Package 'healthcareai'

August 6, 2020

Type Package

Title Tools for Healthcare Machine Learning

Version 2.5.0

Date 2020-08-04

Description A machine learning toolbox tailored to healthcare data.

License MIT + file LICENSE

LazyData TRUE

Depends R (>= 3.6), methods

Imports caret (>= 6.0.81), cowplot, data.table, dplyr (>= 1.0.0), e1071, generics, ggplot2, glmnet, lubridate, MLmetrics, purrr, ranger (>= 0.8.0), recipes (>= 0.1.3.9002), rlang, ROCR, stringr, tibble (>= 3.0.0), tidyr, xgboost

RoxygenNote 7.1.1

Suggests covr, DBI, dbplyr, lintr, odbc, testthat

URL http://docs.healthcare.ai

BugReports https://github.com/HealthCatalyst/healthcareai-r/issues

Encoding UTF-8

NeedsCompilation no

Author Levi Thatcher [aut], Michael Levy [aut], Mike Mastanduno [aut, cre], Taylor Larsen [aut], Taylor Miller [aut], Rex Sumsion [aut]

Maintainer Mike Mastanduno <michael.mastanduno@healthcatalyst.com>

Repository CRAN

Date/Publication 2020-08-05 23:30:02 UTC

R topics documented:

`add_best_levels` *Build efficient features from high-cardinality, multiple-membership factors*

Description

In healthcare, we are often faced with high cardinality variables, where each observation may have zero, one, or more levels, e.g. medications for a model at the patient grain. In these cases, creating a feature variable for each level (each medication) as in one-hot encoding can be prohibitively computationally intensive and can hurt performance by diminishing the signal-to-noise ratio. `get_best_levels` identifies a subset of categories that are likely to be valuable features, and `add_best_levels` adds them to a model data frame.

`get_best_levels` finds levels of `groups` that are likely to be useful predictors in `d` and returns them as a character vector. `add_best_levels` does the same and adds them, pivoted, to `d`. The function attempts to find both positive and negative predictors of `outcome`.

`add_best_levels` stores the identified best levels and passes them through model training so that in deployment, the same columns created in training are again created (see the final example).

`add_best_levels` accepts arguments to `pivot` so that values associated with the levels (e.g. doses of medications) can be used in the new features. However, note that these are not used in determining the best levels. I.e. `get_best_levels` determines which levels are likely to be good predictors looking only at outcomes where the levels are present or absent; it does not use `fill` or `fun` in this determination. See `details` for more info about how levels are selected.

Usage

```
add_best_levels(
  d,
  longsheet,
  id,
  groups,
  outcome,
  n_levels = 100,
```

```
  min_obs = 1,
  positive_class = "Y",
  cohesion_weight = 2,
  levels = NULL,
  fill,
  fun = sum,
  missing_fill = NA
)

get_best_levels(
  d,
  longsheet,
  id,
  groups,
  outcome,
  n_levels = 100,
  min_obs = 1,
  positive_class = "Y",
  cohesion_weight = 2
)
```

Arguments

d	Data frame to use in models, at desired grain. Has id and outcome
longsheet	Data frame containing multiple observations per grain. Has id and groups
id	Name of identifier column, unquoted. Must be present and identical in both tables
groups	Name of grouping column, unquoted
outcome	Name of outcome column, unquoted
n_levels	Number of levels to return, default = 100. An attempt is made to return half levels positively associated with the outcome and half negatively. If n_levels is greater than the number present, all levels will be returned
min_obs	Minimum number of observations a level must be found in in order to be considered. Defaults to one, but larger values are often useful because a level present in only a few observation will rarely be a useful.
positive_class	If classification model, the positive class of the outcome, default = "Y"; ignored if regression
cohesion_weight	For classification problems only, how much to value a level being consistently associated with an outcome relative to its being present in many observations. Default = 2; equal weight is 1. Note that this is a parameter that could potentially be tuned over.
levels	Use this argument when add_best_levels was used in training and you want to add the same columns for deployment. You can pass the model trained on the data frame from `add_best_levels`, the data frame from `add_best_levels`, or a character vector of levels to add.

`fill`	Passed to `pivot`. Column to be used to fill the values of cells in the output, perhaps after aggregation by `fun`. If `fill` is not provided, counts will be used, as though a fill column of 1s had been provided.
`fun`	Passed to `pivot`. Function for aggregation, defaults to `sum`. Custom functions can be used with the same syntax as the apply family of functions, e.g. `fun = function(x) some_function(another_fun(x))`.
`missing_fill`	Passed to `pivot`. Value to fill for combinations of grain and spread that are not present. Defaults to NA, but 0 may be useful as well.

Details

Here is how `get_best_levels` determines the levels of `groups` that are likely to be good predictors.

- For regression: For each group, the difference of the group-mean from the grand-mean is divided by the standard deviation of the group as a sample (i.e. centered_mean(group) / sqrt(var(group) / n(group))), and the groups with the largest absolute values of that statistic are retained.
- For classification: For each group, two "log-loss-like" statistics are calculated. One is the log of the fraction of observations in which the group does not appear, which captures how ubiquitous the group is: more common groups are more useful as predictors. The other captures how far the group is from being always associated with the same outcome: groups that are consistently assoicated with either outcome are more useful as predictors. This is calculated as the log of the proportion of outcomes that are not all the same outcome (e.g. if 4/5 observations are positive class, this statistic is log(.2)). This value is then raised to the `cohesion_weight` power. To ensure retainment of both positive- and negative-predictors, the all-same-outcome that is used as the comparison is determined by which side of the median proportion of positive_class the group falls on.

Value

For `add_best_levels`, d with new columns for the best levels added and best_levels attribute containing a named list of levels added. For `get_best_levels`, a character vector of the best levels.

See Also

`pivot`

Examples

```
set.seed(45796)

# We have two tables we want to use in our models:
# - df is the model table. It has the outcomes (survived), and we want one
#   prediction for each row in df
# - meds has detailed information on each row (patient) in df. Each patient
#   may have zero, one, or more observations (drugs) in meds, and meds may
#   have associated values (doses).

df <- tibble::tibble(
  patient = paste0("Z", sample(10, 5)),
```

```
  age = sample(20:80, 5),
  survived = sample(c("N", "Y"), 5, replace = TRUE, prob = c(1, 2))
)

meds <- tibble::tibble(
  patient = sample(df$patient, 10, replace = TRUE),
  drug = sample(c("Quinapril", "Vancomycin", "Ibuprofen",
                  "Paclitaxel", "Epinephrine", "Dexamethasone"),
                10, replace = TRUE),
  dose = sample(c(100, 250), 10, replace = TRUE)
)

# Identify three drugs likely to be good predictors of survival

get_best_levels(d = df,
                longsheet = meds,
                id = patient,
                groups = drug,
                outcome = survived,
                n_levels = 3)

# Identify four drugs likely to make good features and add them to df.
# The "fill", "fun", and "missing_fill" arguments are passed to
# `pivot`, which allows us to use the total doses of each drug given to the
# patient as our new features

new_df <- add_best_levels(d = df,
                          longsheet = meds,
                          id = patient,
                          groups = drug,
                          outcome = survived,
                          n_levels = 4,
                          fill = dose,
                          fun = sum,
                          missing_fill = 0)
new_df

# The names of the medications that were added to df in new_df are stored in the
# best_levels attribute of new_df so that the same columns can be added in
# deployment. This is useful because you need to have the same columns to make
# predictions as you had in model training. When you are ready to add levels to
# a deployment data frame, you can pass to the "levels" argument of
# add_best_levels either the models trained on new_df, new_df itself, or the
# character vector of levels to add.

deployment_df <- tibble::tibble(
  patient = "p6",
  age = 30
)
deployment_meds <- tibble::tibble(
  patient = rep("p6", 2),
  drug = rep("Vancomycin", 2),
  dose = c(100, 250)
```

```
)

# Now, even though Vancomycin is the only drug that appears in
# deployment_meds, because we pass new_df to "levels", we get all the columns
# needed to make predictions on a model trained on new_df

add_best_levels(d = deployment_df,
                longsheet = deployment_meds,
                id = patient,
                groups = drug,
                levels = new_df,
                fill = dose,
                missing_fill = 0)
```

`add_SAM_utility_cols` *Add SAM utility columns to table*

Description

When working in a Health Catalyst Source Area Mart (SAM), utility columns are added automatically when running a non-R binding

