`
- `r2c`: `MuMIn::r.squaredGLMM`

**Binomial:**

- ROC and AUC: `pROC::roc`

**Value**

tibble with results for each model.

- **Shared across families:** A nested tibble with **coefficients** of the models from all iterations.
- Number of **total folds**.
- Number of **fold columns**.
- Count of **convergence warnings**. Consider discarding models that did not converge on all iterations. Note: you might still see results, but these should be taken with a grain of salt!
- Count of **other warnings**. These are warnings without keywords such as "convergence".
- Count of **Singular Fit messages**. See `lme4::isSingular` for more information.
- Nested tibble with the **warnings and messages** caught for each model.
- A nested **Process** information object with information about the evaluation.
- Name of **dependent** variable.
- Names of **fixed** effects.
- Names of **random** effects, if any.
- Nested tibble with **preprocessing** parameters, if any.

```
Gaussian Results:
```

Average RMSE, MAE, NRMSE(IQR), RRSE, RAE, RMSLE, AIC, AICc, and BIC of all the iterations*, **omitting potential NAs from non-converged iterations**. Note that the Information Criterion metrics (AIC, AICc, and BIC) are also averages.

- See the additional metrics (disabled by default) at `?gaussian_metrics`.
- A nested tibble with the **predictions** and targets.
- A nested tibble with the non-averaged **results** from all iterations.
- * In **repeated cross-validation**, the metrics are first averaged for each fold column (repetition) and then averaged again.

```
Binomial Results:
```

Based on the **collected** predictions from the test folds*, a confusion matrix and a ROC curve are created to get the following:

- ROC:
  - AUC, Lower CI, and Upper CI
- Confusion Matrix:
  - Balanced Accuracy, F1, Sensitivity, Specificity, Positive Predictive Value, Negative Predictive Value, Kappa, Detection Rate, Detection Prevalence, Prevalence, and MCC (Matthews correlation coefficient).

- See the additional metrics (disabled by default) at `?binomial_metrics`.

* In **repeated cross-validation**, the metrics are first averaged for each fold column (repetition) and then averaged again.
Also includes:
A nested tibble with predictions, predicted classes (depends on cutoff), and the targets. Note, that the predictions are not necessarily of the specified positive class, but of the model’s positive class (second level of dependent variable, alphabetically).
The pROC::roc ROC curve object(s).
A nested tibble with the confusion matrix/matrices. The Pos_columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the “positive” class. I.e. the level you wish to predict.
A nested tibble with the results from all fold columns.
The name of the Positive Class.
* In repeated cross-validation, an evaluation is made per fold column (repetition) and averaged.

Author(s)
Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>
Benjamin Hugh Zachariae

See Also
Other validation functions: cross_validate_fn(), validate_fn(), validate()

Examples
# Attach packages
library(cvms)
library(groupdata2) # fold()
library(dplyr) # %>% arrange()

# Data is part of cvms
data <- participant.scores

# Set seed for reproducibility
set.seed(7)

# Fold data
data <- fold(
data, 
  k = 4, 
  cat_col = "diagnosis", 
  id_col = "participant"
) %>%
    arrange(.folds)

# Cross-validate a single model

cross_validate(
data, 
  formulas = "score~diagnosis", 
  family = "gaussian", 
  REML = FALSE
)
# Binomial
cross_validate(
  data,
  formulas = "diagnosis~score",
  family = "binomial"
)

# Cross-validate multiple models
#
formulas <- c(
  "score~diagnosis+(1|session)",
  "score~age+(1|session)"
)
cross_validate(
  data,
  formulas = formulas,
  family = "gaussian",
  REML = FALSE
)

# Use parallelization
#
# Attach doParallel and register four cores
# Uncomment:
# library(doParallel)
# registerDoParallel(4)

# Cross-validate a list of model formulas in parallel
# Make sure to uncomment the parallel argument
cross_validate(
  data,
  formulas = formulas,
  family = "gaussian"
  #, parallel = TRUE  # Uncomment
)

cross_validate_fn
Cross-validate custom model functions for model selection

Description

[Experimental]
Cross-validate your model function with one or multiple model formulas at once. Perform repeated cross-validation. Preprocess the train/test split within the cross-validation. Perform hyperparameter tuning with grid search. Returns results in a tibble for easy comparison, reporting and further analysis.

Compared to cross_validate(), this function allows you supply a custom model function, a predict function, a preprocess function and the hyperparameter values to cross-validate.
Supports regression and classification (binary and multiclass). See `type`.

Note that some metrics may not be computable for some types of model objects.

**Usage**

cross_validate_fn(
  data,
  formulas,
  type,
  model_fn,
  predict_fn,
  preprocess_fn = NULL,
  preprocess_once = FALSE,
  hyperparameters = NULL,
  fold_cols = ".folds",
  cutoff = 0.5,
  positive = 2,
  metrics = list(),
  rm_nc = FALSE,
  parallel = FALSE,
  verbose = TRUE
)

**Arguments**

data data.frame.

Must include one or more grouping factors for identifying folds - as made with `groupdata2::fold()`.

formulas Model formulas as strings. (Character)

Will be converted to `formula` objects before being passed to `model_fn`.

E.g. c("y~x", "y~z").

Can contain random effects.

E.g. c("y~x+(1|r)", "y~z+(1|r)").

type Type of evaluation to perform:

"gaussian" for regression (like linear regression).

"binomial" for binary classification.

"multinomial" for multiclass classification.

model_fn Model function that returns a fitted model object. Will usually wrap an existing model function like `e1071::svm` or `nnet::multinom`.

Must have the following function arguments:

function(train_data, formula, hyperparameters)

predict_fn Function for predicting the targets in the test folds/sets using the fitted model object. Will usually wrap `stats::predict()`, but doesn’t have to.

Must have the following function arguments:

function(test_data, model, formula, hyperparameters, train_data)

Must return predictions in the following formats, depending on `type`:

1. For regression, predictions should be numeric.
2. For binary classification, predictions should be numeric or factor, with values 0 or 1 indicating the class.
3. For multiclass classification, predictions should be a factor indicating the predicted class.
**Binomial:** vector or one-column matrix / data.frame with probabilities (0-1) of the second class, alphabetically. E.g.: 
\[c(0.3, 0.5, 0.1, 0.5)\]
N.B. When unsure whether a model type produces probabilities based off the alphabetic order of your classes, using 0 and 1 as classes in the dependent variable instead of the class names should increase the chance of getting probabilities of the right class.

**Gaussian:** vector or one-column matrix / data.frame with the predicted value. E.g.: 
\[c(3.7, 0.9, 1.2, 7.3)\]

**Multinomial:** data.frame with one column per class containing probabilities of the class. Column names should be identical to how the class names are written in the target column. E.g.:

<table>
<thead>
<tr>
<th>class_1</th>
<th>class_2</th>
<th>class_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.269</td>
<td>0.528</td>
<td>0.203</td>
</tr>
<tr>
<td>0.368</td>
<td>0.322</td>
<td>0.310</td>
</tr>
<tr>
<td>0.375</td>
<td>0.371</td>
<td>0.254</td>
</tr>
</tbody>
</table>

**preprocess_fn** Function for preprocessing the training and test sets.

Can, for instance, be used to standardize both the training and test sets with the scaling and centering parameters from the training set.

Must have the following function arguments:

```r
function(train_data, test_data, formula, hyperparameters)
```

Must return a list with the preprocessed `train_data` and `test_data`. It may also contain a tibble with the parameters used in preprocessing:

```r
list("train" = train_data, 
     "test" = test_data, 
     "parameters" = preprocess_parameters)
```

Additional elements in the returned list will be ignored.

The optional parameters tibble will be included in the output. It could have the following format:

<table>
<thead>
<tr>
<th>Measure</th>
<th>var_1</th>
<th>var_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>37.921</td>
<td>88.231</td>
</tr>
<tr>
<td>SD</td>
<td>12.4</td>
<td>5.986</td>
</tr>
</tbody>
</table>

N.B. When `preprocess_once` is FALSE, the current formula and hyperparameters will be provided. Otherwise, these arguments will be NULL.

**preprocess_once** Whether to apply the preprocessing once (ignoring the formula and hyperparameters arguments in `preprocess_fn`) or for every model separately. (Logical)

When preprocessing does not depend on the current formula or hyperparameters, we can do the preprocessing of each train/test split once, to save time. This **may require holding a lot more data in memory** though, why it is not the default setting.
hyperparameters

Either a named list with hyperparameter values to combine in a grid or a data.frame with one row per hyperparameter combination.

**Named list for grid search:** Add ".n" to sample the combinations. Can be the number of combinations to use, or a percentage between 0 and 1.

E.g.

```r
list(".n" = 10, # sample 10 combinations
  "lrn_rate" = c(0.1, 0.01, 0.001),
  "h_layers" = c(10, 100, 1000),
  "drop_out" = runif(5, 0.3, 0.7))
```

data.frame with specific hyperparameter combinations: One row per combination to test.

E.g.

```
lrn_rate  h_layers  drop_out
0.1        10       0.65
0.1        1000     0.65
0.01       1000     0.63
...
```

fold_cols Name(s) of grouping factor(s) for identifying folds. (Character)
Include names of multiple grouping factors for repeated cross-validation.

cutoff Threshold for predicted classes. (Numeric)

N.B. **Binomial models only**

positive Level from dependent variable to predict. Either as character (preferable) or level index (1 or 2 - alphabetically).

E.g. if we have the levels "cat" and "dog" and we want "dog" to be the positive class, we can either provide "dog" or 2, as alphabetically, "dog" comes after "cat".

**Note:** For reproducibility, it's preferable to specify the name directly, as different locales may sort the levels differently.

Used when calculating confusion matrix metrics and creating ROC curves.

The Process column in the output can be used to verify this setting.

N.B. Only affects evaluation metrics, not the model training or returned predictions.

N.B. **Binomial models only**.

metrics list for enabling/disabling metrics.

E.g. `list("RMSE" = FALSE)` would remove RMSE from the regression results, and `list("Accuracy" = TRUE)` would add the regular Accuracy metric to the classification results. Default values (TRUE/FALSE) will be used for the remaining available metrics.

You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why f.i. `list("all" = FALSE, "RMSE" = TRUE)` would return only the RMSE metric.

The list can be created with `gaussian_metrics()`, `binomial_metrics()`, or `multinomial_metrics()`.

Also accepts the string "all".

rm_nc Remove non-converged models from output. (Logical)
cross_validate_fn

parallel  Whether to cross-validate the list of models in parallel. (Logical)
Remember to register a parallel backend first. E.g. with doParallel::registerDoParallel.

verbose Whether to message process information like the number of model instances to fit. (Logical)

Details

Packages used:

Results:

Shared:
AIC: stats::AIC
AICc: MuMIn::AICc
BIC: stats::BIC

Gaussian:
r2m: MuMIn::r.squaredGLMM
r2c: MuMIn::r.squaredGLMM

Binomial and Multinomial:
ROC and related metrics:
Binomial: pROC::roc
Multinomial: pROC::multiclass.roc

Value
tibble with results for each model.

N.B. The Fold column in the nested tibbles contains the test fold in that train/test split.

Shared across families:
A nested tibble with coefficients of the models from all iterations. The coefficients are extracted from the model object with parameters::model_parameters() or coef() (with some restrictions on the output). If these attempts fail, a default coefficients tibble filled with NAs is returned.

Nested tibble with the used preprocessing parameters, if a passed preprocess_fn returns the parameters in a tibble.

Number of total folds.
Number of fold columns.

Count of convergence warnings, using a limited set of keywords (e.g. "convergence"). If a convergence warning does not contain one of these keywords, it will be counted with other warnings. Consider discarding models that did not converge on all iterations. Note: you might still see results, but these should be taken with a grain of salt!

Nested tibble with the warnings and messages caught for each model.

A nested Process information object with information about the evaluation.

Name of dependent variable.
Names of fixed effects.
Names of random effects, if any.
Gaussian Results:

Average RMSE, MAE, NRMSE(IQR), RRSE, RAE, RMSLE of all the iterations*, omitting potential NAs from non-converged iterations.
See the additional metrics (disabled by default) at ?gaussian_metrics.
A nested tibble with the predictions and targets.
A nested tibble with the non-averaged results from all iterations.

* In repeated cross-validation, the metrics are first averaged for each fold column (repetition) and then averaged again.

Binomial Results:

Based on the collected predictions from the test folds*, a confusion matrix and a ROC curve are created to get the following:
ROC:
AUC, Lower CI, and Upper CI
Confusion Matrix:
Balanced Accuracy, F1, Sensitivity, Specificity, Positive Predictive Value, Negative Predictive Value, Kappa, Detection Rate, Detection Prevalence, Prevalence, and MCC (Matthews correlation coefficient).
See the additional metrics (disabled by default) at ?binomial_metrics.
Also includes:
A nested tibble with predictions, predicted classes (depends on cutoff), and the targets. Note, that the predictions are not necessarily of the specified positive class, but of the model’s positive class (second level of dependent variable, alphabetically).
The pROC::roc ROC curve object(s).
A nested tibble with the confusion matrix/matrices. The Pos_ columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the "positive" class. I.e. the level you wish to predict.
A nested tibble with the results from all fold columns.
The name of the Positive Class.

* In repeated cross-validation, an evaluation is made per fold column (repetition) and averaged.

Multinomial Results:

For each class, a one-vs-all binomial evaluation is performed. This creates a Class Level Results tibble containing the same metrics as the binomial results described above (excluding MCC, AUC, Lower CI and Upper CI), along with a count of the class in the target column (Support). These metrics are used to calculate the macro metrics. The nested class level results tibble is also included in the output tibble, and could be reported along with the macro and overall metrics.
The output tibble contains the macro and overall metrics. The metrics that share their name with the metrics in the nested class level results tibble are averages of those metrics (note: does not remove NAs before averaging). In addition to these, it also includes the Overall Accuracy and the multiclass MCC.
Other available metrics (disabled by default, see metrics): Accuracy, multiclass AUC, Weighted Balanced Accuracy, Weighted Accuracy, Weighted F1, Weighted Sensitivity, Weighted Sensitivity,
cross_validate_fn

Weighted Specificity, Weighted Pos Pred Value, Weighted Neg Pred Value, Weighted Kappa, Weighted Detection Rate, Weighted Detection Prevalence, and Weighted Prevalence. Note that the "Weighted" average metrics are weighted by the Support. Also includes:

A nested tibble with the predictions, predicted classes, and targets. A list of ROC curve objects when AUC is enabled. A nested tibble with the multiclass Confusion Matrix.