Usage

```
add_SAM_utility_cols(d)
```

Arguments

d	A dataframe

Value

A dataframe with three additional columns

Examples

```
d <- data.frame(a = c(1,2,NA,NA),
                b = c(100,300,200,150))
d <- add_SAM_utility_cols(d)
```

`as.model_list` *Make models into model_list object*

Description

Make models into model_list object

Usage

```
as.model_list(
  ...,
  listed_models = NULL,
  target = ".outcome",
  model_class,
  tuned = TRUE,
  recipe = NULL,
  positive_class = NULL,
  model_name = NULL,
  best_levels = NULL,
  original_data_str,
  versions
)
```

Arguments

`...`	`caret`-trained models to put into a model list
`listed_models`	Use this if your models are already in a list
`target`	Quoted name of response variable
`model_class`	"classification" or "regression". Will be determined if not provided
`tuned`	Logical; if FALSE, will have super-class untuned_models
`recipe`	recipe object from prep+_data, or NULL if the data didn't go through prep_data
`positive_class`	If classification, the positive outcome class, otherwise NULL
`model_name`	Quoted, name of the model. Defaults to the name of the outcome variable.
`best_levels`	best_levels list as attached to data frames from add_best_levels
`original_data_str`	zero-row data frame with names and classes of all columns except the outcome as they came into either the model training function such as tune_models or prep_data
`versions`	A list containing the following environmental variables from model training: r_version, hcai_version, and other_packages (a tibble). If not provided, will be extracted from the current session. See healthcareai:::attach_session_info for details

Value

A model_list with child class type_list

`build_connection_string` *Build a connection string for use with MSSQL and dbConnect*

Description

Handy utility to build a connection string to pass into `DBI::dbConnect`. Accepts trusted connections or username/password.

Usage

```
build_connection_string(
  server,
  driver = "SQL Server",
  database,
  trusted = TRUE,
  user_id,
  password
)
```

Arguments

`server`	A string, quoted, required. The name of the server you are trying to connect to.
`driver`	A string, quoted, optional. Defaults to "SQL Server", but use any driver you like.
`database`	A string, quoted, optional. If provided, connection string will include a specific database. If NA (default), it will connect to master and you'll have to specify the database when running a query.
`trusted`	Logical, optional, defaults to TRUE. If FALSE, you must use a user_id and password.
`user_id`	A string, quoted, optional. Don't include if using trusted.
`password`	A string, quoted, optional. Don't include if using trusted.

Value

A connection string

See Also

`db_read`

Examples

```
## Not run:
my_con <- build_connection_string(server = "localhost")
con <- DBI::dbConnect(odbc::odbc(), .connection_string = my_con)

# with username and password
my_con <- build_connection_string(server = "localhost",
                                  user_id = "jules.winnfield",
                                  password = "pathoftherighteous")
con <- DBI::dbConnect(odbc::odbc(), .connection_string = my_con)

## End(Not run)
```

`catalyst_test_deploy_in_prod` *Defunct*

Description

Defunct

Usage

```
catalyst_test_deploy_in_prod(...)
```

Arguments

`...`	Defunct

`control_chart` *Create a control chart*

Description

Create a control chart, aka Shewhart chart: https://en.wikipedia.org/wiki/Control_chart.

Usage

```
control_chart(
  d,
  measure,
  x,
  group1,
  group2,
  center_line = mean,
  sigmas = 3,
```

```
  title = NULL,
  catpion = NULL,
  font_size = 11,
  print = TRUE
)
```

Arguments

d	data frame or a path to a csv file that will be read in
measure	variable of interest mapped to y-axis (quoted, ie as a string)
x	variable to go on the x-axis, often a time variable. If unspecified row indices will be used (quoted)
group1	Optional grouping variable to be panelled horizontally (quoted)
group2	Optional grouping variable to be panelled vertically (quoted)
center_line	Function used to calculate central tendency. Defaults to mean
sigmas	Number of standard deviations above and below the central tendency to call a point influenced by "special cause variation." Defaults to 3
title	Title in upper-left
catpion	Caption in lower-right
font_size	Base font size; text elements will be scaled to this
print	Print the plot? Default = TRUE. Set to FALSE if you want to assign the plot to a variable for further modification, as in the last example.

Value

Generally called for the side effect of printing the control chart. Invisibly, returns a ggplot object for further customization.

Examples

```
d <-
  tibble::tibble(
    day = sample(c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday"),
                 100, TRUE),
    person = sample(c("Tom", "Jane", "Alex"), 100, TRUE),
    count = rbinom(100, 20, ifelse(day == "Friday", .5, .2)),
    date = Sys.Date() - sample.int(100))

# Minimal arguments are the data and the column to put on the y-axis.
# If x is not provided, observations will be plotted in order of the rows

control_chart(d, "count")

# Specify categorical variables for group1 and/or group2 to get a separate
# panel for each category

control_chart(d, "count", group1 = "day", group2 = "person")
```

```
# In addition to printing or writing the plot to file, control_chart
# returns the plot as a ggplot2 obejct, which you can then further customize

library(ggplot2)
my_chart <- control_chart(d, "count", "date")
my_chart +
  ylab("Number of Adverse Events") +
  scale_x_date(name = "Week of ... ", date_breaks = "week") +
  theme(axis.text.x = element_text(angle = -90, vjust = 0.5, hjust=1))
```

`convert_date_cols` *Convert character date columns to dates and times*

Description

This function is called in `prep_data` and so it shouldn't usually need to be called directly. It tries to convert columns ending in "DTS" to type Date or DateTime (POSIXt). It makes a best guess at the format and return a more standard one if possible.

Usage

```
convert_date_cols(d)
```

Arguments

d	A dataframe or tibble containing data to try to convert to dates.

Value

A tibble containing the converted date columns. If no columns needed conversion, the original data will be returned.

Examples

```
d <- tibble::tibble(a_DTS = c("2018-3-25", "2018-3-25"),
                    b_nums = c(2, 4),
                    c_DTS = c("03-01-2018", "03-07-2018"),
                    d_chars = c("a", "b"),
                    e_date = lubridate::mdy(c("3-25-2018", "3-25-2018")))
convert_date_cols(d)
```

countMissingData *Function to find proportion of NAs in each column of a dataframe or matrix*

Description

DEPRICATED. Use `missingness` instead.

Usage

```
countMissingData(x, userNAs = NULL)
```

Arguments

x	A data frame or matrix
userNAs	A vector of user defined NA values.

db_read *Read from a SQL Server database table*

Description

Use a database connection to read from an existing SQL Server table with a SQL query.

Usage

```
db_read(con, query, pull_into_memory = TRUE)
```

Arguments

con	An odbc database connection. Can be made using `build_connection_string`. Required.
query	A string, quoted, required. This sql query will be executed against the database you are connected to.
pull_into_memory	Logical, optional, defaults to TRUE. If FALSE, `db_read` will create a reference to the queried data rather than pulling into memory. Set to FALSE for very large tables.

Details

Use `pull_into_memory` when working with large tables. Rather than returning the data into memory, this function will return a reference to the specified query. It will be executed only when needed, in a "lazy" style. Or, you can execute using the `collect()` function.

Value

A tibble of data or reference to the table.

See Also

build_connection_string importFrom dbplyr as.sql

Examples

```
## Not run:
my_con <- build_connection_string(server = "HPHI-EDWDEV")
con <- DBI::dbConnect(odbc::odbc(), .connection_string = my_con)
d <- db_read(con,
             "SELECT TOP 10 * FROM [Shared].[Cost].[FacilityAccountCost]")

# Get a reference and collect later
ref <- db_read(con,
               "SELECT TOP 10 * FROM [Shared].[Cost].[FacilityAccountCost]",
               pull_into_memory = FALSE)
d <- collect(ref)

## End(Not run)
```

---

evaluate *Get model performance metrics*

Description

Get model performance metrics

Usage

```
evaluate(x, ...)

## S3 method for class 'predicted_df'
evaluate(x, na.rm = FALSE, ...)

## S3 method for class 'model_list'
evaluate(x, all_models = FALSE, ...)
```

Arguments

x	Object to be evaluted
...	Not used
na.rm	Logical. If FALSE (default) performance metrics will be NA if any rows are missing an outcome value. If TRUE, performance will be evaluted on the rows that have an outcome value. Only used when evaluating a prediction data frame.

`all_models` Logical. If FALSE (default), a numeric vector giving performance metrics for the best-performing model is returned. If TRUE, a data frame with performance metrics for all trained models is returned. Only used when evaluating a model_list.

Details

This function gets model performance from a model_list object that comes from `machine_learn`, `tune_models`, `flash_models`, or a data frame of predictions from `predict.model_list`. For the latter, the data passed to `predict.model_list` must contain observed outcomes. If you have predictions and outcomes in a different format, see `evaluate_classification` or `evaluate_regression` instead.

You may notice that `evaluate(models)` and `evaluate(predict(models))` return slightly different performance metrics, even though they are being calculated on the same (out-of-fold) predictions. This is because metrics in training (returned from `evaluate(models)`) are calculated within each cross-validation fold and then averaged, while metrics calculated on the prediction data frame (`evaluate(predict(models))`) are calculated once on all observations.

Value

Either a numeric vector or a data frame depending on the value of all_models

Examples

```
models <- machine_learn(pima_diabetes[1:40, ],
                        patient_id,
                        outcome = diabetes,
                        models = c("XGB", "RF"),
                        tune = FALSE,
                        n_folds = 3)

# By default, evaluate returns performance of only the best model
evaluate(models)

# Set all_models = TRUE to see the performance of all trained models
evaluate(models, all_models = TRUE)

# Can also get performance on a test dataset
predictions <- predict(models, newdata = pima_diabetes[41:50, ])
evaluate(predictions)
```

---

evaluate_classification

Get performance metrics for classification predictions

Description

Get performance metrics for classification predictions

Usage

```
evaluate_classification(predicted, actual)
```

Arguments

predicted	Vector of predicted probabilities
actual	Vector of realized outcomes, must be 0/1

Value

Numeric vector of scores with metric as names

Examples

```
evaluate_classification(c(.7, .1, .6, .9, .4), c(1, 0, 0, 1, 1))
```

---

evaluate_multiclass *Get performance metrics for multiclass predictions*

Description

Get performance metrics for multiclass predictions

Usage

```
evaluate_multiclass(predicted, actual)
```

Arguments

predicted	Vector of predicted probabilities
actual	Vector of realized outcomes, must be 0/1

Value

Numeric vector of scores with metric as names

Examples

```
evaluate_multiclass(iris$Species, sample(iris$Species))
```

evaluate_regression *Get performance metrics for regression predictions*

Description

Get performance metrics for regression predictions

Usage

```
evaluate_regression(predicted, actual)
```

Arguments

predicted	Vector of predicted values
actual	Vector of realized values

Value

Numeric vector of scores with metric as names

Examples

```
evaluate_regression(c(2, 4, 6), c(1.5, 4.1, 6.2))
```

explore *Explore a model's "reasoning" via counterfactual predictions*

Description

Make predictions for observations that vary over features of interest. There are two major use cases for this function. One is to understand how the model responds to features, not just individually but over combinations of features (i.e. interaction effects). The other is to explore how an individual prediction would vary if feature values were different. **Note, however, that this function does not establish causality and the latter use case should be deployed judiciously.**

Usage

```
explore(
  models,
  vary = 4,
  hold = list(numerics = median, characters = Mode),
  numerics = c(0.05, 0.25, 0.5, 0.75, 0.95),
  characters = 5
)
```

Arguments

models	A model_list object. The data the model was trained on must have been prepared, either by training with machine_learn or by preparing with prep_data before model training.
vary	Which (or how many) features to vary? Default is 4; if vary is a single integer (n), the n-most-important features are varied (see Details for how importance is determined). If vary is a vector of integers, those rankings of features are used (e.g. vary = 2:4 varies the 2nd, 3rd, and 4th most-important features). Alternatively, you can specify which features to vary by passing a vector of feature names. For the finest level of control, you can choose the alternative values to use by passing a list with names being features names and entries being values to use; in this case numerics and characters are ignored.
hold	How to choose the values of features not being varied? To make counterfactual predictions for a particular patient, this can be a row of the training data frame (or a one-row data frame containing values for all of the non-varying features). Alternatively, this can be functions to determine the values of non-varying features, in which case it must be a length-2 list with names "numerics" and "characters", each being a function to determine the values of non-varying features of that data type. The default is list(numerics = median,characters = Mode); numerics is applied to the column from the training data, characters is applied to a frequency table of the column from the training data.
numerics	How to determine values of numeric features being varied? By default, the 5th, 25th, 50th (median), 75th, and 95th percentile values from the training dataset will be used. To specify evenly spaced quantiles, starting with the 5th and ending with the 95th, pass an integer to this argument. To specify which quantiles to use, pass a numeric vector in [0, 1] to this argument, e.g. c(0,.5,1) for the minimum, median, and maximum values from the training dataset.
characters	Integer. For categorical variables being varied, how many values to use? Values are used from most- to least-common; default is 5.

Details

If vary is an integer, the most important features are determined by get_variable_importance, unless glm is the only model present, in which case interpret is used with a warning. When selecting the most important features to vary, for categorical features the sum of feature importance of all the levels as dummies is used.

Value

A tibble with values of features used to make predictions and predictions. Has class explore_df and attribute vi giving information about the varying features.

See Also

plot.explore_df

Examples

```
# First, we need a model on which to make counterfactual predictions
set.seed(5176)
m <- machine_learn(pima_diabetes, patient_id, outcome = diabetes,
                   tune = FALSE, models = "xgb")

# By default, the four most important features are varied, with numeric
# features taking their 5, 25, 50, 75, and 95 percentile values, and
# categoricals taking their five most common values. Others features are
# held at their median and modal values for numeric and categorical features,
# respectively. This can provide insight into how the model responds to
# different features
explore(m)

# It is easy to plot counterfactual predictions. By default, only the two most
# important features are plotted over; see `?plot.explore_df` for
# customization options
explore(m) %>%
  plot()

# You can specify which features vary and what values they take in a variety of
# ways. For example, you could vary only "weight_class" and "plasma_glucose"
explore(m, vary = c("weight_class", "plasma_glucose"))

# You can also control what values non-varying features take.
# For example, if you want to simulate alternative scenarios for patient 321
patient321 <- dplyr::filter(pima_diabetes, patient_id == 321)
patient321
explore(m, hold = patient321)

# Here is an example in which both the varying and non-varying feature values
# are explicitly specified.
explore(m,
        vary = list(weight_class = c("normal", "overweight", "obese"),
                    plasma_glucose = seq(60, 200, 10)),
        hold = list(pregnancies = 2,
                    pedigree = .5,
                    age = 25,
                    insulin = NA,
                    skinfold = NA,
                    diastolic_bp = 85)) %>%
  plot()
```

`flash_models` *Train models without tuning for performance*

Description

Train models without tuning for performance

Usage

```
flash_models(
  d,
  outcome,
  models,
  metric,
  positive_class,
  n_folds = 5,
  model_class,
  model_name = NULL,
  allow_parallel = FALSE
)
```

Arguments

d	A data frame from prep_data. If you want to prepare your data on your own, use prep_data(...,no_prep = TRUE).
outcome	Optional. Name of the column to predict. When omitted the outcome from prep_data is used; otherwise it must match the outcome provided to prep_data.
models	Names of models to try. See get_supported_models for available models. Default is all available models.
metric	Which metric should be used to assess model performance? Options for classification: "ROC" (default) (area under the receiver operating characteristic curve) or "PR" (area under the precision-recall curve). Options for regression: "RMSE" (default) (root-mean-squared error, default), "MAE" (mean-absolute error), or "Rsquared." Options for multiclass: "Accuracy" (default) or "Kappa" (accuracy, adjusted for class imbalance).
positive_class	For classification only, which outcome level is the "yes" case, i.e. should be associated with high probabilities? Defaults to "Y" or "yes" if present, otherwise is the first level of the outcome variable (first alphabetically if the training data outcome was not already a factor).
n_folds	How many folds to train the model on. Default = 5, minimum = 2. Whie flash_models doesn't use cross validation to tune hyperparameters, it trains n_folds models to evaluate performance out of fold.
model_class	"regression" or "classification". If not provided, this will be determined by the class of 'outcome' with the determination displayed in a message.
model_name	Quoted, name of the model. Defaults to the name of the outcome variable.
allow_parallel	Depreciated. Instead, control the number of cores though your parallel back end (e.g. with doMC).

Details

This function has two major differences from tune_models: 1. It uses fixed default hyperparameter values to train models instead of using cross-validation to optimize hyperparameter values for predictive performance, and, as a result, 2. It is much faster.

If you want to train a model at a single set of non-default hyperparameter values use tune_models and pass a single-row data frame to the hyperparameters arguemet.

Value

A model_list object. You can call `plot`, `summary`, `evaluate`, or `predict` on a model_list.