Class Level Results

Besides the binomial evaluation metrics and the Support, the nested class level results tibble also contains a nested tibble with the Confusion Matrix from the one-vs-all evaluation. The Pos_ columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the "positive" class. In our case, 1 is the current class and 0 represents all the other classes together.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other validation functions: cross_validate(), validate_fn(), validate()

Examples

# Attach packages
library(cvms)
library(groupdata2) # fold()
library(dplyr) # %>% arrange() mutate()

# Note: More examples of custom functions can be found at:
# model_fn: model_functions()
# predict_fn: predict_functions()
# preprocess_fn: preprocess_functions()

# Data is part of cvms
data <- participant.scores

# Set seed for reproducibility
set.seed(7)

# Fold data
data <- fold(
data,
k = 4,
cat_col = "diagnosis",
id_col = "participant"
) %>%
mutate(diagnosis = as.factor(diagnosis)) %>%
arrange(.folds)

# Cross-validate multiple formulas

formulas_gaussian <- c(
"score ~ diagnosis",
"score ~ age"
)
cross_validate_fn

formulas_binomial <- c(
  "diagnosis ~ score",
  "diagnosis ~ age"
)

# Gaussian

# Create model function that returns a fitted model object
lm_model_fn <- function(train_data, formula, hyperparameters) {
  lm(formula = formula, data = train_data)
}

# Create predict function that returns the predictions
lm_predict_fn <- function(test_data, model, formula, hyperparameters, train_data) {
  stats::predict(
    object = model,
    newdata = test_data,
    type = "response",
    allow.new.levels = TRUE
  )
}

# Cross-validate the model function
cross_validate_fn(
  data,
  formulas = formulas_gaussian,
  type = "gaussian",
  model_fn = lm_model_fn,
  predict_fn = lm_predict_fn,
  fold_cols = ".folds"
)

# Binomial

# Create model function that returns a fitted model object
glm_model_fn <- function(train_data, formula, hyperparameters) {
  glm(formula = formula, data = train_data, family = "binomial")
}

# Create predict function that returns the predictions
glm_predict_fn <- function(test_data, model, formula, hyperparameters, train_data) {
  stats::predict(
    object = model,
    newdata = test_data,
    type = "response",
    allow.new.levels = TRUE
  )
}

# Cross-validate the model function
cross_validate_fn(
  data,
  formulas = formulas_binomial,
  type = "binomial",
  model_fn = glm_model_fn,
  predict_fn = glm_predict_fn,
  fold_cols = ".folds"
)

# Support Vector Machine (svm)
# with hyperparameter tuning
#
# Only run if the `e1071` package is installed
if (requireNamespace("e1071", quietly = TRUE)){

# Create model function that returns a fitted model object
# We use the hyperparameters arg to pass in the kernel and cost values
svm_model_fn <- function(train_data, formula, hyperparameters) {
  # Expected hyperparameters:
  # - kernel
  # - cost
  if (!"kernel" %in% names(hyperparameters))
    stop("hyperparameters' must include 'kernel'")
  if (!"cost" %in% names(hyperparameters))
    stop("hyperparameters' must include 'cost'")

e1071::svm(
  formula = formula,
  data = train_data,
  kernel = hyperparameters["kernel"],
  cost = hyperparameters["cost"],
  scale = FALSE,
  type = "C-classification",
  probability = TRUE
)
}

# Create predict function that returns the predictions
svm_predict_fn <- function(test_data, model, formula,
                            hyperparameters, train_data) {
  predictions <- stats::predict(
    object = model,
    newdata = test_data,
    allow.new.levels = TRUE,
    probability = TRUE
  )

  # Extract probabilities
  probabilities <- dplyr::as_tibble(
    attr(predictions, "probabilities")
  )

  # Return second column
  probabilities[[2]]
# Specifying Hyperparameters to Try

The optional ".n" samples 4 combinations

```r
svm_hparams <- list(
  ".n" = 4,
  "kernel" = c("linear", "radial"),
  "cost" = c(1, 5, 10)
)
```

# Cross-Validate the Model Function

```r
cv <- cross_validate_fn(
  data,
  formulas = formulas_binomial,
  type = "binomial",
  model_fn = svm_model_fn,
  predict_fn = svm_predict_fn,
  hyperparameters = svm_hparams,
  fold_cols = ".folds"
)
```

# The `HParams` column has the nested hyperparameter values

```r
cv %>%
  select(Dependent, Fixed, HParams, `Balanced Accuracy`, F1, AUC, MCC) %>%
  tidyr::unnest(cols = "HParams") %>%
  arrange(desc("Balanced Accuracy"), desc(F1))
```

# Use parallelization

The below examples show the speed gains when running in parallel

```r
# Attach doParallel and register four cores
# Uncomment:
# library(doParallel)
# registerDoParallel(4)

# Specify hyperparameters such that we will cross-validate 20 models
hparams <- list(
  "kernel" = c("linear", "radial"),
  "cost" = 1:5
)

# Cross-validate a list of 20 models in parallel
# Make sure to uncomment the parallel argument
system.time({
  cross_validate_fn(
    data,
    formulas = formulas_gaussian,
    type = "gaussian",
    model_fn = svm_model_fn,
    predict_fn = svm_predict_fn,
    hyperparameters = hparams,
    fold_cols = ".folds"
  )
})
```
Cross-validate a list of 20 models sequentially:

```r
system.time({
cross_validate_fn(
data,
  formulas = formulas_gaussian,
  type = "gaussian",
  model_fn = svm_model_fn,
  predict_fn = svm_predict_fn,
  hyperparameters = hparams,
  fold_cols = ",.folds"
  , parallel = TRUE  # Uncomment
)
})
```

# closes `e1071` package check

---

cvms: A package for cross-validating regression and classification models

**Description**

Perform (repeated) cross-validation on a list of model formulas. Validate the best model on a validation set. Perform baseline evaluations on your test set. Generate model formulas by combining your fixed effects. Evaluate predictions from an external model.

**Details**

Returns results in a tibble for easy comparison, reporting and further analysis.

The main functions are: `cross_validate()`, `cross_validate_fn()`, `validate()`, `validate_fn()`, `baseline()`, and `evaluate()`.

**Author(s)**

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

---

evaluate

**Description**

Evaluate your model’s performance

[Maturing]

Evaluate your model’s predictions on a set of evaluation metrics.
Create ID-aggregated evaluations by multiple methods.
Currently supports regression and classification (binary and multiclass). See `type`.
Usage

```r
evaluate(
  data,
  target_col,
  prediction_cols,
  type,
  id_col = NULL,
  id_method = "mean",
  apply_softmax = FALSE,
  cutoff = 0.5,
  positive = 2,
  metrics = list(),
  include_predictions = TRUE,
  parallel = FALSE,
  models = deprecated()
)
```

Arguments

data: data.frame with predictions, targets and (optionally) an ID column. Can be grouped with `group_by`.

Multinomial: When `type` is "multinomial", the predictions can be passed in one of two formats.

*Probabilities (Preferable):* One column per class with the probability of that class. The columns should have the name of their class, as they are named in the target column. E.g.:

```r
class_1  class_2  class_3  target
0.269    0.528   0.203    class_2
0.368    0.322   0.310    class_3
0.375    0.371   0.254    class_2
...      ...     ...      ...
```

*Classes:* A single column of type character with the predicted classes. E.g.:

```r
prediction  target
class_2     class_2
class_1     class_3
class_1     class_2
...         ...
```

Binomial: When `type` is "binomial", the predictions can be passed in one of two formats.

*Probabilities (Preferable):* One column with the probability of class being the second class alphabetically (1 if classes are 0 and 1). E.g.:

```r
prediction  target
0.769       1
0.368       1
0.375       0
```
Note: At the alphabetical ordering of the class labels, they are of type character, why e.g. 100 would come before 7.

**Classes:**
A single column of type character with the predicted classes. E.g.:

<table>
<thead>
<tr>
<th>prediction</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>class_0</td>
<td>class_1</td>
</tr>
<tr>
<td>class_1</td>
<td>class_1</td>
</tr>
<tr>
<td>class_1</td>
<td>class_0</td>
</tr>
</tbody>
</table>

Note: The prediction column will be converted to the probability 0.0 for the first class alphabetically and 1.0 for the second class alphabetically.

**Gaussian:** When `type` is "gaussian", the predictions should be passed as one column with the predicted values. E.g.:

<table>
<thead>
<tr>
<th>prediction</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.9</td>
<td>30.2</td>
</tr>
<tr>
<td>33.2</td>
<td>27.1</td>
</tr>
<tr>
<td>23.4</td>
<td>21.3</td>
</tr>
</tbody>
</table>

**target_col** Name of the column with the true classes/values in `data`.

When `type` is "multinomial", this column should contain the class names, not their indices.

**prediction_cols** Name(s) of column(s) with the predictions.

Columns can be either numeric or character depending on which format is chosen. See `data` for the possible formats.

**type** Type of evaluation to perform:
"gaussian" for regression (like linear regression).
"binomial" for binary classification.
"multinomial" for multiclass classification.

**id_col** Name of ID column to aggregate predictions by.

N.B. Current methods assume that the target class/value is constant within the IDs.

N.B. When aggregating by ID, some metrics may be disabled.

**id_method** Method to use when aggregating predictions by ID. Either "mean" or "majority".

When `type` is gaussian, only the "mean" method is available.

**mean:** The average prediction (value or probability) is calculated per ID and evaluated. This method assumes that the target class/value is constant within the IDs.

**majority:** The most predicted class per ID is found and evaluated. In case of a tie, the winning classes share the probability (e.g. P = 0.5 each when two majority classes). This method assumes that the target class/value is constant within the IDs.
evaluate

apply_softmax
Whether to apply the softmax function to the prediction columns when `type` is "multinomial".
N.B. Multinomial models only.

cutoff
Threshold for predicted classes. (Numeric)
N.B. Binomial models only.

positive
Level from dependent variable to predict. Either as character (preferable) or level index (1 or 2 - alphabetically).
E.g. if we have the levels "cat" and "dog" and we want "dog" to be the positive class, we can either provide "dog" or 2, as alphabetically, "dog" comes after "cat".

Note: For reproducibility, it’s preferable to specify the name directly, as different locales may sort the levels differently.

metrics
list for enabling/disabling metrics.
E.g. list("RMSE" = FALSE) would remove RMSE from the regression results, and list("Accuracy" = TRUE) would add the regular Accuracy metric to the classification results. Default values (TRUE/FALSE) will be used for the remaining available metrics.

You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why f.i. list("all" = FALSE, "RMSE" = TRUE) would return only the RMSE metric.

The list can be created with gaussian_metrics(), binomial_metrics(), or multinomial_metrics().
Also accepts the string "all".

include_predictions
Whether to include the predictions in the output as a nested tibble. (Logical)

parallel
Whether to run evaluations in parallel, when `data` is grouped with group_by.

models
Deprecated.

Details

Packages used:

**Binomial and Multinomial:**
ROC and AUC:
Binomial: pROC::roc
Multinomial: pROC::multiclass.roc

Value

Gaussian Results:

tibble containing the following metrics by default:
evaluate

Average RMSE, MAE, NRMSE(IQR), RRSE, RAE, RMSLE.
See the additional metrics (disabled by default) at ?gaussian_metrics.
Also includes:
A nested tibble with the Predictions and targets.
A nested Process information object with information about the evaluation.

---

Binomial Results:

A tibble with the following evaluation metrics, based on a confusion matrix and a ROC curve fitted to the predictions:
- **Confusion Matrix**: Balanced Accuracy, Accuracy, F1, Sensitivity, Specificity, Positive Predictive Value, Negative Predictive Value, Kappa, Detection Rate, Detection Prevalence, Prevalence, and MCC (Matthews correlation coefficient).
- **ROC**: AUC, Lower CI, and Upper CI
Note, that the ROC curve is only computed if AUC is enabled. See metrics.
Also includes:
A nested tibble with the predictions and targets.
A list of ROC curve objects (if computed).
A nested tibble with the confusion matrix. The Pos_ columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the "positive" class. I.e. the level you wish to predict.
A nested Process information object with information about the evaluation.

---

Multinomial Results:

For each class, a one-vs-all binomial evaluation is performed. This creates a Class Level Results tibble containing the same metrics as the binomial results described above (excluding Accuracy, MCC, AUC, Lower CI and Upper CI), along with a count of the class in the target column (Support). These metrics are used to calculate the macro metrics. The nested class level results tibble is also included in the output tibble, and could be reported along with the macro and overall metrics.
The output tibble contains the macro and overall metrics. The metrics that share their name with the metrics in the nested class level results tibble are averages of those metrics (note: does not remove NAs before averaging). In addition to these, it also includes the Overall Accuracy and the multiclass MCC.
Other available metrics (disabled by default, see metrics): Accuracy, multiclass AUC, Weighted Balanced Accuracy, Weighted Accuracy, Weighted F1, Weighted Sensitivity, Weighted Specificity, Weighted Pos Pred Value, Weighted Neg Pred Value, Weighted Kappa, Weighted Detection Rate, Weighted Detection Prevalence, and Weighted Prevalence.
Note that the "Weighted" average metrics are weighted by the Support.
When having a large set of classes, consider keeping AUC disabled.
Also includes:
A nested tibble with the Predictions and targets.
A list of ROC curve objects when AUC is enabled.
A nested tibble with the multiclass Confusion Matrix.
A nested Process information object with information about the evaluation.
Class Level Results:
Besides the binomial evaluation metrics and the Support, the nested class level results tibble also contains a nested tibble with the Confusion Matrix from the one-vs-all evaluation. The Pos. columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the "positive" class. In our case, 1 is the current class and 0 represents all the other classes together.