See Also

For setting up model training: `prep_data`, `supported_models`, `hyperparameters`

For evaluating models: `plot.model_list`, `evaluate.model_list`

For making predictions: `predict.model_list`

For optimizing performance: `tune_models`

To prepare data and tune models in a single step: `machine_learn`

Examples

```
## Not run:
# Prepare data
prepped_data <- prep_data(pima_diabetes, patient_id, outcome = diabetes)

# Get models quickly at default hyperparameter values
flash_models(prepped_data)

# Speed comparison of no tuning with flash_models vs. tuning with tune_models:
# ~15 seconds:
system.time(
  tune_models(prepped_data, diabetes)
)
# ~3 seconds:
system.time(
  flash_models(prepped_data, diabetes)
)

## End(Not run)
```

`get_cutoffs` *Get cutoff values for group predictions*

Description

Get cutoff values for group predictions

Usage

```
get_cutoffs(x)
```

Arguments

`x`	Data frame from `predict.model_list` where `outcome_groups` or `risk_groups` was specified

Value

A message is printed about the thresholds. If `outcome_groups` were defined the return value is a single numeric value, the threshold used to separate predicted probabilities into outcome groups. If `risk_groups` were defined the return value is a data frame with one column giving the group names and another column giving the minimum predicted probability for an observation to be in that group.

Examples

```
machine_learn(pima_diabetes[1:20, ], patient_id, outcome = diabetes,
              models = "xgb", tune = FALSE) %>%
  predict(risk_groups = 5) %>%
  get_cutoffs()
```

`get_hyperparameter_defaults` *Get hyperparameter values*

Description

Get hyperparameter values

Usage

```
get_hyperparameter_defaults(
  models = get_supported_models(),
  n = 100,
  k = 10,
  model_class = "classification"
)

get_random_hyperparameters(
  models = get_supported_models(),
  n = 100,
  k = 10,
  tune_depth = 5,
  model_class = "classification"
)
```

Arguments

`models`	which algorithms?
`n`	Number observations
`k`	Number features
`model_class`	"classification" or "regression"
`tune_depth`	How many combinations of hyperparameter values?

Details

Get hyperparameters for model training. `get_hyperparameter_defaults` returns a list of 1-row data frames (except for glm, which is a 10-row data frame) with default hyperparameter values that are used by `flash_models`. `get_random_hyperparameters` returns a list of data frames with combinations of random values of hyperparameters to tune over in `tune_models`; the number of rows in the data frames is given by 'tune_depth'.

For `get_hyperparameter_defaults` XGBoost defaults are from caret and XGBoost documentation: eta = 0.3, gamma = 0, max_depth = 6, subsample = 0.7, colsample_bytree = 0.8, min_child_weight = 1, and nrounds = 50. Random forest defaults are from Intro to Statistical Learning and caret: mtry = sqrt(k), splitrule = "extratrees", min.node.size = 1 for classification, 5 for regression. glm defaults are from caret: alpha = 1, and because glmnet fits sequences of lambda nearly as fast as an individual value, lambda is a sequence from 1e-4 to 8.

Value

Named list of data frames. Each data frame corresponds to an algorithm, and each column in each data fram corresponds to a hyperparameter for that algorithm. This is the same format that should be provided to `tune_models(hyperparameters = )` to specify hyperparameter values.

See Also

`models` for model and hyperparameter details

---

get_supported_models *Supported models and their hyperparameters*

Description

Random Forest: "rf". Regression and classification. Implemented via `ranger`.

- mtry: Number of variables to consider for each split
- splitrule: Splitting rule. For classification either "gini" or "extratrees". For regression either "variance" or "extratrees".
- min.node.size: Minimal node size.

XGBoost: "xgb". eXtreme Gradient Boosting Implemented via `xgboost`. Note that XGB has many more hyperparameters than the other models. Because of this, it may require greater tune_depth to optimize performance.

- eta: Control for learning rate, [0, 1]
- gamma: Threshold for further cutting of leaves, [0, Inf]. Larger is more conservative.
- max_depth: Maximum tree depth, [0, Inf]. Larger means more complex models and so greater likelihood of overfitting. 0 produces no limit on depth.
- subsample: Fraction of data to use in each training instance, (0, 1].
- colsample_bytree: Fraction of features to use in each tree, (0, 1].

- min_child_weight: Minimum sum of instance weight need to keep partitioning, [0, Inf]. Larger values mean more conservative models.
- nrounds: Number of rounds of boosting, [0, Inf). Larger values produce a greater likelihood of overfitting.

Regularized regression: "glm". Regression and classification. Implemented via `glmnet`.

- alpha: Elasticnet mixing parameter, in [0, 1]. 0 = ridge regression; 1 = lasso.
- lambda: Regularization parameter, > 0. Larger values make for stronger regularization.

Usage

```
get_supported_models()
```

Value

Vector of currently-supported algorithms.

See Also

`hyperparameters` for more detail on hyperparameter defaults and specifications

---

`get_thresholds` *Get class-separating thresholds for classification predictions*

Description

healthcareai gives you predicted probabilities for classification problems, but sometimes you need to convert probabilities into predicted classes. That requires choosing a threshold, where probabilities above the threshold are predicted as the positive class and probabilities below the threshold are predicted as the negative class. This function helps you do that by calculating a bunch of model-performance metrics at every possible threshold.

"cost" is an especially useful measure as it allows you to weight how bad a false alarm is relative to a missed detection. E.g. if for your use case a missed detection is five times as bad as a false alarm (another way to say that is that you're willing to allow five false positives for every one false negative), set `cost_fn = 5` and use the threshold that minimizes cost (see `examples`).

We recommend plotting the thresholds with their performance measures to see how optimizing for one measure affects performance on other measures. See `plot.thresholds_df` for how to do this.

Usage

```
get_thresholds(x, optimize = NULL, measures = "all", cost_fp = 1, cost_fn = 1)
```

Arguments

x	Either a predictions data frame (from predict) or a model_list (e.g. from machine_learn).
optimize	Optional. If provided, one of the entries in measures. A logical column named "optimal" will be added with one TRUE entry corresponding to the threshold that optimizes this measure.
measures	Character vector of performance metrics to calculate, or "all", which is equivalent to using all of the following measures. The returned data frame will have one column for each metric. • cost: Captures how bad all the errors are. You can adjust the relative costs of false alarms and missed detections by setting cost_fp or cost_fn. At the default of equal costs, this is directly inversely proportional to accuracy. • acc: Accuracy • tpr: True positive rate, aka sensitivity, aka recall • tnr: True negative rate, aka specificity • fpr: False positive rate, aka fallout • fnr: False negative rate • ppv: Positive predictive value, aka precision • npv: Negative predictive value
cost_fp	Cost of a false positive. Default = 1. Only affects cost.
cost_fn	Cost of a false negative. Default = 1. Only affects cost.

Value

Tibble with rows for each possible threshold and columns for the thresholds and each value in measures.

Examples

```
library(dplyr)
models <- machine_learn(pima_diabetes[1:15, ], patient_id, outcome = diabetes,
                        models = "xgb", tune = FALSE)
get_thresholds(models)

# Identify the threshold that maximizes accuracy:
get_thresholds(models, optimize = "acc")

# Assert that one missed detection is as bad as five false alarms and
# filter to the threshold that minimizes "cost" based on that assertion:
get_thresholds(models, optimize = "cost", cost_fn = 5) %>%
  filter(optimal)

# Use that threshold to make class predictions
(class_preds <- predict(models, outcome_groups = 5))
attr(class_preds$predicted_group, "cutpoints")

# Plot performance on all measures across threshold values
```

```
get_thresholds(models) %>%
  plot()

# If a measure is provided to optimize, the best threshold will be highlighted in plots
get_thresholds(models, optimize = "acc") %>%
  plot()

## Transform probability predictions into classes based on an optimal threshold ##
# Pull the threshold that minimizes cost
optimal_threshold <-
  get_thresholds(models, optimize = "cost") %>%
  filter(optimal) %>%
  pull(threshold)

# Add a Y/N column to predictions based on whether the predicted probability
# is greater than the threshold
class_predictions <-
  predict(models) %>%
  mutate(predicted_class_diabetes = case_when(
    predicted_diabetes > optimal_threshold ~ "Y",
    predicted_diabetes <= optimal_threshold ~ "N"
  ))

class_predictions %>%
  select_at(vars(ends_with("diabetes"))) %>%
  arrange(predicted_diabetes)

# Examine the expected volume of false-and-true negatives-and-positive
table(Actual = class_predictions$diabetes,
      Predicted = class_predictions$predicted_class_diabetes)
```

`get_variable_importance` *Get variable importances*

Description

Get variable importances

Usage

```
get_variable_importance(models, remove_zeros = TRUE, top_n)
```

Arguments

`models`	model_list object
`remove_zeros`	Remove features with zero variable importance? Default is TRUE
`top_n`	Integer: How many variables to return? The top_n most important variables be returned. If missing (default), all variables are returned

Details

Some algorithms provide variable importance, others don't. The best-performing model that offers variable importance will be used.

Value

Data frame of variables and their importance for predictive power

See Also

plot.variable_importance

Examples

```
m <- machine_learn(mtcars, outcome = mpg, models = "rf", tune = FALSE)
(vi <- get_variable_importance(m))
plot(vi)
```

---

hcai_impute *Specify imputation methods for an existing recipe*

Description

'hcai-impute' adds various imputation methods to an existing recipe. Currently supports mean (numeric only), new_category (categorical only), bagged trees, or knn.

Usage

```
hcai_impute(
  recipe,
  nominal_method = "new_category",
  numeric_method = "mean",
  numeric_params = NULL,
  nominal_params = NULL
)
```

Arguments

recipe	A recipe object. imputation will be added to the sequence of operations for this recipe.
nominal_method	Defaults to "new_category". Other choices are "bagimpute", "knnimpute" or "locfimpute".
numeric_method	Defaults to "mean". Other choices are "bagimpute", "knnimpute" or "locfimpute".
numeric_params	A named list with parmeters to use with chosen imputation method on numeric data. Options are bag_model (bagimpute only), bag_trees (bagimpute only), bag_options (bagimpute only), bag_trees (bagimpute only), knn_K (knnimpute only), impute_with (knnimpute only), (bag or knn) or seed_val (bag or knn). See step_bagimpute or step_knnimpute for details.

`nominal_params` A named list with parmeters to use with chosen imputation method on nominal data. Options are `bag_model` (bagimpute only), `bag_trees` (bagimpute only), `bag_options` (bagimpute only), `bag_trees` (bagimpute only), `knn_K` (knnimpute only), `impute_with` (knnimpute only), (bag or knn) or `seed_val` (bag or knn). See step_bagimpute or step_knnimpute for details.

Value

An updated version of 'recipe' with the new step added to the sequence of existing steps.

Examples

```
library(recipes)

n = 100
set.seed(9)
d <- tibble::tibble(patient_id = 1:n,
            age = sample(c(30:80, NA), size = n, replace = TRUE),
            hemoglobin_count = rnorm(n, mean = 15, sd = 1),
            hemoglobin_category = sample(c("Low", "Normal", "High", NA),
                                         size = n, replace = TRUE),
            disease = ifelse(hemoglobin_count < 15, "Yes", "No")
)

# Initialize
my_recipe <- recipe(disease ~ ., data = d)

# Create recipe
my_recipe <- my_recipe %>%
  hcai_impute()
my_recipe

# Train recipe
trained_recipe <- prep(my_recipe, training = d)

# Apply recipe
data_modified <- bake(trained_recipe, new_data = d)
missingness(data_modified)


# Specify methods:
my_recipe <- my_recipe %>%
  hcai_impute(numeric_method = "bagimpute",
    nominal_method = "locfimpute")
my_recipe

# Specify methods and params:
my_recipe <- my_recipe %>%
  hcai_impute(numeric_method = "knnimpute",
    numeric_params = list(knn_K = 4))
my_recipe
```

healthcareai *Machine Learning Made Easy*

Description

healthcare.ai makes it as easy as possible to pull data from a database, get it ready for machine learning, optimize multiple models, and deploy predictions.

Details

The package website – https://docs.healthcare.ai/ – contains vignettes that demonstrate how to use the package, as well as documentation of all the important functions.

impute *Impute data and return a reusable recipe*

Description

`impute` will impute your data using a variety of methods for both nominal and numeric data. Currently supports mean (numeric only), new_category (categorical only), bagged trees, or knn.

Usage

```
impute(
  d = NULL,
  ...,
  recipe = NULL,
  numeric_method = "mean",
  nominal_method = "new_category",
  numeric_params = NULL,
  nominal_params = NULL,
  verbose = FALSE
)
```

Arguments

`d`	A dataframe or tibble containing data to impute.
`...`	Optional. Unquoted variable names to not be imputed. These will be returned unaltered.
`recipe`	Optional, a recipe object or an imputed data frame (containing a recipe object as an attribute). If provided, this recipe will be applied to impute new data contained in d with values saved in the recipe. Use this param if you'd like to apply the same values used for imputation on a training dataset in production.
`numeric_method`	Defaults to `"mean"`. Other choices are `"bagimpute"` or `"knnimpute"`.

nominal_method	Defaults to "new_category". Other choices are "bagimpute" or "knnimpute".
numeric_params	A named list with parmeters to use with chosen imputation method on numeric data. Options are bag_model (bagimpute only), bag_trees (bagimpute only), bag_options (bagimpute only), bag_trees (bagimpute only), knn_K (knnimpute only), impute_with (knnimpute only), (bag or knn) or seed_val (bag or knn). See step_bagimpute or step_knnimpute for details.
nominal_params	A named list with parmeters to use with chosen imputation method on nominal data. Options are bag_model (bagimpute only), bag_trees (bagimpute only), bag_options (bagimpute only), bag_trees (bagimpute only), knn_K (knnimpute only), impute_with (knnimpute only), (bag or knn) or seed_val (bag or knn). See step_bagimpute or step_knnimpute for details.
verbose	Gives a print out of what will be imputed and which method will be used.

Value

Imputed data frame with reusable recipe object for future imputation in attribute "recipe".

Examples

```
d <- pima_diabetes
d_train <- d[1:700, ]
d_test <- d[701:768, ]
# Train imputer
train_imputed <- impute(d = d_train, patient_id, diabetes)
# Apply to new data
impute(d = d_test, patient_id, diabetes, recipe = train_imputed)
# Specify methods:
impute(d = d_train, patient_id, diabetes, numeric_method = "bagimpute",
nominal_method = "new_category")
# Specify method and param:
impute(d = d_train, patient_id, diabetes, nominal_method = "knnimpute",
nominal_params = list(knn_K = 4))
```

---

interpret *Interpret a model via regularized coefficient estimates*

Description

Interpret a model via regularized coefficient estimates

Usage

```
interpret(x, sparsity = NULL, remove_zeros = TRUE, top_n)
```

Arguments

x	a model_list object containing a glmnet model
sparsity	If NULL (default) coefficients for the best-performing model will be returned. Otherwise, a value in [0, 1] that determines the sparseness of the model for which coefficients will be returned, with 0 being maximally sparse (i.e. having the fewest non-zero coefficients) and 1 being minimally sparse
remove_zeros	Remove features with coefficients equal to 0? Default is TRUE
top_n	Integer: How many coefficients to return? The largest top_n absolute-value coefficients will be returned. If missing (default), all coefficients are returned

Details

WARNING Coefficients are on the scale of the predictors; they are not standardized, so unless features were scaled before training (e.g. with `prep_data(...,scale = TRUE)`, the magnitude of coefficients does not necessarily reflect their importance.

If x was trained with more than one value of alpha the best value of alpha is used; sparsity is determined only via the selection of lambda. Using only lasso regression (i.e. alpha = 1) will produce a sparser set of coefficients and can be obtained by not tuning hyperparameters.

Value

A data frame of variables and their regularized regression coefficient estimates with parent class "interpret"

See Also

`plot.interpret`

Examples

```
m <- machine_learn(pima_diabetes, patient_id, outcome = diabetes, models = "glm")
interpret(m)
interpret(m, .2)
interpret(m) %>%
  plot()
```

---

is.model_list *Type checks*

Description

Type checks

Usage

```
is.model_list(x)

is.classification_list(x)

is.regression_list(x)

is.multiclass_list(x)
```

Arguments

x	Object

Value

Logical

`is.predicted_df` *Class check*

Description

Class check

Usage

```
is.predicted_df(x)
```

Arguments

x	object

Value

logical

machine_learn *Machine learning made easy*

Description

Prepare data and train machine learning models.