Author(s)
Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also
Other evaluation functions: binomial_metrics(), confusion_matrix(), evaluate_residuals(), gaussian_metrics(), multinomial_metrics()

Examples
# Attach packages
library(cvms)
library(dplyr)

# Load data
data <- participant.scores

# Fit models
gaussian_model <- lm(age ~ diagnosis, data = data)
binomial_model <- glm(diagnosis ~ score, data = data)

# Add predictions
data[["gaussian_predictions"]]
 prediction_cols = "gaussian_predictions",
type = "gaussian"

# Gaussian evaluation
evaluate("gaussian")

data = data, target_col = "age",

# Binomial evaluation
evaluate(

data = data, target_col = "diagnosis",

# Multinomial
# Create a tibble with predicted probabilities and targets
data_mc <- multiclass_probability_tibble(
  num_classes = 3, num_observations = 45,
  apply_softmax = TRUE, FUN = runif,
  class_name = "class_",
  add_targets = TRUE
)

class_names <- paste0("class_", 1:3)

# Multinomial evaluation
evaluate(
  data = data_mc, target_col = "Target",
  prediction_cols = class_names,
  type = "multinomial"
)

# ID evaluation

# Gaussian ID evaluation
# Note that 'age' is the same for all observations
# of a participant
evaluate(
  data = data, target_col = "age",
  prediction_cols = "gaussian_predictions",
  id_col = "participant",
  type = "gaussian"
)

# Binomial ID evaluation
evaluate(
  data = data, target_col = "diagnosis",
  prediction_cols = "binomial_predictions",
  id_col = "participant",
  id_method = "mean", # alternatively: "majority"
  type = "binomial"
)

# Multinomial ID evaluation
# Add IDs and new targets (must be constant within IDs)
data_mc[["Target"]]<- NULL
data_mc[["ID"]]<- rep(1:9, each = 5)
id_classes <- tibble::tibble(
  "ID" = 1:9,
  "Target" = sample(x = class_names, size = 9, replace = TRUE)
)
data_mc <- data_mc %>%
  dplyr::left_join(id_classes, by = "ID")

# Perform ID evaluation
evaluate(
  data = data_mc, target_col = "Target",
  prediction_cols = class_names,
  id_col = "ID"
Evaluate residuals from a regression task

evaluate_residuals

id_method = "mean", # alternatively: "majority"
type = "multinomial"
)

# Training and evaluating a multinomial model with nnet
#
# Only run if 'nnet' is installed
if (requireNamespace("nnet", quietly = TRUE)){
  # Create a data frame with some predictors and a target column
class_names <- paste0("class_", 1:4)
data_for_nnet <- multiclass_probability_tibble(
  num_classes = 3, # Here, number of predictors
  num_observations = 30,
  apply_softmax = FALSE,
  FUN = rnorm,
  class_name = "predictor_"
)
  dplyr::mutate(Target = sample(
    class_names,
    size = 30,
    replace = TRUE
  ))

  # Train multinomial model using the nnet package
  mn_model <- nnet::multinom(
    "Target ~ predictor_1 + predictor_2 + predictor_3",
    data = data_for_nnet
  )

  # Predict the targets in the dataset
  # (we would usually use a test set instead)
predictions <- predict(
    mn_model,
    data_for_nnet,
    type = "probs"
  )
  dplyr::as_tibble()

  # Add the targets
  predictions["Target"] <- data_for_nnet["Target"]

  # Evaluate predictions
  evaluate(
    data = predictions,
    target_col = "Target",
    prediction_cols = class_names,
    type = "multinomial"
  )
  )
}
**Description**

[Experimental]

Calculates a large set of error metrics from regression residuals.

**Note:** In most cases you should use `evaluate()` instead. It works in magrittr pipes (e.g. `%>%`) and with `dplyr::group_by()`. `evaluate_residuals()` is more lightweight and may be preferred in programming when you don’t need the extra stuff in `evaluate()`.

**Usage**

```r
evaluate_residuals(data, target_col, prediction_col, metrics = list())
```

**Arguments**

- `data` data.frame with predictions and targets.
- `target_col` Name of the column with the true values in `data`.
- `prediction_col` Name of column with the predicted values in `data`.
- `metrics` list for enabling/disabling metrics. E.g. `list("RMSE" = FALSE)` would disable RMSE. Default values (TRUE/FALSE) will be used for the remaining available metrics. You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why for instance `list("all" = FALSE, "RMSE" = TRUE)` would return only the RMSE metric. The list can be created with `gaussian_metrics()`. Also accepts the string "all".

**Details**

The metric formulas are listed in `The Available Metrics` vignette.

**Value**

tibble data.frame with the calculated metrics.

The following metrics are available (see `metrics`):

<table>
<thead>
<tr>
<th>Metric</th>
<th>Name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Absolute Error</td>
<td>&quot;MAE&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Root Mean Square Error</td>
<td>&quot;RMSE&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Normalized RMSE (by target range)</td>
<td>&quot;NRMSE(RNG)&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>Normalized RMSE (by target IQR)</td>
<td>&quot;NRMSE(IQR)&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Normalized RMSE (by target STD)</td>
<td>&quot;NRMSE(STD)&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>Normalized RMSE (by target mean)</td>
<td>&quot;NRMSE(AVG)&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>Relative Squared Error</td>
<td>&quot;RSE&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>Root Relative Squared Error</td>
<td>&quot;RRSE&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Relative Absolute Error</td>
<td>&quot;RAE&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Root Mean Squared Log Error</td>
<td>&quot;RMSLE&quot;</td>
<td>Enabled</td>
</tr>
<tr>
<td>Mean Absolute Log Error</td>
<td>&quot;MALE&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>Mean Absolute Percentage Error</td>
<td>&quot;MAPE&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>Mean Squared Error</td>
<td>&quot;MSE&quot;</td>
<td>Disabled</td>
</tr>
<tr>
<td>Total Absolute Error</td>
<td>&quot;TAE&quot;</td>
<td>Disabled</td>
</tr>
</tbody>
</table>
Description

[Experimental]

Creates a list of font settings for plotting with cvms plotting functions.

NOTE: This is very experimental and will likely change.

Usage

```r
font(
  size = NULL,
  color = NULL,
  alpha = NULL,
  nudge_x = NULL,
  nudge_y = NULL,
  angle = NULL,
  family = NULL,
  fontface = NULL,
  hjust = NULL,
)```

See Also

Other evaluation functions: `binomial_metrics()`, `confusion_matrix()`, `evaluate()`, `gaussian_metrics()`, `multinomial_metrics()`
gaussian_metrics

vjust = NULL,
lineheight = NULL,
digits = NULL,
prefix = NULL,
suffix = NULL
)

Arguments

size, color, alpha, nudge_x, nudge_y, angle, family, fontface, hjust, vjust, lineheight
As passed to ggplot2::geom_text.
digits Number of digits to round to. If negative, no rounding will take place.
prefix A string prefix.
suffix A string suffix.

Value

List of settings.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other plotting functions: plot_confusion_matrix(), plot_metric_density(), plot_probabilities_ecdf(), plot_probabilities(), sum_tile_settings()

gaussian_metrics Select metrics for Gaussian evaluation

Description

[Experimental]
Enable/disable metrics for Gaussian evaluation. Can be supplied to the `metrics` argument in many of the cvms functions.
Note: Some functions may have slightly different defaults than the ones supplied here.

Usage

gaussian_metrics(
  all = NULL,
  rmse = NULL,
  mae = NULL,
  nrmse_rng = NULL,
  nrmse_iqr = NULL,
  nrmse_std = NULL,
  nrmse_avg = NULL,
  rae = NULL,
  rse = NULL,
)
gaussian_metrics

rrse = NULL,
rmsle = NULL,
mae = NULL,
mape = NULL,
mse = NULL,
tae = NULL,
tse = NULL,
r2m = NULL,
r2c = NULL,
aic = NULL,
aicc = NULL,
bic = NULL
)

Arguments

all Enable/disable all arguments at once. (Logical)
Specifying other metrics will overwrite this, why you can use (all = FALSE,
rrse = TRUE) to get only the RMSE metric.

rmse RMSE. (Default: TRUE)
Root Mean Square Error.

mae MAE. (Default: TRUE)
Mean Absolute Error.

nrmse_rng NRMSE(RNG). (Default: FALSE)
Normalized Root Mean Square Error (by target range).

nrmse_iqr NRMSE(IQR). (Default: TRUE)
Normalized Root Mean Square Error (by target interquartile range).

nrmse_std NRMSE(STD). (Default: FALSE)
Normalized Root Mean Square Error (by target standard deviation).

nrmse_avg NRMSE(AVG). (Default: FALSE)
Normalized Root Mean Square Error (by target mean).

rae RAE. (Default: TRUE)
Relative Absolute Error.

rse RSE. (Default: FALSE)
Relative Squared Error.

rrse RRSE. (Default: TRUE)
Root Relative Squared Error.

rmsle RMSLE. (Default: TRUE)
Root Mean Square Log Error.

male MALE. (Default: FALSE)
Mean Absolute Log Error.

mape MAPE. (Default: FALSE)
Mean Absolute Percentage Error.

mse MSE. (Default: FALSE)
Mean Square Error.

tae TAE. (Default: FALSE)
Total Absolute Error
Examples of model functions

Description

[Experimental]
Examples of model functions that can be used in `cross_validate_fn()`. They can either be used directly or be starting points.

The `update_hyperparameters()` function updates the list of hyperparameters with default values for missing hyperparameters. You can also specify required hyperparameters.
Usage

model_functions(name)

Arguments

name

Name of model to get model function for, as it appears in the following list:

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Hyperparameters (default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;lm&quot;</td>
<td>stats::lm()</td>
<td></td>
</tr>
<tr>
<td>&quot;lmer&quot;</td>
<td>lme4::lmer()</td>
<td>REML (FALSE)</td>
</tr>
<tr>
<td>&quot;glm_binomial&quot;</td>
<td>stats::glm()</td>
<td></td>
</tr>
<tr>
<td>&quot;glmer_binomial&quot;</td>
<td>lme4::glmer()</td>
<td></td>
</tr>
<tr>
<td>&quot;svm_gaussian&quot;</td>
<td>e1071::svm()</td>
<td>kernel (&quot;radial&quot;), cost (1)</td>
</tr>
<tr>
<td>&quot;svm_binomial&quot;</td>
<td>e1071::svm()</td>
<td>kernel (&quot;radial&quot;), cost (1)</td>
</tr>
<tr>
<td>&quot;svm_multinomial&quot;</td>
<td>e1071::svm()</td>
<td>kernel (&quot;radial&quot;), cost (1)</td>
</tr>
<tr>
<td>&quot;naive_bayes&quot;</td>
<td>e1071::naiveBayes()</td>
<td>laplace (0)</td>
</tr>
</tbody>
</table>

Value

A function with the following form:

```r
function(train_data, formula, hyperparameters) {
  # Return fitted model object
}
```

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other example functions: predict_functions(), preprocess_functions(), update_hyperparameters()

most_challenging

Find the data points that were hardest to predict

Description

[Experimental] Finds the data points that, overall, were the most challenging to predict, based on a prediction metric.

Usage

```r
most_challenging(
  data,
  type,
  obs_id_col = "Observation",
  target_col = "Target",
  prediction_cols = ifelse(type == "gaussian", "Prediction", "Predicted Class"),
  threshold = 0.15,
  threshold_is = "percentage",
)```
metric = NULL,
cutoff = 0.5
)

Arguments
data data.frame with predictions, targets and observation IDs. Can be grouped by dplyr::group_by(). Predictions can be passed as values, predicted classes or predicted probabilities: N.B. Adds .Machine$double.eps to all probabilities to avoid log(0).

Multinomial: When `type` is "multinomial", the predictions can be passed in one of two formats.

Probabilities (Preferable):
One column per class with the probability of that class. The columns should have the name of their class, as they are named in the target column. E.g.:

<table>
<thead>
<tr>
<th>class_1</th>
<th>class_2</th>
<th>class_3</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.269</td>
<td>0.528</td>
<td>0.203</td>
<td>class_2</td>
</tr>
<tr>
<td>0.368</td>
<td>0.322</td>
<td>0.310</td>
<td>class_3</td>
</tr>
<tr>
<td>0.375</td>
<td>0.371</td>
<td>0.254</td>
<td>class_2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Classes:
A single column of type character with the predicted classes. E.g.:

<table>
<thead>
<tr>
<th>prediction</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>class_2</td>
<td>class_2</td>
</tr>
<tr>
<td>class_1</td>
<td>class_3</td>
</tr>
<tr>
<td>class_1</td>
<td>class_2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Binomial: When `type` is "binomial", the predictions can be passed in one of two formats.

Probabilities (Preferable): One column with the probability of class being the second class alphabetically ("dog" if classes are "cat" and "dog"). E.g.:

<table>
<thead>
<tr>
<th>prediction</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.769</td>
<td>&quot;dog&quot;</td>
</tr>
<tr>
<td>0.368</td>
<td>&quot;dog&quot;</td>
</tr>
<tr>
<td>0.375</td>
<td>&quot;cat&quot;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Note: At the alphabetical ordering of the class labels, they are of type character, why e.g. 100 would come before 7.

Classes:
A single column of type character with the predicted classes. E.g.:

<table>
<thead>
<tr>
<th>prediction</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>class_0</td>
<td>class_1</td>
</tr>
<tr>
<td>class_1</td>
<td>class_1</td>
</tr>
<tr>
<td>class_1</td>
<td>class_0</td>
</tr>
</tbody>
</table>
**Gaussian:** When `type` is "gaussian", the predictions should be passed as one column with the predicted values. E.g.:

<table>
<thead>
<tr>
<th>prediction</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.9</td>
<td>30.2</td>
</tr>
<tr>
<td>33.2</td>
<td>27.1</td>
</tr>
<tr>
<td>23.4</td>
<td>21.3</td>
</tr>
</tbody>
</table>

**type**
Type of task used to get the predictions:
"gaussian" for regression (like linear regression).
"binomial" for binary classification.
"multinomial" for multiclass classification.

**obs_id_col**
Name of column with observation IDs. This will be used to aggregate the performance of each observation.

**target_col**
Name of column with the true classes/values in `data`.

**prediction_cols**
Name(s) of column(s) with the predictions.

**threshold**
Threshold to filter observations by. Depends on `type` and `threshold_is`.
The threshold can either be a **percentage** or a **score**. For percentages, a lower threshold returns fewer observations. For scores, this depends on `type`.

**Gaussian:**

- **threshold_is** "percentage": (Approximate) percentage of the observations with the largest root mean square errors to return.
- **threshold_is** "score": Observations with a root mean square error larger than or equal to the **threshold** will be returned.

**Binomial, Multinomial:**

- **threshold_is** "percentage": (Approximate) percentage of the observations to return with:
  - **MAE**, **Cross Entropy**: Highest error scores.
  - **Accuracy**: Lowest accuracies
- **threshold_is** "score": **MAE**, **Cross Entropy**: Observations with an error score above or equal to the **threshold** will be returned.
  - **Accuracy**: Observations with an accuracy below or equal to the **threshold** will be returned.

**threshold_is**
Either "score" or "percentage". See `threshold`.

**metric**
The metric to use. If NULL, the default metric depends on the format of the prediction columns.

- **Binomial, Multinomial**: "Accuracy", "MAE" or "Cross Entropy".
  - When one prediction column with predicted **classes** is passed, the default is "Accuracy". In this configuration, the other metrics are not calculated.
  - When one or more prediction columns with predicted **probabilities** are passed, the default is "MAE". This is the Mean Absolute Error of the probability of the target class.
- **Gaussian**: Ignored. Always uses "RMSE".

**cutoff**
Threshold for predicted classes. (Numeric)

*N.B. Binomial only.*
**Value**

data.frame with the most challenging observations and their metrics. `
`~=` `>=` `/ `<=` denotes the threshold as score.

**Author(s)**

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

**Examples**

```r
# Attach packages
library(cvms)
library(dplyr)

##
## Multinomial
##

# Find the most challenging data points (per classifier)
# in the predicted.musicians dataset
# which resembles the "Predictions" tibble from the evaluation results

# Passing predicted probabilities
# Observations with 30% highest MAE scores
most_challenging(
predicted.musicians,
  obs_id_col = "ID",
  prediction_cols = c("A", "B", "C", "D"),
  type = "multinomial",
  threshold = 0.30
)

# Observations with 25% highest Cross Entropy scores
most_challenging(
predicted.musicians,
  obs_id_col = "ID",
  prediction_cols = c("A", "B", "C", "D"),
  type = "multinomial",
  threshold = 0.25,
  metric = "Cross Entropy"
)

# Passing predicted classes
# Observations with 30% lowest Accuracy scores
most_challenging(
predicted.musicians,
  obs_id_col = "ID",
  prediction_cols = "Predicted Class",
  type = "multinomial",
  threshold = 0.30
)

# The 40% lowest-scoring on accuracy per classifier
predicted.musicians %>%
dplyr::group_by(Classifier) %>%
most_challenging()
```
most_challenging

```r
obs_id_col = "ID",
prediction_cols = "Predicted Class",
type = "multinomial",
threshold = 0.40
)

# Accuracy scores below 0.05
most_challenging(
predicted.musicians,
ob_id_col = "ID",
type = "multinomial",
threshold = 0.05,
threshold_is = "score"
)

##
## Binomial
##

# Subset the predicted.musicians
binom_data <- predicted.musicians %>%
dplyr::filter(Target %in% c("A","B")) %>%
dplyr::rename(Prediction = B)

# Passing probabilities
# Observations with 30% highest MAE
most_challenging(
binom_data,
ob_id_col = "ID",
type = "binomial",
prediction_cols = "Prediction",
threshold = 0.30
)

# Observations with 30% highest Cross Entropy
most_challenging(
binom_data,
ob_id_col = "ID",
type = "binomial",
prediction_cols = "Prediction",
threshold = 0.30,
metric = "Cross Entropy"
)

# Passing predicted classes
# Observations with 30% lowest Accuracy scores
most_challenging(
binom_data,
ob_id_col = "ID",
type = "binomial",
prediction_cols = "Predicted Class",
threshold = 0.30
)

##
## Gaussian
##
```
set.seed(1)

df <- data.frame(
  "Observation" = rep(1:10, n = 3),
  "Target" = rnorm(n = 30, mean = 25, sd = 5),
  "Prediction" = rnorm(n = 30, mean = 27, sd = 7)
)

# The 20% highest RMSE scores
most_challenging(
  df,
  type = "gaussian",
  threshold = 0.2
)

# RMSE scores above 9
most_challenging(
  df,
  type = "gaussian",
  threshold = 9,
  threshold_is = "score"
)

---

multiclass_probability_tibble

*Generate a multiclass probability tibble*

**Description**

[Maturing]
Generate a tibble with random numbers containing one column per specified class. When the softmax function is applied, the numbers become probabilities that sum to 1 row-wise. Optionally, add columns with targets and predicted classes.

**Usage**

```r
multiclass_probability_tibble(
  num_classes,
  num_observations,
  apply_softmax = TRUE,
  FUN = runif,
  class_name = "class_",
  add_predicted_classes = FALSE,
  add_targets = FALSE
)
```

**Arguments**

- `num_classes`: The number of classes. Also the number of columns in the tibble.
- `num_observations`: The number of observations. Also the number of rows in the tibble.
multiclass_probability_tibble

apply_softmax  Whether to apply the softmax function row-wise. This will transform the numbers to probabilities that sum to 1 row-wise.

FUN  Function for generating random numbers. The first argument must be the number of random numbers to generate, as no other arguments are supplied.

class_name  The prefix for the column names. The column index is appended.

add_predicted_classes  Whether to add a column with the predicted classes. (Logical)
  The class with the highest value is the predicted class.

add_targets  Whether to add a column with randomly selected target classes. (Logical)

Author(s)
Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

Examples

# Attach cvms
library(cvms)

# Create a tibble with 5 classes and 10 observations
# Apply softmax to make sure the probabilities sum to 1
multiclass_probability_tibble(
  num_classes = 5,
  num_observations = 10,
  apply_softmax = TRUE
)

# Using the rnorm function to generate the random numbers
multiclass_probability_tibble(
  num_classes = 5,
  num_observations = 10,
  apply_softmax = TRUE,
  FUN = rnorm
)

# Add targets and predicted classes
multiclass_probability_tibble(
  num_classes = 5,
  num_observations = 10,
  apply_softmax = TRUE,
  FUN = rnorm,
  add_predicted_classes = TRUE,
  add_targets = TRUE
)

# Creating a custom generator function that
# exponentiates the numbers to create more "certain" predictions
rcertain <- function(n) {
  (runif(n, min = 1, max = 100)^1.4) / 100
}

multiclass_probability_tibble(
  num_classes = 5,
  num_observations = 10,
  apply_softmax = TRUE,
  FUN = rcertain
)
multinomial_metrics

Select metrics for multinomial evaluation

Description

[Experimental]
Enable/disable metrics for multinomial evaluation. Can be supplied to the `metrics` argument in many of the cvms functions.

Note: Some functions may have slightly different defaults than the ones supplied here.

Usage

```r
multinomial_metrics(
  all = NULL,
  overall_accuracy = NULL,
  balanced_accuracy = NULL,
  w_balanced_accuracy = NULL,
  accuracy = NULL,
  w_accuracy = NULL,
  f1 = NULL,
  w_f1 = NULL,
  sensitivity = NULL,
  w_sensitivity = NULL,
  specificity = NULL,
  w_specificity = NULL,
  pos_pred_value = NULL,
  w_pos_pred_value = NULL,
  neg_pred_value = NULL,
  w_neg_pred_value = NULL,
  auc = NULL,
  kappa = NULL,
  w_kappa = NULL,
  mcc = NULL,
  detection_rate = NULL,
  w_detection_rate = NULL,
  detection_prevalence = NULL,
  w_detection_prevalence = NULL,
  prevalence = NULL,
  w_prevalence = NULL,
  false_neg_rate = NULL,
  w_false_neg_rate = NULL,
  false_pos_rate = NULL,
  w_false_pos_rate = NULL,
  false_discovery_rate = NULL,
  w_false_discovery_rate = NULL,
  false_omission_rate = NULL,
  w_false_omission_rate = NULL,
)```
threat_score = NULL,
w_threat_score = NULL,
aic = NULL,
aicc = NULL,
bic = NULL
)

Arguments

all Enable/disable all arguments at once. (Logical)

Specifying other metrics will overwrite this, why you can use (all = FALSE, accuracy = TRUE) to get only the Accuracy metric.

overall_accuracy Overall Accuracy (Default: TRUE)
balanced_accuracy Balanced Accuracy (Default: TRUE)
w_balanced_accuracy Weighted Balanced Accuracy (Default: FALSE)
accuracy Accuracy (Default: FALSE)
w_accuracy Weighted Accuracy (Default: FALSE)
f1 F1 (Default: TRUE)
w_f1 Weighted F1 (Default: FALSE)
sensitivity Sensitivity (Default: TRUE)
w_sensitivity Weighted Sensitivity (Default: FALSE)
specificity Specificity (Default: TRUE)
w_specificity Weighted Specificity (Default: FALSE)
pos_pred_value Pos Pred Value (Default: TRUE)
w_pos_pred_value Weighted Pos Pred Value (Default: FALSE)
neg_pred_value Neg Pred Value (Default: TRUE)
w_neg_pred_value Weighted Neg Pred Value (Default: FALSE)
auc AUC (Default: FALSE)
kappa Kappa (Default: TRUE)
w_kappa Weighted Kappa (Default: FALSE)
mcc MCC (Default: TRUE)

Multiclass Matthews Correlation Coefficient.
detection_rate Detection Rate (Default: TRUE)
w_detection_rate Weighted Detection Rate (Default: FALSE)
detection_prevalence Detection Prevalence (Default: TRUE)
w_detection_prevalence Weighted Detection Prevalence (Default: FALSE)
prevalence Prevalence (Default: TRUE)
w_prevalence Weighted Prevalence (Default: FALSE)
false_neg_rate False Neg Rate (Default: FALSE)
w_false_neg_rate Weighted False Neg Rate (Default: FALSE)
false_pos_rate False Pos Rate (Default: FALSE)
w_false_pos_rate Weighted False Pos Rate (Default: FALSE)
false_discovery_rate False Discovery Rate (Default: FALSE)
w_false_discovery_rate Weighted False Discovery Rate (Default: FALSE)
false_omission_rate False Omission Rate (Default: FALSE)
w_false_omission_rate Weighted False Omission Rate (Default: FALSE)
threat_score Threat Score (Default: FALSE)
w_threat_score Weighted Threat Score (Default: FALSE)
aic AIC. (Default: FALSE)
aicc AICc. (Default: FALSE)
bic BIC. (Default: FALSE)

Author(s)
Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also
Other evaluation functions: \code{binomial_metrics()}, \code{confusion_matrix()}, \code{evaluate_residuals()}, \code{evaluate()}, \code{gaussian_metrics()}

Examples

# Attach packages
library(cvms)

# Enable only Balanced Accuracy
multinomial_metrics(all = FALSE, balanced_accuracy = TRUE)

# Enable all but Balanced Accuracy
multinomial_metrics(all = TRUE, balanced_accuracy = FALSE)

# Disable Balanced Accuracy
multinomial_metrics(balanced_accuracy = FALSE)
Description

Made-up data on 60 musicians in 4 groups for multiclass classification.

Format

A data.frame with 60 rows and 9 variables:

ID  Musician identifier, 60 levels
Age  Age of the musician. Between 17 and 66 years.
Class The class of the musician. One of "A", "B", "C", and "D".
Height Height of the musician. Between 146 and 196 centimeters.
Drums Whether the musician plays drums. 0 = No, 1 = Yes.
Bass  Whether the musician plays bass. 0 = No, 1 = Yes.
Guitar Whether the musician plays guitar. 0 = No, 1 = Yes.
Keys Whether the musician plays keys. 0 = No, 1 = Yes.
Vocals Whether the musician sings. 0 = No, 1 = Yes.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

predicted.musicians

Description

Made-up experiment data with 10 participants and two diagnoses. Test scores for 3 sessions per participant, where participants improve their scores each session.

Format

A data.frame with 30 rows and 5 variables:

participant participant identifier, 10 levels
age  age of the participant, in years
diagnosis diagnosis of the participant, either 1 or 0
score  test score of the participant, on a 0-100 scale
session testing session identifier, 1 to 3

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>
plot_confusion_matrix  Plot a confusion matrix

Description

[Experimental]

Creates a ggplot2 object representing a confusion matrix with counts, overall percentages, row percentages and column percentages. An extra row and column with sum tiles and the total count can be added.

The confusion matrix can be created with `evaluate()`. See `Examples`.

While this function is intended to be very flexible (hence the large number of arguments), the defaults should work in most cases for most users. See the `Examples`.

NEW: Our Plot Confusion Matrix web application allows using this function without code. Select from multiple design templates or make your own.

Usage

```r
plot_confusion_matrix(
  conf_matrix,
  target_col = "Target",
  prediction_col = "Prediction",
  counts_col = "N",
  sub_col = NULL,
  class_order = NULL,
  add_sums = FALSE,
  add_counts = TRUE,
  add_normalized = TRUE,
  add_row_percentages = TRUE,
  add_col_percentages = TRUE,
  diag_percentages_only = FALSE,
  rm_zero_percentages = TRUE,
  rm_zero_text = TRUE,
  add_zero_shading = TRUE,
  amount_3d_effect = 1,
  add_arrows = TRUE,
  counts_on_top = FALSE,
  palette = "Blues",
  intensity_by = "counts",
  intensity_lims = NULL,
  intensity_beyond_lims = "truncate",
  theme_fn = ggplot2::theme_minimal,
  place_x_axis_above = TRUE,
  rotate_y_text = TRUE,
  digits = 1,
  font_counts = font(),
  font_normalized = font(),
  font_row_percentages = font(),
  font_col_percentages = font(),
  arrow_size = 0.048,
)```

Arguments

conf_matrix Confusion matrix tibble with each combination of targets and predictions along with their counts.
E.g. for a binary classification:

<table>
<thead>
<tr>
<th>Target</th>
<th>Prediction</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>class_1</td>
<td>class_1</td>
<td>5</td>
</tr>
<tr>
<td>class_1</td>
<td>class_2</td>
<td>9</td>
</tr>
<tr>
<td>class_2</td>
<td>class_1</td>
<td>3</td>
</tr>
<tr>
<td>class_2</td>
<td>class_2</td>
<td>2</td>
</tr>
</tbody>
</table>

As created with the various evaluation functions in cvms, like evaluate().
An additional `sub_col` column (character) can be specified as well. Its content will replace the bottom text ('counts' by default or 'normalized' when `counts_on_top` is enabled).