Usage

```
machine_learn(
  d,
  ...,
  outcome,
  models,
  metric,
  tune = TRUE,
  positive_class,
  n_folds = 5,
  tune_depth = 10,
  impute = TRUE,
  model_name = NULL,
  allow_parallel = FALSE
)
```

Arguments

d	A data frame
...	Columns to be ignored in model training, e.g. ID columns, unquoted.
outcome	Name of the target column, i.e. what you want to predict. Unquoted. Must be named, i.e. you must specify `outcome =`
models	Names of models to try. See `get_supported_models` for available models. Default is all available models.
metric	Which metric should be used to assess model performance? Options for classification: "ROC" (default) (area under the receiver operating characteristic curve) or "PR" (area under the precision-recall curve). Options for regression: "RMSE" (default) (root-mean-squared error, default), "MAE" (mean-absolute error), or "Rsquared." Options for multiclass: "Accuracy" (default) or "Kappa" (accuracy, adjusted for class imbalance).
tune	If TRUE (default) models will be tuned via `tune_models`. If FALSE, models will be trained via `flash_models` which is substantially faster but produces less-predictively powerful models.
positive_class	For classification only, which outcome level is the "yes" case, i.e. should be associated with high probabilities? Defaults to "Y" or "yes" if present, otherwise is the first level of the outcome variable (first alphabetically if the training data outcome was not already a factor).

n_folds	How many folds to use to assess out-of-fold accuracy? Default = 5. Models are evaluated on out-of-fold predictions whether tune is TRUE or FALSE.
tune_depth	How many hyperparameter combinations to try? Default = 10. Value is multiplied by 5 for regularized regression. Ignored if tune is FALSE.
impute	Logical, if TRUE (default) missing values will be filled by hcai_impute
model_name	Quoted, name of the model. Defaults to the name of the outcome variable.
allow_parallel	Depreciated. Instead, control the number of cores though your parallel back end (e.g. with doMC).

Details

This is a high-level wrapper function. For finer control of data cleaning and preparation use prep_data or the functions it wraps. For finer control of model tuning use tune_models.

Value

A model_list object. You can call plot, summary, evaluate, or predict on a model_list.

Examples

```
# These examples take about 30 seconds to execute so aren't run automatically,
# but you should be able to execute this code locally.
## Not run:
# Split the data into training and test sets
d <- split_train_test(d = pima_diabetes,
                      outcome = diabetes,
                      percent_train = .9)

### Classification ###

# Clean and prep the training data, specifying that patient_id is an ID column,
# and tune algorithms over hyperparameter values to predict diabetes
diabetes_models <- machine_learn(d$train, patient_id, outcome = diabetes)

# Inspect model specification and performance
diabetes_models

# Make predictions (predicted probability of diabetes) on test data
predict(diabetes_models, d$test)

### Regression ###

# If the outcome variable is numeric, regression models will be trained
age_model <- machine_learn(d$train, patient_id, outcome = age)

# Get detailed information about performance over tuning values
summary(age_model)

# Get available performance metrics
evaluate(age_model)
```

```
# Plot training performance on tuning metric (default = RMSE)
plot(age_model)

# If new data isn't specifed, get predictions on training data
predict(age_model)

### Faster model training without tuning hyperparameters ###

# Train models at set hyperparameter values by setting tune to FALSE. This is
# faster (especially on larger datasets), but produces models with less
# predictive power.
machine_learn(d$train, patient_id, outcome = diabetes, tune = FALSE)

### Train models optimizing given metric ###

machine_learn(d$train, patient_id, outcome = diabetes, metric = "PR")

## End(Not run)
```

`make_na` *Replace missingness values with NA and correct columns types*

Description

This function replaces given missingness values with NA in a given dataframe or tibble. Numeric vectors that were originally loaded as character or factor vectors (because of missingness values in the column), are also converted to numeric vectors when values are replaced.

Usage

```
make_na(d, to_replace, drop_levels = TRUE)
```

Arguments

`d`	A dataframe or tibble
`to_replace`	A value or vector of values that will be replaced with NA
`drop_levels`	If TRUE (default) unused factor levels are dropped

Value

A tibble where the missing value/values is/are replaced with NA, columns that only have numbers left are coerced to numeric type

Examples

```
dat <- data.frame(gender = c("male", "male", "female", "male", "missing"),
                  name = c("Paul", "Jim", "Sarah", "missing", "Alex"),
                  weight = c(139, 0, 193, 158, 273))

# Replace "missing" in `dat`
make_na(dat, "missing")

# If there are multiple missing values, pass them through a vector.
dat <- data.frame(gender = c("male", "??", "female", "male", "NULL"),
                  age = c(64, 52, 75, "NULL", 70),
                  weight = c(139, 0, 193, "??", 273),
                  stringsAsFactors = FALSE)

make_na(dat, c("??", "NULL"))

# Run `missingness()` to find possible missingness values in `dat`. It will
# suggest the default implementation of `make_na` to replace all found
# missingness values (the suggested default implementation for this example
# is `make_na(dat, c("??", "NULL"))`).
missingness(dat)
make_na(dat, c("??", "NULL"))

# Note: In this last example, `age` should be loaded as a numeric vector, but
# since "NULL" is present, it is stored as a character vector. When "NULL" is
# replaced, `age` will be converted to a numeric vector.
```

`missingness` *Find missingness in each column and search for strings that might represent missing values*

Description

Finds the percent of NAs in a vector or in each column of a dataframe or matrix or in a vector. Possible mis-coded missing values are searched for and a warning issued if they are found.

Usage

```
missingness(
  d,
  return_df = TRUE,
  to_search = c("NA", "NAs", "na", "NaN", "?", "??", "nil", "NULL", " ", "")
)
```

Arguments

`d`	A data frame or matrix

return_df	If TRUE (default) a data frame is returned, which generally makes reading the output easier. If variable names are so long that the data frame gets wrapped poorly, set this to FALSE.
to_search	A vector of strings that might represent missingness. If found in d, a warning is issued.

Value

A data frame with two columns: variable names in d and the percent of entries in each variable that are missing.

See Also

plot.missingness

Examples

```
d <- data.frame(x = c("a", "nil", "b"),
                y = c(1, NaN, 3),
                z = c(1:2, NA))
missingness(d)
missingness(d) %>% plot()
```

Mode *Mode*

Description

Mode

Usage

```
Mode(x)
```

Arguments

x	Either a vector or a frequency table from table

Value

The modal value of x

Examples

```
x <- c(3, 1:5)
Mode(x)
Mode(table(x))
```

`pima_diabetes` *Patient diabetes dataset*

Description

A dataset containing diabetes status and other health-related variables for 768 females, at least 21 years old, of Pima Indian heritage. As pointed out (see source URL below), the source data had some biologically impossible zero values. We have replaced zero values in every variable except `Pregnancies` with NA.

Usage

```
pima_diabetes
```

Format

A tibble data frame with 768 rows and 10 variables:

patient_id Unique identifier

pregnancies Number of times pregnant

plasma_glucose Plasma glucose concentration 2 hours in an oral glucose tolerance test

diastolic_bp Diastolic blood pressure (mm Hg)

skinfold Triceps skin fold thickness (mm)

insulin 2-Hour serum insulin (mu U/ml)

weight_class Derived from BMI

pedigree Diabetes pedigree function

age Age (years)

diabetes Y/N diagnosis per WHO criteria

Source

https://archive.ics.uci.edu/ml/datasets/pima+indians+diabetes

See Also

`pima_meds`

pima_meds *Patient medications dataset*

Description

This is a companion dataset for pima_diabetes. The pima_diabetes dataset is real; this dataset is synthetic. You can see how it was generated here: https://docs.healthcare.ai/articles/site_only/best_levels.html#appendix-data-generation. Briefly, each patient in pima_diabetes is assigned 0-4 medications from the following six: insulin and metformin are more common among diabetics, prednisone and metoprolol are less common among diabetics, and nexium and tiotropium are equally likely among diabetic and non-diabetic patients. Each patient-medication has a years_taken value associated with it, which is a random number drawn from an exponential distribution.

Usage

```
pima_meds
```

Format

A tibble data frame with 1,604 rows and 3 variables:

patient_id Unique identifier, used to join pima_diabetes

medication One of the six medications described above

years_taken Numeric value indicating the duration the medication has been used

See Also

pima_diabetes

pip *Patient Impact Predictor*

Description

Identify opportunities to improve patient outcomes by exploring changes in predicted outcomes over changes to input variables. **Note that causality cannot be established by this function.** Omitted variable bias and other statistical phenomena may mean that the impacts predicted here are not realizable. Clinical guidance is essential in choosing new_values and acting on impact predictions. Extensive options are provided to control what impact predictions are surfaced, including variable_direction and prohibited_transitions.