Note: If you supply the results from evaluate() or confusion_matrix() directly, the confusion matrix tibble is extracted automatically, if possible.

target_col Name of column with target levels.
prediction_col Name of column with prediction levels.
counts_col Name of column with a count for each combination of the target and prediction levels.
sub_col Name of column with text to replace the bottom text ('counts' by default or 'normalized' when `counts_on_top` is enabled).
It simply replaces the text, so all settings will still be called e.g. `font_counts` etc. When other settings make it so, that no bottom text is displayed (e.g. `add_counts = FALSE`), this text is not displayed either.
class_order Names of the classes in `conf_matrix` in the desired order. When NULL, the classes are ordered alphabetically. Note that the entire set of unique classes from both `target_col` and `prediction_col` must be specified.
add_sums Add tiles with the row/column sums. Also adds a total count tile. (Logical)
The appearance of these tiles can be specified in `sums_settings`.
Note: Adding the sum tiles with a palette requires the ggnewscale package.
add_counts Add the counts to the middle of the tiles. (Logical)
add_normalized Normalize the counts to percentages and add to the middle of the tiles. (Logical)
add_row_percentages Add the row percentages, i.e. how big a part of its row the tile makes up. (Logical)
By default, the row percentage is placed to the right of the tile, rotated 90 degrees.
add_col_percentages
Add the column percentages, i.e. how big a part of its column the tile makes up.
(Logical)
By default, the row percentage is placed at the bottom of the tile.

diag_percentages_only
Whether to only have row and column percentages in the diagonal tiles. (Logical)

rm_zero_percentages
Whether to remove row and column percentages when the count is 0. (Logical)

rm_zero_text
Whether to remove counts and normalized percentages when the count is 0. (Logical)

add_zero_shading
Add image of skewed lines to zero-tiles. (Logical)
Note: Adding the zero-shading requires the rsvg and ggimage packages.

amount_3d_effect
Amount of 3D effect (tile overlay) to add. Passed as whole number from 0 (no effect) up to 6 (biggest effect). This helps separate tiles with the same intensities.
Note: The overlay may not fit the tiles in many-class cases that haven’t been tested. If the boxes do not overlap properly, simply turn it off.

add_arrows
Add the arrows to the row and col percentages. (Logical)
Note: Adding the arrows requires the rsvg and ggimage packages.

counts_on_top
Switch the counts and normalized counts, such that the counts are on top. (Logical)

palette
Color scheme. Passed directly to `palette` in ggplot2::scale_fill_distiller.
Try these palettes: "Greens", "Oranges", "Greys", "Purples", "Reds", as well as the default "Blues".
Alternatively, pass a named list with limits of a custom gradient as e.g. `list("low"="#B1F9E8", "high"="#239895")`. These are passed to ggplot2::scale_fill_gradient.

intensity_by
The measure that should control the color intensity of the tiles. Either `counts`, `normalized` or one of `log counts`, `log2 counts`, `log10 counts`, `arcsinh counts`.
For `normalized`, the color limits become 0-100 (except when `intensity_lims` are specified), why the intensities can better be compared across plots.
For the 'log*' and 'arcsinh' versions, the log/arcsinh transformed counts are used.
Note: In 'log*' transformed counts, 0-counts are set to '0', why they won't be distinguishable from 1-counts.

intensity_lims
A specific range of values for the color intensity of the tiles. Given as a numeric vector with c(min, max).
This allows having the same intensity scale across plots for better comparison of prediction sets.

intensity_beyond_lims
What to do with values beyond the `intensity_lims`. One of "truncate", "grey".

theme_fn
The ggplot2 theme function to apply.

place_x_axis_above
Move the x-axis text to the top and reverse the levels such that the "correct" diagonal goes from top left to bottom right. (Logical)
rotate_y_text  Whether to rotate the y-axis text to be vertical instead of horizontal. (Logical)
digits  Number of digits to round to (percentages only). Set to a negative number for no rounding.
        Can be set for each font individually via the font_* arguments.
font_counts  list of font settings for the counts. Can be provided with font().
font_normalized  list of font settings for the normalized counts. Can be provided with font().
font_row_percentages  list of font settings for the row percentages. Can be provided with font().
font_col_percentages  list of font settings for the column percentages. Can be provided with font().
arrow_size  Size of arrow icons. (Numeric)
        Is divided by sqrt(nrow(conf_matrix)) and passed on to ggimage::geom_icon().
arrow_nudge_from_text  Distance from the percentage text to the arrow. (Numeric)
tile_border_color  Color of the tile borders. Passed as 'colour' to ggplot2::geom_tile.
tile_border_size  Size of the tile borders. Passed as 'size' to ggplot2::geom_tile.
tile_border_linetype  Linetype for the tile borders. Passed as 'linetype' to ggplot2::geom_tile.
sums_settings  A list of settings for the appearance of the sum tiles. Can be provided with sum_tile_settings().
darkness  How dark the darkest colors should be, between 0 and 1, where 1 is darkest.
        Technically, a lower value increases the upper limit in ggplot2::scale_fill_distiller.

Details

Inspired by Antoine Sachet’s answer at https://stackoverflow.com/a/53612391/11832955

Value

A ggplot2 object representing a confusion matrix. Color intensity depends on either the counts (default) or the overall percentages.

By default, each tile has the normalized count (overall percentage) and count in the middle, the column percentage at the bottom, and the row percentage to the right and rotated 90 degrees.

In the "correct" diagonal (upper left to bottom right, by default), the column percentages are the class-level sensitivity scores, while the row percentages are the class-level positive predictive values.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other plotting functions: font(), plot_metric_density(), plot_probabilities_ecdf(), plot_probabilities(), sum_tile_settings()
Examples

# Attach cvms
library(cvms)
library(ggplot2)

# Two classes
# Create targets and predictions data frame
data <- data.frame(
  stringsAsFactors = FALSE)

# Evaluate predictions and create confusion matrix
evaluation <- evaluate(
  data = data,
  target_col = "target",
  prediction_cols = "prediction",
  type = "binomial"
)

# Inspect confusion matrix tibble
evaluation["Confusion Matrix"][[1]]

# Plot confusion matrix
# Supply confusion matrix tibble directly
plot_confusion_matrix(evaluation["Confusion Matrix"][[1]])

# Plot first confusion matrix in evaluate() output
plot_confusion_matrix(evaluation)

# Add sum tiles
plot_confusion_matrix(evaluation, add_sums = TRUE)

# Three (or more) classes

# Create targets and predictions data frame
data <- data.frame(
  stringsAsFactors = FALSE)

# Evaluate predictions and create confusion matrix
evaluation <- evaluate(
  data = data,
  target_col = "target",
  prediction_cols = "prediction",
  type = "multinomial"
)

# Inspect confusion matrix tibble

evaluation[["Confusion Matrix"]][[1]]

# Plot confusion matrix
# Supply confusion matrix tibble directly
plot_confusion_matrix(evaluation[["Confusion Matrix"]][[1]])
# Plot first confusion matrix in evaluate() output
plot_confusion_matrix(evaluation)

# Add sum tiles
plot_confusion_matrix(evaluation, add_sums = TRUE)

# Counts only
plot_confusion_matrix(
  evaluation[["Confusion Matrix"]][[1]],
  add_normalized = FALSE,
  add_row_percentages = FALSE,
  add_col_percentages = FALSE
)

# Change color palette to green
# Change theme to `theme_light`
plot_confusion_matrix(
  evaluation[["Confusion Matrix"]][[1]],
  palette = "Greens",
  theme_fn = ggplot2::theme_light
)

# Change colors palette to custom gradient
# with a different gradient for sum tiles
plot_confusion_matrix(
  evaluation[["Confusion Matrix"]][[1]],
  palette = list("low" = "#B1F9E8", "high" = "#239895"),
  sums_settings = sum_tile_settings(
    palette = list("low" = "#e9e1fc", "high" = "#BE94E6")
  ),
  add_sums = TRUE
)

# The output is a ggplot2 object
# that you can add layers to
# Here we change the axis labels
plot_confusion_matrix(evaluation[["Confusion Matrix"]][[1]]) +
  ggplot2::labs(x = "True", y = "Guess")

# Replace the bottom tile text
# with some information
# First extract confusion matrix
# Then add new column with text
cm <- evaluation[["Confusion Matrix"]][[1]]
cm[["Trials"]] <- c(
  "(8/9)", "(3/9)", 
  "(1/9)", "(3/9)", "(7/9)", "(4/9)",
  "(1/9)", "(2/9)", 
  "(8/9)"
)

# Now plot with the `sub_col` argument specified
plot_confusion_matrix(cm, sub_col = "Trials")
plot_metric_density  Density plot for a metric

Description

[Experimental]

Creates a 

\texttt{ggplot2} object with a density plot for one of the columns in the passed data.frame(s).

Note: In its current form, it is mainly intended as a quick way to visualize the results from cross-validations and baselines (random evaluations). It may change significantly in future versions.

Usage

\begin{verbatim}
plot_metric_density(
  results = NULL,
  baseline = NULL,
  metric = "",
  fill = c("darkblue", "lightblue"),
  alpha = 0.6,
  theme_fn = ggplot2::theme_minimal,
  xlim = NULL
)
\end{verbatim}

Arguments

- \texttt{results} data.frame with a metric column to create density plot for.
  To only plot the baseline, set to NULL.

- \texttt{baseline} data.frame with the random evaluations from \texttt{baseline()}. Should contain a column for the metric.
  To only plot the results, set to NULL.

- \texttt{metric} Name of the metric column in `results` to plot. (Character)

- \texttt{fill} Colors of the plotted distributions. The first color is for the `baseline`, the second for the `results`.

- \texttt{alpha} Transparency of the distribution (0 - 1).

- \texttt{theme_fn} The ggplot2 theme function to apply.

- \texttt{xlim} Limits for the x-axis. Can be set to NULL.
  E.g. \texttt{c(0, 1)}.

Value

A ggplot2 object with the density of a metric, possibly split in `Results` and `Baseline`.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>
See Also

Other plotting functions: `font()`, `plot_confusion_matrix()`, `plot_probabilities_ecdf()`, `plot_probabilities()`, `sum_tile_settings()`

Examples

```r
# Attach packages
library(cvms)
library(dplyr)

# We will use the musicians and predicted.musicians datasets
musicians
predicted.musicians

# Set seed
set.seed(42)

# Create baseline for targets
bsl <- baseline_multinomial(
  test_data = musicians,
  dependent_col = "Class",
  n = 20 # Normally 100
)

# Evaluate predictions grouped by classifier and fold column
eval <- predicted.musicians %>%
dplyr::group_by(Classifier, "Fold Column") %>%
evaluate(
  target_col = "Target",
  prediction_cols = c("A", "B", "C", "D"),
  type = "multinomial"
)

# Plot density of the Overall Accuracy metric
plot_metric_density(
  results = eval,
  baseline = bsl$random_evaluations,
  metric = "Overall Accuracy",
  xlim = c(0,1)
)

# The bulk of classifier results are much better than
# the baseline results
```

---

**Description**

Fixed effect combinations for model formulas with/without two- and three-way interactions. Up to eight fixed effects in total with up to five fixed effects per formula.
predicted.musicians

Format
A data.frame with 259,358 rows and 5 variables:

- `formula_` combination of fixed effects, separated by "+" and "*
- `max_interaction_size` maximum interaction size in the formula, up to 3
- `max_effect_frequency` maximum count of an effect in the formula, e.g. the 3 A's in "A * B + A * C + A * D"
- `num_effects` number of unique effects included in the formula
- `min_num_fixed_effects` minimum number of fixed effects required to use the formula, i.e. the index in the alphabet of the last of the alphabetically ordered effects (letters) in the formula, so 4 for the formula: "A + B + D"

Details
Effects are represented by the first eight capital letters.
Used by `combine_predictors`.

Author(s)
Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

predicted.musicians  Predicted musician groups

Description
Predictions by 3 classifiers of the 4 classes in the `musicians` dataset. Obtained with 5-fold stratified cross-validation (3 repetitions). The three classifiers were fit using `nnet::multinom`, `randomForest::randomForest`, and `e1071::svm`.

Format
A data.frame with 540 rows and 10 variables:

- `Classifier` The applied classifier. One of "nnet_multinom", "randomForest", and "e1071_svm".
- `Fold Column` The fold column name. Each is a unique 5-fold split. One of ".folds_1", ".folds_2", and ".folds_3".
- `Fold` The fold, 1 to 5.
- `ID` Musician identifier, 60 levels
- `Target` The actual class of the musician. One of "A", "B", "C", and "D".
- `A` The probability of class "A".
- `B` The probability of class "B".
- `C` The probability of class "C".
- `D` The probability of class "D".
- `Predicted Class` The predicted class. The argmax of the four probability columns.
predict_functions

Details

Used formula: "Class ~ Height + Age + Drums + Bass + Guitar + Keys + Vocals"

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

musicians

Examples

# Attach packages
library(cvms)
library(dplyr)

# Evaluate each fold column
predicted.musicians %>%
  dplyr::group_by(Classifier, `Fold Column`) %>%
  evaluate(target_col = "Target",
    prediction_cols = c("A", "B", "C", "D"),
    type = "multinomial")

# Overall ID evaluation
# I.e. if we average all 9 sets of predictions,
# how well did we predict the targets?
overall_id_eval <- predicted.musicians %>%
  evaluate(target_col = "Target",
    prediction_cols = c("A", "B", "C", "D"),
    type = "multinomial",
    id_col = "ID")

overall_id_eval
# Plot the confusion matrix
plot_confusion_matrix(overall_id_eval$`Confusion Matrix`[[1]])

predict_functions

Examples of predict_fn functions

Description

[Experimental]

Examples of predict functions that can be used in cross_validate_fn(). They can either be used directly or be starting points.