Usage

```
pip(
  model,
  d,
  new_values,
  n = 3,
  allow_same = FALSE,
  repeated_factors = FALSE,
  smaller_better = TRUE,
  variable_direction = NULL,
  prohibited_transitions = NULL,
  id
)
```

Arguments

model	A model_list object, as from machine_learn or tune_models
d	A data frame on which model can make predictions
new_values	A list of alternative values for variables of interest. The names of the list must be variables in d and the entries are the alternative values to try.
n	Integer, default = 3. The maximum number of alternatives to return for each patient. Note that the actual number returned may be less than n, for example if length(new_values) < n or if allow_same is FALSE.
allow_same	Logical, default = FALSE. If TRUE, pip may return rows with modified_value = original_value and improvement = 0. This happens when there are fewer than n modifications for a patient that result in improvement. If allow_same is TRUE and length(new_values) >= n you are likely to get n results for each patient; however, contraints from variable_direction or prohibited_transitions could make recommendations for some variables impossible, resulting in fewer than n recommendations.
repeated_factors	Logical, default = FALSE. Do you want multiple modifications of the same variable for the same patient?
smaller_better	Logical, default = TRUE. Are lesser values of the outcome variable in model preferable?
variable_direction	Named numeric vector or list with entries of -1 or 1. This specifies the direction numeric variables are permitted to move to produce improvements. Names of the vector are names of variables in d; entries are 1 to indicate only increases can yield improvements or -1 to indicate only decreases can yield improvements. Numeric variables not appearing in this list may increase or decrease to surface improvements.
prohibited_transitions	A list of data frames that contain variable modifications that won't be considered by pip. Names of the list are names of variables in d, and data frames have two columns, "from" and "to", indicating the original value and modified value,

respectively, of the prohibited transition. If column names are not "from" and "to", the first column will be assumed to be the "from" column. This is intended for categorical variables, but could be used for integers as well.

`id` Optional. A unquoted variable name in `d` representing an identifier column; it will be included in the returned data frame. If not provided, an ID column from `model`'s data prep will be used if available.

Value

A tibble with any id columns and "variable": the name of the variable being altered, "original_value": the patient's observed value of "variable", "modified_value": the altered value of "variable", "original_prediction": the patient's original prediction, "modified_prediction": the patient's prediction given the that "variable" changes to "modified_value", "improvement": the difference between the original and modified prediction with positive values reflecting improvement based on the value of `smaller_better`, and "impact_rank": the rank of the modification for that patient.

Examples

```
# First, we need a model to make recommendations
set.seed(52760)
m <- machine_learn(pima_diabetes, patient_id, outcome = diabetes,
                   tune = FALSE, models = "xgb")
# Let's look at changes in predicted outcomes for three patients changing their
# weight class, blood glucose, and blood pressure
modifications <- list(weight_class = c("underweight", "normal", "overweight"),
                      plasma_glucose = c(75, 100),
                      diastolic_bp = 70)
pip(model = m, d = pima_diabetes[1:3, ], new_values = modifications)

# In the above example, only the first patient has a positive predicted impact
# from changing their diastolic_bp, so for the other patients fewer than the
# default n=3 predictions are provided. We can get n=3 predictions for each
# patient by specifying allow_same, which will recommend the other two patients
# maintain their current diastolic_bp.
pip(model = m, d = pima_diabetes[1:3, ], new_values = modifications, allow_same = TRUE)

# Sometimes clinical knowledge trumps machine learning. In particular, machine
# learning models don't establish causality, they only leverage correlation.
# Patient impact predictor suggests causality, so clinicians should always be
# consulted to ensure that the causal impacts are medically sound.
#
# If there is clinical knowledge to suggest what impact a variable should have,
# that knowledge can be provided to pip. The way it is provided depends on
# whether the variable is categorical (prohibited_transitions) or numeric
# (variable_direction).

### Constraining categorical variables ###
# Suppose a clinician says that recommending a patient change their weight class
# to underweight from any value except normal is a bad idea. We can disallow
# those suggestions using prohibited_transitions. Note the change in patient
# 1's second recommendation goes from underweight to normal.
```

```
prohibit <- data.frame(from = setdiff(unique(pima_diabetes$weight_class), "normal"),
                       to = "underweight")
pip(model = m, d = pima_diabetes[1:3, ], new_values = modifications,
    prohibited_transitions = list(weight_class = prohibit))

### Constraining numeric variables ###
# Suppose a clinician says that increasing diastolic_bp should never be
# recommended to improve diabetes outcomes, and likewise for reducing
# plasma_glucose (which is clinically silly, but provides an illustration). The
# following code ensures that diastolic_bp is only recommended to decrease and
# plasma_glucose is only recommended to increase. Note that the plasma_glucose
# recommendations disappear, because no patient would see their outcomes
# improve by increasing their plasma_glucose.
directional_changes <- c(diastolic_bp = -1, plasma_glucose = 1)
pip(model = m, d = pima_diabetes[1:3, ], new_values = modifications,
    variable_direction = directional_changes)
```

`pivot` *Pivot multiple rows per observation to one row with multiple columns*

Description

Pivot multiple rows per observation to one row with multiple columns

Usage

```
pivot(d, grain, spread, fill, fun = sum, missing_fill = NA, extra_cols)
```

Arguments

`d`	data frame
`grain`	Column that defines rows. Unquoted.
`spread`	Column that will become multiple columns. Unquoted.
`fill`	Column to be used to fill the values of cells in the output, perhaps after aggregation by `fun`. If `fill` is not provided, counts will be used, as though a fill column of 1s had been provided.
`fun`	Function for aggregation, defaults to `sum`. Custom functions can be used with the same syntax as the apply family of functions, e.g. `fun = function(x) some_function(another_fun(x))`.
`missing_fill`	Value to fill for combinations of grain and spread that are not present. Defaults to NA, but 0 may be useful as well.
`extra_cols`	Values of `spread` to create all-`missing_fill` columns, for e.g. if you want to add levels that were observed in training but are not present in deployment.

Details

pivot is useful when you want to change the grain of your data, for example from the procedure grain to the patient grain. In that example, each patient might have 0, 1, or more medications. To make a patient-level table, we need a column for each medication, which is what it means to make a wide table. The fill argument dictates what to put in each of the medication columns, e.g. the dose the patient got. fill defaults to "1", as an indicator variable. If any patients have multiple rows for the same medication (say they recieved a med more than once), we need a way to deal with that, which is what the fun argument handles. By default it uses sum, so if fill is left as its default, the count of instances for each patient will be used.

Value

A tibble data frame with one row for each unique value of grain, and one column for each unique value of spread plus one column for the entries in grain.

Entries in the tibble are defined by the fill column. Combinations of grain x spread that are not present in d will be filled in with missing_fill. If there are grain x spread pairs that appear more than once in d, they will be aggregated by fun.

Examples

```
meds <-
  tibble::tibble(
    patient_id = c("A", "A", "A", "B"),
    medication = c("zoloft", "asprin", "lipitor", "asprin"),
    pills_per_day = c(1, 8, 2, 4)
  )
meds

# Number of pills of each medication each patient gets:
pivot(
  d = meds,
  grain = patient_id,
  spread = medication,
  fill = pills_per_day,
  missing_fill = 0
)

bills <-
  tibble::tibble(
    patient_id = rep(c("A", "B"), each = 4),
    dept_id = rep(c("ED", "ICU"), times = 4),
    charge = runif(8, 0, 1e4),
    date = as.Date("2024-12-25") - sample(0:2, 8, TRUE)
  )
bills

# Total charges per patient x department:
pivot(bills, patient_id, dept_id, charge, sum)

# Count of charges per patient x day:
pivot(bills, patient_id, date)
```

```
# Can provide a custom function to fun, which will take fill as input.
# Get the difference between the greatest and smallest charge in each
# department for each patient and format it as currency.
pivot(d = bills,
      grain = patient_id,
      spread = dept_id,
      fill = charge,
      fun = function(x) paste0("$", round(max(x) - min(x), 2))
)
```

`plot.explore_df` *Plot Counterfactual Predictions*

Description

Plot Counterfactual Predictions

Usage

```
## S3 method for class 'explore_df'
plot(
  x,
  n_use = 2,
  aggregate_fun = median,
  reorder_categories = TRUE,
  x_var,
  color_var,
  jitter_y = TRUE,
  sig_fig = 3,
  font_size = 11,
  strip_font_size = 0.85,
  line_width = 0.5,
  line_alpha = 0.7,
  rotate_x = FALSE,
  nrows = 1,
  title = NULL,
  caption,
  print = TRUE,
  ...
)
```