Usage

predict_functions(name)
**Arguments**

- **name**: Name of model to get predict function for, as it appears in the following table. The **Model HParams** column lists hyperparameters used in the respective model function.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Model HParams</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;lm&quot;</td>
<td>stats::lm()</td>
<td></td>
</tr>
<tr>
<td>&quot;lmer&quot;</td>
<td>lme4::lmer()</td>
<td></td>
</tr>
<tr>
<td>&quot;glm_binomial&quot;</td>
<td>stats::glm()</td>
<td>family = &quot;binomial&quot;</td>
</tr>
<tr>
<td>&quot;glmer_binomial&quot;</td>
<td>lme4::glmer()</td>
<td>family = &quot;binomial&quot;</td>
</tr>
<tr>
<td>&quot;svm_gaussian&quot;</td>
<td>e1071::svm()</td>
<td>type = &quot;eps-regression&quot;</td>
</tr>
<tr>
<td>&quot;svm_binomial&quot;</td>
<td>e1071::svm()</td>
<td>type = &quot;C-classification&quot;, probability = TRUE</td>
</tr>
<tr>
<td>&quot;svm_multinomial&quot;</td>
<td>e1071::svm()</td>
<td>type = &quot;C-classification&quot;, probability = TRUE</td>
</tr>
<tr>
<td>&quot;naive_bayes&quot;</td>
<td>e1071::naiveBayes()</td>
<td>linout = TRUE</td>
</tr>
<tr>
<td>&quot;nnet_multinom&quot;</td>
<td>nnet::multinom()</td>
<td></td>
</tr>
<tr>
<td>&quot;nnet_gaussian&quot;</td>
<td>nnet::nnet()</td>
<td></td>
</tr>
<tr>
<td>&quot;nnet_binomial&quot;</td>
<td>nnet::nnet()</td>
<td></td>
</tr>
<tr>
<td>&quot;randomForest_gaussian&quot;</td>
<td>randomForest::randomForest()</td>
<td></td>
</tr>
<tr>
<td>&quot;randomForest_binomial&quot;</td>
<td>randomForest::randomForest()</td>
<td></td>
</tr>
<tr>
<td>&quot;randomForest_multinomial&quot;</td>
<td>randomForest::randomForest()</td>
<td></td>
</tr>
</tbody>
</table>

**Value**

A function with the following form:

```r
function(test_data, model, formula, hyperparameters, train_data) {
  # Use model to predict test_data
  # Return predictions
}
```

**Author(s)**

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

**See Also**

Other example functions: `model_functions()`, `preprocess_functions()`, `update_hyperparameters()`

---

**Description**

*Experimental*

Examples of preprocess functions that can be used in `cross_validate_fn()` and `validate_fn()`. They can either be used directly or be starting points. The examples use `recipes`, but you can also use `caret::preProcess()` or similar functions. In these examples, the preprocessing will only affect the numeric predictors. You may prefer to hardcode a formula like "y ~ ." (where y is your dependent variable) as that will allow you to set `preprocess_one` to TRUE in `cross_validate_fn()` and `validate_fn()` and save time.
Usage

preprocess_functions(name)

Arguments

name Name of preprocessing function as it appears in the following list:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;standardize&quot;</td>
<td>Centers and scales the numeric predictors</td>
</tr>
<tr>
<td>&quot;range&quot;</td>
<td>Normalizes the numeric predictors to the 0-1 range</td>
</tr>
<tr>
<td>&quot;scale&quot;</td>
<td>Scales the numeric predictors to have a standard deviation of one</td>
</tr>
<tr>
<td>&quot;center&quot;</td>
<td>Centers the numeric predictors to have a mean of zero</td>
</tr>
<tr>
<td>&quot;warn&quot;</td>
<td>Identity function that throws a warning and a message</td>
</tr>
</tbody>
</table>

Value

A function with the following form:

function(train_data, test_data, formula, hyperparameters) {
  # Preprocess train_data and test_data
  # Return a list with the preprocessed datasets
  # and optionally a data frame with preprocessing parameters
  list(  
    "train" = train_data,
    "test" = test_data,
    "parameters" = tidy_parameters
  )
}

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other example functions: model_functions(), predict_functions(), update_hyperparameters()
Usage

process_info_binomial(
  data,
  target_col,
  prediction_cols,
  id_col,
  cat_levels,
  positive,
  cutoff,
  locale = NULL
)

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

## S3 method for class 'process_info_binomial'
as.character(x, ...)

process_info_multinomial(
  data,
  target_col,
  prediction_cols,
  pred_class_col,
  id_col,
  cat_levels,
  apply_softmax,
  locale = NULL
)

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

## S3 method for class 'process_info_multinomial'
as.character(x, ...)

process_info_gaussian(data, target_col, prediction_cols, id_col, locale = NULL)

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

## S3 method for class 'process_info_gaussian'
as.character(x, ...)

Arguments

data       Data frame.
target_col  Name of target column.
prediction_cols  Names of prediction columns.
id_col    Name of ID column.
cat_levels Categorical levels (classes).
### reconstruct_formulas

**Description**

[Maturing]

In the (cross-)validation results from functions like `cross_validate()`, the model formulas have been split into the columns `Dependent`, `Fixed` and `Random`. Quickly reconstruct the model formulas from these columns.

**Usage**

```r
reconstruct_formulas(results, topn = NULL)
```

**Arguments**

- `results` data.frame with results from `cross_validate()` or `validate()`. (tbl)
  
  Must contain at least the columns "Dependent" and "Fixed". For random effects, the "Random" column should be included.

- `topn` Number of top rows to return. Simply applies `head()` to the results tibble.

**Value**

- list of model formulas.

**Author(s)**

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`
select_definitions  Select model definition columns

Description

[Experimental]
Select the columns that define the models, such as the formula terms and hyperparameters.
If an expected column is not in the `results` tibble, it is simply ignored.

Usage

```r
select_definitions(results, unnest_hparams = TRUE, additional_includes = NULL)
```

Arguments

- `results` Results tibble. E.g. from `cross_validate()` or `evaluate()`.
- `unnest_hparams` Whether to unnest the HParams column. (Logical)
- `additional_includes` Names of additional columns to select. (Character)

Value

The model definition columns from the results tibble.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

select_metrics  Select columns with evaluation metrics and model definitions

Description

[Maturing]
When reporting results, we might not want all the nested tibbles and process information columns.
This function selects the evaluation metrics and model formulas only.
If an expected column is not in the `results` tibble, it is simply ignored.

Usage

```r
select_metrics(results, include_definitions = TRUE, additional_includes = NULL)
```

Arguments

- `results` Results tibble. E.g. from `cross_validate()` or `evaluate()`.
- `include_definitions` Whether to include the Dependent, Fixed and (possibly) Random and HParams columns. (Logical)
- `additional_includes` Names of additional columns to select. (Character)
Value

The results tibble with only the metric and model definition columns.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

simplify_formula  
Simplify formula with inline functions

Description

[Experimental]
Extracts all variables from a formula object and creates a new formula with all predictor variables added together without the inline functions.

E.g.:
y ~ x*z + log(a) + (1|b)
becomes
y ~ x + z + a + b.
This is useful when passing a formula to recipes::recipe() for preprocessing a dataset, as used in the preprocess_functions().

Usage

simplify_formula(formula, data = NULL, string_out = FALSE)

Arguments

formula  
Formula object.
If a string is passed, it will be converted with as.formula().
When a side only contains a NULL, it is kept. Otherwise NULLs are removed.
An intercept (1) will only be kept if there are no variables on that side of the formula.
data  
data.frame. Used to extract variables when the formula contains a ".".
string_out  
Whether to return as a string. (Logical)

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>
summarize_metrics

Examples

# Attach cvms
library(cvms)

# Create formula
f1 <- "y ~ x*z + log(a) + (1|b)"

# Simplify formula (as string)
simplify_formula(f1)

# Simplify formula (as formula)
simplify_formula(as.formula(f1))

summarize_metrics Summarize metrics with common descriptors

Description

[Experimental]
Summarizes all numeric columns. Counts the NAs and Inf's in the columns.

Usage

summarize_metrics(data, cols = NULL, na.rm = TRUE, inf.rm = TRUE)

Arguments

data data.frame with numeric columns to summarize.

cols Names of columns to summarize. Non-numeric columns are ignored. (Character)

na.rm Whether to remove NAs before summarizing. (Logical)

inf.rm Whether to remove Inf's before summarizing. (Logical)

Value

tibble where each row is a descriptor of the column.

The Measure column contains the name of the descriptor.

The NAs row is a count of the NAs in the column.

The INFs row is a count of the Inf's in the column.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>
### Examples

```r
# Attach packages
library(cvms)
library(dplyr)

df <- data.frame("a" = c("a", "a", "a", "b", "b", "b", "c", "c", "c"),
  "b" = c(0.8, 0.6, 0.3, 0.2, 0.4, 0.5, 0.8, 0.1, 0.5),
  "c" = c(0.2, 0.3, 0.4, 0.6, 0.5, 0.8, 0.1, 0.8, 0.3))

# Summarize all numeric columns
summarize_metrics(df)

# Summarize column "b"
summarize_metrics(df, cols = "b")
```

### sum_tile_settings

Create a list of settings for the sum tiles in `plot_confusion_matrix()`

**Description**

[Experimental] Creates a list of settings for plotting the column/row sums in `plot_confusion_matrix()`.

The `tc_` in the arguments refers to the **total count** tile.

NOTE: This is very experimental and will likely change.

**Usage**

```r
sum_tile_settings(
  palette = NULL,
  label = NULL,
  tile_fill = NULL,
  font_color = NULL,
  tile_border_color = NULL,
  tile_border_size = NULL,
  tile_border_linetype = NULL,
  tc_tile_fill = NULL,
  tc_font_color = NULL,
  tc_tile_border_color = NULL,
  tc_tile_border_size = NULL,
  tc_tile_border_linetype = NULL,
  intensity_by = NULL,
  intensity_lims = NULL,
  intensity_beyond_lims = NULL
)
```

**Arguments**

- **palette**: Color scheme to use for sum tiles. Should be different from the `palette` used for the regular tiles.
  Passed directly to `palette` in `ggplot2::scale_fill_distiller`. 
Try these palettes: "Greens", "Oranges", "Greys", "Purples", "Reds", and "Blues".
Alternatively, pass a named list with limits of a custom gradient as e.g. `list("low"="#e9e1fc", "high"="#BE94E6")`. These are passed to `ggplot2::scale_fill_gradient`.
Note: When `tile_fill` is specified, the `palette` is ignored.

**label**
The label to use for the sum column and the sum row.

**tc_tile_fill, tile_fill**
Specific background color for the tiles. Passed as `fill` to `ggplot2::geom_tile`.
If specified, the `palette` is ignored.

**tc_font_color, font_color**
Color of the text in the tiles with the column and row sums.

**tc_tile_border_color, tile_border_color**
Color of the tile borders. Passed as `colour` to `ggplot2::geom_tile`.

**tc_tile_border_size, tile_border_size**
Size of the tile borders. Passed as `size` to `ggplot2::geom_tile`.

**tc_tile_border_linetype, tile_border_linetype**
Linetype for the tile borders. Passed as `linetype` to `ggplot2::geom_tile`.

**intensity_by**
The measure that should control the color intensity of the tiles. Either `counts`, `normalized` or one of `log counts`, `log2 counts`, `log10 counts`, `arcsinh counts`.
For `normalized`, the color limits become 0-100 (except when `intensity_lims` are specified), why the intensities can better be compared across plots.
For the `log*` and `arcsinh` versions, the log/arcsinh transformed counts are used.
**Note:** In `log*` transformed counts, 0-counts are set to 0, why they won’t be distinguishable from 1-counts.

**intensity_lims**
A specific range of values for the color intensity of the tiles. Given as a numeric vector with c(min, max).
This allows having the same intensity scale across plots for better comparison of prediction sets.

**intensity_beyond_lims**
What to do with values beyond the `intensity_lims`. One of "truncate", "grey".

**Value**
List of settings.

**Author(s)**
Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

**See Also**
Other plotting functions: `font()`, `plot_confusion_matrix()`, `plot_metric_density()`, `plot_probabilities_ecdf()`, `plot_probabilities()`
update_hyperparameters

Check and update hyperparameters

Description

[Experimental]

1. Checks if the required hyperparameters are present and throws an error when it is not the case.
2. Inserts the missing hyperparameters with the supplied default values.

For managing hyperparameters in custom model functions for `cross_validate_fn()` or `validate_fn()`.

Usage

`update_hyperparameters(..., hyperparameters, .required = NULL)`

Arguments

- `...`  
  Default values for missing hyperparameters.
  E.g.:
  ```r
  kernel = "linear", cost = 10
  ```

- `hyperparameters`  
  List of hyperparameters as supplied to `cross_validate_fn()`. Can also be a single-row `data.frame`.

- `.required`  
  Names of required hyperparameters. If any of these are not present in the hyperparameters, an error is thrown.

Value

A named list with the updated hyperparameters.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other example functions: `model_functions()`, `predict_functions()`, `preprocess_functions()`

Examples

```r
# Attach packages
library(cvms)

# Create a list of hyperparameters
hpars <- list(
  "kernel" = "radial",
  "scale" = TRUE
)

# Update hyperparameters with defaults
```
validate

Validate regression models on a test set

Description

[Stable]

Train linear or logistic regression models on a training set and validate it by predicting a test/validation set. Returns results in a tibble for easy reporting, along with the trained models.

See validate_fn() for use with custom model functions.

Usage

validate(
train_data,
formulas,
family,
test_data = NULL,
partitions_col = ".partitions",
control = NULL,
REML = FALSE,
cutoff = 0.5,
positive = 2,
metrics = list(),
preprocessing = NULL,
err_nc = FALSE,
rm_nc = FALSE,
parallel = FALSE,
verbose = FALSE,
link = deprecated(),

def update_hyperparameters(
  cost = 10,
  kernel = "linear",
  "scale" = FALSE,
  hyperparameters = hparams
)

# 'cost' is required
# throws error
if (requireNamespace("xpectr", quietly = TRUE)){
xpectr::capture_side_effects(
  update_hyperparameters(
    kernel = "linear",
    "scale" = FALSE,
    hyperparameters = hparams,
    .required = "cost"
  )
)
}


models = deprecated(),
model_verbose = deprecated()
)

Arguments

- **train_data** data.frame.
  Can contain a grouping factor for identifying partitions - as made with `groupdata2::partition()`.
  See `partitions_col`.

- **formulas** Model formulas as strings. (Character)
  E.g. `c("y~x", "y~z")`.
  Can contain random effects.
  E.g. `c("y-x+(1|r)", "y-z+(1|r)")`.

- **family** Name of the family. (Character)
  Currently supports "gaussian" for linear regression with `lm()` / `lme4::lmer()` and "binomial" for binary classification with `glm()` / `lme4::glmer()`.
  See `cross_validate_fn()` for use with other model functions.

- **test_data** data.frame. If specifying `partitions_col`, this can be `NULL`.

- **partitions_col** Name of grouping factor for identifying partitions. (Character)
  Rows with the value 1 in `partitions_col` are used as training set and rows with the value 2 are used as test set.
  **N.B. Only used if ‘test_data’ is NULL.**

- **control** Construct control structures for mixed model fitting (with `lme4::lmer()` or `lme4::glmer()`). See `lme4::lmerControl` and `lme4::glmerControl`.
  N.B. Ignored if fitting `lm()` or `glm()` models.

- **REML** Restricted Maximum Likelihood. (Logical)

- **cutoff** Threshold for predicted classes. (Numeric)
  **N.B. Binomial models only**
  Positive Level from dependent variable to predict. Either as character (`preferable`) or level index (1 or 2 - alphabetically).
  E.g. if we have the levels "cat" and "dog" and we want "dog" to be the positive class, we can either provide "dog" or 2, as alphabetically, "dog" comes after "cat".
  **Note:** For reproducibility, it’s preferable to specify the name directly, as different locales may sort the levels differently.
  Used when calculating confusion matrix metrics and creating ROC curves.
  The Process column in the output can be used to verify this setting.
  **N.B. Only affects evaluation metrics, not the model training or returned predictions.**
  **N.B. Binomial models only.**

- **metrics** list for enabling/disabling metrics.
  E.g. `list("RMSE" = FALSE)` would remove RMSE from the results, and `list("Accuracy" = TRUE)` would add the regular Accuracy metric to the classification results. Default values (TRUE/FALSE) will be used for the remaining available metrics.
  You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why `list("all" = FALSE, "RMSE" = TRUE)` would return only the RMSE metric.
  The list can be created with `gaussian_metrics()` or `binomial_metrics()`.
  Also accepts the string "all".
validate

preprocessing  Name of preprocessing to apply.

Available preprocessings are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;standardize&quot;</td>
<td>Centers and scales the numeric predictors.</td>
</tr>
<tr>
<td>&quot;range&quot;</td>
<td>Normalizes the numeric predictors to the 0-1 range. Values outside the min/max range in the test fold are truncated to 0/1.</td>
</tr>
<tr>
<td>&quot;scale&quot;</td>
<td>Scales the numeric predictors to have a standard deviation of one.</td>
</tr>
<tr>
<td>&quot;center&quot;</td>
<td>Centers the numeric predictors to have a mean of zero.</td>
</tr>
</tbody>
</table>

The preprocessing parameters (mean, SD, etc.) are extracted from the training folds and applied to both the training folds and the test fold. They are returned in the Preprocess column for inspection.

N.B. The preprocessings should not affect the results to a noticeable degree, although "range" might due to the truncation.

err_nc  Whether to raise an error if a model does not converge. (Logical)
rm_nc  Remove non-converged models from output. (Logical)
parallel  Whether to validate the list of models in parallel. (Logical)

Remember to register a parallel backend first. E.g. with doParallel::registerDoParallel.

verbose  Whether to message process information like the number of model instances to fit and which model function was applied. (Logical)

link, models, model_verbose  Deprecated.

Details

Packages used:

Models:

Gaussian: stats::lm, lme4::lmer
Binomial: stats::glm, lme4::glmer

Results:

Shared:

AIC: stats::AIC
AICc: MuMIn::AICC
BIC: stats::BIC

Gaussian:

r2m: MuMIn::r.squaredGLMM
r2c: MuMIn::r.squaredGLMM

Binomial:

ROC and AUC: pROC::roc

Value

tibble with the results and model objects.

Shared across families:

A nested tibble with coefficients of the models from all iterations.

Count of convergence warnings. Consider discarding models that did not converge.
Count of **other warnings**. These are warnings without keywords such as "convergence".
Count of **Singular Fit messages**. See `lme4::isSingular` for more information.
Nested tibble with the **warnings and messages** caught for each model.
Specified **family**.
Nested **model** objects.
Name of dependent variable.
Names of **fixed** effects.
Names of **random** effects, if any.
Nested tibble with **preprocessing** parameters, if any.

---

**Gaussian Results:**

- RMSE, MAE, NRMSE (IQR), RRSE, RAE, RMSLE, AIC, AICc, and BIC.
- See the additional metrics (disabled by default) at `?gaussian_metrics`.
- A nested tibble with the **predictions** and targets.

---

**Binomial Results:**

Based on predictions of the test set, a confusion matrix and ROC curve are used to get the following:

- ROC:
  - AUC, Lower CI, and Upper CI.
- Confusion Matrix:
  - Balanced Accuracy, F1, Sensitivity, Specificity, Positive Predictive Value, Negative Predictive Value, Kappa, Detection Rate, Detection Prevalence, Prevalence, and MCC (Matthews correlation coefficient).
  - See the additional metrics (disabled by default) at `?binomial_metrics`.
  - Also includes:
    - A nested tibble with **predictions**, predicted classes (depends on cutoff), and the targets. Note, that the predictions are **not necessarily** of the **specified** positive class, but of the **model**'s positive class (second level of dependent variable, alphabetically).
    - The `pROC::roc` ROC curve object(s).
    - A nested tibble with the **confusion matrix/matrices**. The Pos_ columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the "positive" class. I.e. the level you wish to predict.
    - The name of the **Positive Class**.

---

**Author(s)**

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

**See Also**

Other validation functions: `cross_validate_fn()`, `cross_validate()`, `validate_fn()`
validate

Examples

# Attach packages
library(cvms)
library(groupdata2) # partition()
library(dplyr) # %>% arrange()

# Data is part of cvms
data <- participant.scores

# Set seed for reproducibility
set.seed(7)

# Partition data
# Keep as single data frame
# We could also have fed validate() separate train and test sets.
data_partitioned <- partition(
data,
p = 0.7,
cat_col = "diagnosis",
id_col = "participant",
list_out = FALSE
) %>%
arrange(.partitions)

# Validate a model
# Gaussian
validate(data_partitioned,
formulas = "score~diagnosis",
partitions_col = ".partitions",
family = "gaussian",
REML = FALSE
)

# Binomial
validate(data_partitioned,
formulas = "diagnosis~score",
partitions_col = ".partitions",
family = "binomial"
)

## Feed separate train and test sets

# Partition data to list of data frames
# The first data frame will be train (70% of the data)
# The second will be test (30% of the data)
data_partitioned <- partition(
data,
p = 0.7,
cat_col = "diagnosis",
id_col = "participant",
list_out = TRUE
)

train_data <- data_partitioned[[1]]
test_data <- data_partitioned[[2]]
# Validate a model

# Gaussian
validate(
  train_data,
  test_data = test_data,
  formulas = "score~diagnosis",
  family = "gaussian",
  REML = FALSE
)

validate_fn

Validates a custom model function on a test set

**Description**

**[Experimental]**

Fit your model function on a training set and validate it by predicting a test/validation set. Validate different hyperparameter combinations and formulas at once. Preprocess the train/test split. Returns results and fitted models in a tibble for easy reporting and further analysis.

Compared to `validate()`, this function allows you supply a custom model function, a predict function, a preprocess function and the hyperparameter values to validate.

Supports regression and classification (binary and multiclass). See `type`.

Note that some metrics may not be computable for some types of model objects.

**Usage**

```r
validate_fn(
  train_data,
  formulas,
  type,
  model_fn,
  predict_fn,
  test_data = NULL,
  preprocess_fn = NULL,
  preprocess_once = FALSE,
  hyperparameters = NULL,
  partitions_col = ".partitions",
  cutoff = 0.5,
  positive = 2,
  metrics = list(),
  rm_nc = FALSE,
  parallel = FALSE,
  verbose = TRUE
)
```
validate_fn

Arguments

**train_data**  data.frame.
Can contain a grouping factor for identifying partitions - as made with `groupdata2::partition()`.
See `partitions_col`.

**formulas**  Model formulas as strings. (Character)
Will be converted to `formula` objects before being passed to `model_fn`.
E.g. c("y~x", "y~z").
Can contain random effects.
E.g. c("y~x+(1|r)", "y~z+(1|r)").

**type**  Type of evaluation to perform:
"gaussian" for regression (like linear regression).
"binomial" for binary classification.
"multinomial" for multiclass classification.

**model_fn**  Model function that returns a fitted model object. Will usually wrap an existing model function like `e1071::svm` or `nnet::multinom`.
Must have the following function arguments:
function(train_data, formula, hyperparameters)

**predict_fn**  Function for predicting the targets in the test folds/sets using the fitted model object. Will usually wrap `stats::predict()`, but doesn’t have to.
Must have the following function arguments:
function(test_data, model, formula, hyperparameters, train_data)
Must return predictions in the following formats, depending on `type`:

- **Binomial**: vector or one-column matrix / data.frame with probabilities (0-1) of the second class, alphabetically. E.g.:
c(0.3, 0.5, 0.1, 0.5)
N.B. When unsure whether a model type produces probabilities based off the alphabetic order of your classes, using 0 and 1 as classes in the dependent variable instead of the class names should increase the chance of getting probabilities of the right class.

- **Gaussian**: vector or one-column matrix / data.frame with the predicted value. E.g.:
c(3.7, 0.9, 1.2, 7.3)

- **Multinomial**: data.frame with one column per class containing probabilities of the class. Column names should be identical to how the class names are written in the target column. E.g.:

<table>
<thead>
<tr>
<th></th>
<th>class_1</th>
<th>class_2</th>
<th>class_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.269</td>
<td>0.528</td>
<td>0.203</td>
</tr>
<tr>
<td>2</td>
<td>0.368</td>
<td>0.322</td>
<td>0.310</td>
</tr>
<tr>
<td>3</td>
<td>0.375</td>
<td>0.371</td>
<td>0.254</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**test_data**  data.frame. If specifying `partitions_col`, this can be NULL.

**preprocess_fn**  Function for preprocessing the training and test sets.
Can, for instance, be used to standardize both the training and test sets with the scaling and centering parameters from the training set.
validate_fn

Must have the following function arguments:

```r
function(train_data, test_data,
    formula, hyperparameters)
```

Must return a list with the preprocessed `train_data` and `test_data`. It may also contain a tibble with the parameters used in preprocessing:

```r
list("train" = train_data,
    "test" = test_data,
    "parameters" = preprocess_parameters)
```

Additional elements in the returned list will be ignored.

The optional parameters tibble will be included in the output. It could have the following format:

<table>
<thead>
<tr>
<th>Measure</th>
<th>var_1</th>
<th>var_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>37.921</td>
<td>88.231</td>
</tr>
<tr>
<td>SD</td>
<td>12.4</td>
<td>5.986</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

N.B. When `preprocess_once` is FALSE, the current formula and hyperparameters will be provided. Otherwise, these arguments will be NULL.

### preprocess_once

Whether to apply the preprocessing once (ignoring the formula and hyperparameters arguments in `preprocess_fn`) or for every model separately. (Logical)

When preprocessing does not depend on the current formula or hyperparameters, we can do the preprocessing of each train/test split once, to save time. This may require holding a lot more data in memory though, why it is not the default setting.

### hyperparameters

Either a named list with hyperparameter values to combine in a grid or a data.frame with one row per hyperparameter combination.

- **Named list for grid search**: Add ".n" to sample the combinations. Can be the number of combinations to use, or a percentage between 0 and 1.

  E.g.

  ```r
  list(".n" = 10, # sample 10 combinations
       "lrn_rate" = c(0.1, 0.01, 0.001),
       "h_layers" = c(10, 100, 1000),
       "drop_out" = runif(5, 0.3, 0.7))
  ```

- **data.frame with specific hyperparameter combinations**: One row per combination to test.

  E.g.

  ```r
  lrn_rate  h_layers  drop_out
  0.1       10        0.65
  0.1       1000      0.65
  0.01      1000      0.63
  ...      ...       ...
  ```

### partitions_col

Name of grouping factor for identifying partitions. (Character)

Rows with the value 1 in `partitions_col` are used as training set and rows with the value 2 are used as test set.

N.B. **Only used if `test_data` is NULL.**
cutoff: Threshold for predicted classes. (Numeric)

**N.B. Binomial models only**

positive: Level from dependent variable to predict. Either as character (*preferable*) or level index (1 or 2 - alphabetically).

E.g. if we have the levels "cat" and "dog" and we want "dog" to be the positive class, we can either provide "dog" or 2, as alphabetically, "dog" comes after "cat".

**Note:** For *reproducibility*, it's preferable to *specify the name directly*, as different locales may sort the levels differently.

Used when calculating confusion matrix metrics and creating ROC curves.

The `Process` column in the output can be used to verify this setting.

**N.B.** Only affects evaluation metrics, not the model training or returned predictions.

**N.B. Binomial models only.**

metrics: list for enabling/disabling metrics.

E.g. `list("RMSE" = FALSE)` would remove RMSE from the regression results, and `list("Accuracy" = TRUE)` would add the regular Accuracy metric to the classification results. Default values (TRUE/FALSE) will be used for the remaining available metrics.

You can enable/disable all metrics at once by including "all" = TRUE/FALSE in the list. This is done prior to enabling/disabling individual metrics, why f.i. `list("all" = FALSE, "RMSE" = TRUE)` would return only the RMSE metric.

The list can be created with `gaussian_metrics()`, `binomial_metrics()`, or `multinomial_metrics()`.

Also accepts the string "all".

rm_nc: Remove non-converged models from output. (Logical)

parallel: Whether to cross-validate the list of models in parallel. (Logical)

Remember to register a parallel backend first. E.g. with `doParallel::registerDoParallel`.

verbose: Whether to message process information like the number of model instances to fit. (Logical)

**Details**

Packages used:

**Results:**

*Shared:*

- **AIC**: `stats::AIC`
- **AICC**: `MuMIn::AICc`
- **BIC**: `stats::BIC`

*Gaussian:*

- **r2m**: `MuMIn::r.squaredGLMM`
- **r2c**: `MuMIn::r.squaredGLMM`

*Binomial and Multinomial:*

ROC and related metrics:

- **Binomial**: `pROC::roc`
- **Multinomial**: `pROC::multiclass.roc`
**Value**

tibble with the results and model objects.

**Shared across families:**
A nested tibble with coefficients of the models. The coefficients are extracted from the model object with `parameters::model_parameters()` or `coef()` (with some restrictions on the output). If these attempts fail, a default coefficients tibble filled with NAs is returned.

Nested tibble with the used preprocessing parameters, if a passed `preprocess_fn` returns the parameters in a tibble.

Count of convergence warnings, using a limited set of keywords (e.g. 'convergence'). If a convergence warning does not contain one of these keywords, it will be counted with other warnings. Consider discarding models that did not converge on all iterations. Note: you might still see results, but these should be taken with a grain of salt!

Nested tibble with the warnings and messages caught for each model.

Specified family.

Nested model objects.

Name of dependent variable.

Names of fixed effects.

Names of random effects, if any.

---

**Gaussian Results:**

RMSE, MAE, NRMSE(IQR), RRSE, RAE, and RMSLE.

See the additional metrics (disabled by default) at `?gaussian_metrics`.

A nested tibble with the predictions and targets.

---

**Binomial Results:**

Based on predictions of the test set, a confusion matrix and a ROC curve are created to get the following:

ROC:

- AUC, Lower CI, and Upper CI

Confusion Matrix:

Balanced Accuracy, F1, Sensitivity, Specificity, Positive Predictive Value, Negative Predictive Value, Kappa, Detection Rate, Detection Prevalence, Prevalence, and MCC (Matthews correlation coefficient).

See the additional metrics (disabled by default) at `?binomial_metrics`.

Also includes:

- A nested tibble with predictions, predicted classes (depends on cutoff), and the targets. Note, that the predictions are not necessarily of the specified positive class, but of the model’s positive class (second level of dependent variable, alphabetically).

The pROC::roc ROC curve object(s).

A nested tibble with the confusion matrix/matrices. The Pos_ columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the "positive" class. I.e. the level you wish to predict.

The name of the Positive Class.
Multinomial Results:

For each class, a one-vs-all binomial evaluation is performed. This creates a Class Level Results tibble containing the same metrics as the binomial results described above (excluding MCC, AUC, Lower CI and Upper CI), along with a count of the class in the target column (Support). These metrics are used to calculate the macro metrics. The nested class level results tibble is also included in the output tibble, and could be reported along with the macro and overall metrics.

The output tibble contains the macro and overall metrics. The metrics that share their name with the metrics in the nested class level results tibble are averages of those metrics (note: does not remove NAs before averaging). In addition to these, it also includes the Overall Accuracy and the multiclass MCC.

Other available metrics (disabled by default, see metrics): Accuracy, multiclass AUC, Weighted Balanced Accuracy, Weighted Accuracy, Weighted F1, Weighted Sensitivity, Weighted Specificity, Weighted Pos Pred Value, Weighted Neg Pred Value, Weighted Kappa, Weighted Detection Rate, Weighted Detection Prevalence, and Weighted Prevalence.

Note that the "Weighted" average metrics are weighted by the Support.

Also includes:
A nested tibble with the predictions, predicted classes, and targets.
A list of ROC curve objects when AUC is enabled.
A nested tibble with the multiclass Confusion Matrix.

Class Level Results

Besides the binomial evaluation metrics and the Support, the nested class level results tibble also contains a nested tibble with the Confusion Matrix from the one-vs-all evaluation. The Pos_ columns tells you whether a row is a True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN), depending on which level is the "positive" class. In our case, 1 is the current class and 0 represents all the other classes together.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other validation functions: cross_validate_fn(), cross_validate(), validate()

Examples

# Attach packages
library(cvms)
library(groupdata2) # fold()
library(dplyr) # %>% arrange() mutate()

# Note: More examples of custom functions can be found at:
# model_fn: model_functions()
# predict_fn: predict_functions()
# preprocess_fn: preprocess_functions()

# Data is part of cvms
data <- participant.scores

# Set seed for reproducibility
set.seed(7)

# Fold data
data <- partition(
  data,
  p = 0.8,
  cat_col = "diagnosis",
  id_col = "participant",
  list_out = FALSE
) %>%
  mutate(diagnosis = as.factor(diagnosis)) %>%
  arrange(.partitions)

# Formulas to validate
formula_gaussian <- "score ~ diagnosis"
formula_binomial <- "diagnosis ~ score"

# Gaussian

# Create model function that returns a fitted model object
lm_model_fn <- function(train_data, formula, hyperparameters) {
  lm(formula = formula, data = train_data)
}

# Create predict function that returns the predictions
lm_predict_fn <- function(test_data, model, formula, hyperparameters, train_data) {
  stats::predict(
    object = model,
    newdata = test_data,
    type = "response",
    allow.new.levels = TRUE
  )
}

# Validate the model function
v <- validate_fn(
  data,
  formulas = formula_gaussian,
  type = "gaussian",
  model_fn = lm_model_fn,
  predict_fn = lm_predict_fn,
  partitions_col = ".partitions"
)

v

# Extract model object
v$Model[[1]]

# Binomial

# Create model function that returns a fitted model object

glm_model_fn <- function(train_data, formula, hyperparameters) {
  glm(formula = formula, data = train_data, family = "binomial")
}

# Create predict function that returns the predictions

glm_predict_fn <- function(test_data, model, formula, hyperparameters, train_data) {
  stats::predict(
    object = model,
    newdata = test_data,
    type = "response",
    allow.new.levels = TRUE
  )
}

# Validate the model function

validate_fn(
  data,
  formulas = formula_binomial,
  type = "binomial",
  model_fn = glm_model_fn,
  predict_fn = glm_predict_fn,
  partitions_col = ".partitions"
)

# Support Vector Machine (svm)
# Only run if the `e1071` package is installed
if (requireNamespace("e1071", quietly = TRUE)) {

  # Create model function that returns a fitted model object
  # We use the hyperparameters arg to pass in the kernel and cost values
  # These will usually have been found with cross_validate_fn()

svm_model_fn <- function(train_data, formula, hyperparameters) {

    # Expected hyperparameters:
    # - kernel
    # - cost

    if (!"kernel" %in% names(hyperparameters))
      stop("'hyperparameters' must include 'kernel'")

    if (!"cost" %in% names(hyperparameters))
      stop("'hyperparameters' must include 'cost'")

    e1071::svm(
      formula = formula,
      data = train_data,
      kernel = hyperparameters["Kernel"],
      cost = hyperparameters["cost"],
      scale = FALSE,
      type = "C-classification",
      probability = TRUE
    )
  }
}
# Create predict function that returns the predictions
svm_predict_fn <- function(test_data, model, formula, hyperparameters, train_data) {
    predictions <- stats::predict(
        object = model,
        newdata = test_data,
        allow.new.levels = TRUE,
        probability = TRUE
    )

    # Extract probabilities
    probabilities <- dplyr::as_tibble(
        attr(predictions, "probabilities")
    )

    # Return second column
    probabilities[[2]]
}

# Specify hyperparameters to use
# We found these in the examples in ?cross_validate_fn()
svm_hparams <- list(
    "kernel" = "linear",
    "cost" = 10
)

# Validate the model function
validate_fn(
    data,
    formulas = formula_binomial,
    type = "binomial",
    model_fn = svm_model_fn,
    predict_fn = svm_predict_fn,
    hyperparameters = svm_hparams,
    partitions_col = ".partitions"
)
# closes `e1071` package check

---

**Wines**

<table>
<thead>
<tr>
<th><strong>Wine varieties</strong></th>
</tr>
</thead>
</table>

**Description**

A list of wine varieties in an approximately Zipfian distribution, ordered by descending frequencies.

**Format**

A data.frame with 368 rows and 1 variable:

**Variety** Wine variety, 10 levels
Details

Based on the wine-reviews (v4) kaggle dataset by Zack Thoult: https://www.kaggle.com/zynicide/wine-reviews

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>
Index

* baseline functions
  baseline, 3
  baseline_binomial, 9
  baseline_gaussian, 12
  baseline_multinomial, 15
* data
  compatible.formula.terms, 23
  musicians, 65
  participant.scores, 65
  precomputed.formulas, 73
  predicted.musicians, 74
  wines, 99
* evaluation functions
  binomial_metrics, 19
  confusion_matrix, 21
  evaluate, 42
  evaluate_residuals, 49
  gaussian_metrics, 52
  multinomial_metrics, 62
* example functions
  model_functions, 54
  predict_functions, 75
  preprocess_functions, 76
  update_hyperparameters, 85
* plotting functions
  font, 51
  plot_confusion_matrix, 66
  plot_metric_density, 72
  sum_tile_settings, 83
* validation functions
  cross_validate, 27
  cross_validate_fn, 32
  validate, 86
  validate_fn, 91
 .Machine$double.eps, 56
 ?binomial_metrics, 30, 37, 89, 95
 ?gaussian_metrics, 6, 14, 30, 37, 46, 89, 95

as.character.process_info_binomial
  (process_info_binomial), 77
as.character.process_info_gaussian
  (process_info_binomial), 77
as.character.process_info_multinomial
  (process_info_binomial), 77
as.formula(), 81
baseline, 3, 11, 14, 18
baseline(), 42, 72
baseline_binomial, 8, 9, 14, 18
baseline_binomial(), 3
baseline_gaussian, 8, 11, 12, 18
baseline_gaussian(), 3
baseline_multinomial, 8, 11, 14, 15
baseline_multinomial(), 3
binomial_metrics, 19, 27, 47, 51, 54, 64
binomial_metrics(), 4, 10, 17, 24, 29, 35, 45, 87, 94
coeff(), 36, 95
combine_predictors, 21, 23, 74
compatible.formula.terms, 23
confusion_matrix, 21, 24, 47, 51, 54, 64
confusion_matrix(), 67
cross_validate, 27, 38, 89, 96
cross_validate(), 32, 42, 79, 80
cross_validate_fn, 31, 32, 89, 96
cross_validate_fn(), 27, 28, 42, 54, 75, 76, 85, 87
cvms, 42
dplyr::group_by(), 24, 50, 56
e1071::naiveBayes(), 55, 76
e1071::svm, 33, 92
e1071::svm(), 55, 76
evaluate, 21, 27, 42, 51, 54, 64
evaluate(), 24, 42, 50, 66, 67, 80
evaluate_residuals, 21, 27, 47, 49, 54, 64
font, 51, 69, 73, 84
font(), 69
formula, 33, 92
gaussian_metrics, 21, 27, 47, 51, 52, 64
gaussian_metrics(), 4, 13, 29, 35, 45, 50, 87, 94
generate_formulas(combine_predictors), 21
ggimage::geom_icon(), 69
hardest (most challenging), 55

lm(), 28, 87
lme4::glmer, 29, 88
lme4::glmer(), 28, 55, 76, 87
lme4::glmerControl, 28, 87
lme4::isSingular, 30, 89
lme4::lmer, 5, 14, 29, 88
lme4::lmerControl, 28, 87
locales, 4, 10, 24, 28, 35, 45, 87, 94

model_functions, 54, 76, 77, 85
most_challenging, 55
multiclass_probability_tibble, 60
multiclass_probability_tibble(), 5, 16
multinomial_metrics, 21, 27, 47, 51, 54, 62
multinomial_metrics(), 4, 16, 24, 35, 45, 94
MuMIn::AICc, 5, 14, 29, 36, 88, 94
MuMIn::r.squaredGLMM, 5, 14, 30, 36, 88, 94
musicians, 65, 74

nnet::multinom, 33, 92
nnet::multinom(), 76
nnet::nnet(), 76

parameters::model_parameters(), 36, 95
participant.scores, 65
plot_confusion_matrix, 52, 66, 73, 84
plot_confusion_matrix(), 83
plot_metric_density, 52, 69, 72, 84
plot_probabilities, 52, 69, 73, 84
plot_probabilities_ecdf, 52, 69, 73, 84
precomputed.formulas, 73
predict_functions, 55, 75, 77, 85
predicted.musicians, 74
preprocess_functions, 55, 76, 76, 85
preprocess_functions(), 81
print.process_info_binomial
   (process_info_binomial), 77
print.process_info_gaussian
   (process_info_binomial), 77
print.process_info_multinomial
   (process_info_multinomial), 77
randomForest::randomForest(), 76
recipes, 76
recipes::recipe(), 81
reconstruct_formulas, 79
select_definitions, 80
select_metrics, 80
simplify_formula, 81
stats::AIC, 5, 14, 29, 36, 88, 94
stats::BIC, 5, 14, 29, 36, 88, 94
stats::glm, 29, 88
stats::glm(), 55, 76
stats::lm, 5, 14, 29, 88
stats::lm(), 55, 76
stats::predict(), 33, 92
sum_tile_settings, 52, 69, 73, 83
sum_tile_settings(), 69
summarize_metrics, 82
tidyr::unnest, 7, 17

update_hyperparameters, 55, 76, 77, 85
validate, 31, 38, 86, 96
validate(), 42, 79, 91
validate_fn, 31, 38, 89, 91
validate_fn(), 42, 76, 85, 86
wines, 99