Arguments

x	A explore_df object from explore
n_use	Number of features to vary, default = 4. If the number of features varied in explore is greater than n_use, additional features will be aggregated over by aggregate_fun

aggregate_fun	Default = median. Varying features in x are mapped to the x-axis, line color, and vertical- and horizontal facets. If more than four features vary, this function is used to aggreagate across the least-important varying features.
reorder_categories	If TRUE (default) varying categorical features are arranged by their median predicted outcome. If FALSE, the incoming level orders are retained, which is alphabetical by default, but you can set your own level orders with `reorder`
x_var	Feature to put on the x-axis (unquoted). If not provided, the most important feature is used, with numerics prioritized if one varies
color_var	Feature to color lines (unquoted). If not provided, the most important feature excluding `x_var` is used.
jitter_y	If TRUE (default) and a feature is mapped to color (i.e. if there is more than one varying feature), the vertical location of the lines will be jittered slightly (no more than 1 avoid overlap.
sig_fig	Number of significant figures (digits) to use in labels of numeric features. Default = 3; set to Inf to not truncate decimals.
font_size	Parent font size for the plot. Default = 11
strip_font_size	Relative font size for facet strip title font. Default = 0.85
line_width	Width of lines. Default = 0.5
line_alpha	Opacity of lines. Default = 0.7
rotate_x	If FALSE (default), x axis tick labels are positioned horizontally. If TRUE, they are rotated one quarter turn, which can be helpful when a categorical feature with long labels is mapped to x.
nrows	Only used when the number of varying features is three. The number of rows into which the facets will be arranged. Default = 1. NULL lets the number be determined algorithmically
title	Plot title
caption	Plot caption. Defaults to model used to make counterfactual predictions. Can be a string for custom caption or NULL for no caption.
print	Print the plot? Default is FALSE. Either way, the plot is invisibly returned
...	Not used

Value

ggplot object, invisibly

Examples

```
# First, we need a model
set.seed(4956)
m <- machine_learn(pima_diabetes, patient_id, outcome = pregnancies,
                   models = "rf", tune = FALSE)
# Then we can explore our model through counterfactual predictions
counterfactuals <- explore(m)
```

```
# By default only the two most important varying features are plotted. This
# example shows how counterfactual predictions can provide insight into how a
# model maps inputs (features) to the output (outcome). This plot shows that for
# this dataset, age is the most important predictor of the number of pregnancies
# a woman has had, and the predicted number of pregnancies rises basically
# linearly from approximately 20 to 40 and then levels off.
plot(counterfactuals)

# To see the effects of more features in the model, increase the value of
# `n_use`. You can also specify which of the varying features are mapped to the
# x-axis and the color scale, and you can customize a variety of plot attributes
plot(counterfactuals, n_use = 3, x_var = weight_class, color_var = age,
    font_size = 9, strip_font_size = 1, line_width = 2, line_alpha = .5,
    rotate_x = TRUE, nrows = 1)

# And you can further modify the plot like any other ggplot object
p <- plot(counterfactuals, n_use = 1, print = FALSE)
p +
  ylab("predicted number of pregnancies") +
  theme_classic() +
  theme(aspect.ratio = 1,
        panel.background = element_rect(fill = "slateblue"),
        plot.caption = element_text(face = "italic"))
```

`plot.interpret` *Plot regularized model coefficients*

Description

Plot regularized model coefficients

Usage

```
## S3 method for class 'interpret'
plot(
  x,
  include_intercept = FALSE,
  max_char = 40,
  title,
  caption,
  font_size = 11,
  point_size = 3,
  print = TRUE,
  ...
)
```

Arguments

x	A interpret object or a data frame with columns "variable" and "coefficient"
include_intercept	If FALSE (default) the intercept estimate will not be plotted
max_char	Maximum length of variable names to leave untruncated. Default = 40; use Inf to prevent truncation. Variable names longer than this will be truncated to leave the beginning and end of each variable name, bridged by " ... ".
title	Plot title. NULL for no title; character for custom title. If left blank contains the model class and outcome variable
caption	Plot caption, appears in lower-right. NULL for no caption; character for custom caption. If left blank the caption will contain info including the hyperparameter values of the model used by interpret to determine coefficient estimates.
font_size	Relative size of all fonts in plot, default = 11
point_size	Size of dots, default = 3
print	Print the plot? Default = TRUE
...	Unused

Value

A ggplot object, invisibly.

See Also

interpret

Examples

```
machine_learn(mtcars, outcome = mpg, models = "glm", tune = FALSE) %>%
  interpret() %>%
  plot(font_size = 14)
```

---

plot.missingness *Plot missingness*

Description

Plot missingness

Usage

```
## S3 method for class 'missingness'
plot(
  x,
  remove_zeros = FALSE,
  max_char = 40,
  title = NULL,
  font_size = 11,
  point_size = 3,
  print = TRUE,
  ...
)
```

Arguments

x	Data frame from missingness
remove_zeros	Remove variables with no missingness from the plot? Default = FALSE
max_char	Maximum length of variable names to leave untruncated. Default = 40; use Inf to prevent truncation. Variable names longer than this will be truncated to leave the beginning and end of each variable name, bridged by " ... ".
title	Plot title
font_size	Relative size of all fonts in plot, default = 11
point_size	Size of dots, default = 3
print	Print the plot? Default = TRUE
...	Unused

Value

A ggplot object, invisibly.

See Also

missingness

Examples

```
pima_diabetes %>%
  missingness() %>%
  plot()
```

plot.model_list *Plot performance of models*

Description

Plot performance of models

Usage

```
## S3 method for class 'model_list'
plot(x, font_size = 11, point_size = 1, print = TRUE, ...)
```

Arguments

x	modellist object as returned by tune_models or machine_learn
font_size	Relative size of all fonts in plot, default = 11
point_size	Size of dots, default = 3
print	If TRUE (default) plot is printed
...	Unused

Value

Plot of model performance as a function of algorithm and hyperparameter values tuned over. Generally called for the side effect of printing a plot, but the plot is also invisibly returned. The best-performing model within each algorithm will be plotted as a triangle.

Examples

```
models <- machine_learn(mtcars, outcome = mpg, models = "glm")
plot(models)
```

plot.predicted_df *Plot model predictions vs observed outcomes*

Description

Plot model predictions vs observed outcomes

Usage

```
## S3 method for class 'predicted_df'
plot(
  x,
  caption = TRUE,
  title = NULL,
  font_size = 11,
  outcomes = NULL,
  fixed_aspect = attr(x, "model_info")$type == "Regression",
  print = TRUE,
  ...
)

plot_regression_predictions(x, point_size = 1, point_alpha = 1, target)

plot_classification_predictions(
  x,
  fill_colors = c("firebrick", "steelblue"),
  fill_alpha = 0.7,
  curve_flex = 1,
  add_labels = TRUE,
  target
)

plot_multiclass_predictions(
  x,
  conf_colors = c("black", "steelblue"),
  text_color = "yellow",
  text_size = 3,
  text_angle = 60,
  diag_color = "red",
  target
)
```

Arguments

x	data frame as returned ‘predict.model_list‘
caption	Put model performance in plot caption? TRUE (default) prints all available metrics, FALSE prints nothing. Can also provide metric name (e.g. "RMSE"), in which case the caption will include only that metric.
title	Character: Plot title, default NULL produces no title.
font_size	Number: Relative size of all font in plot, default = 11
outcomes	Vector of outcomes if not present in x
fixed_aspect	Logical: If TRUE (default for regression only), units of the x- and y-axis will have the same spacing.
print	Logical, if TRUE (default) the plot is printed on the current graphics device. The plot is always (silently) returned.

...	Parameters specific to plot_regression_predictions or plot_classification_predictions; listed below. These must be named.
point_size	Number: Point size, relative to 1
point_alpha	Number in [0, 1] giving point opacity
target	Not meant to be set by user. outcome column name
fill_colors	Length-2 character vector: colors to fill density curves. Default is c("firebrick", "steelblue"). If named, names must match `unique(x[[target]])`, in any order.
fill_alpha	Number in [0, 1] giving opacity of fill colors.
curve_flex	Numeric. Kernal adjustment for density curves. Default is 1. Less than 1 makes curves more flexible, analogous to smaller bins in a histogram; greater than 1 makes curves more rigid.
add_labels	If TRUE (default) and a predicted_group column was added to predictions by specifying `risk_groups` or `outcome_groups` in `link{predict.model_list}`, labels specifying groups are added to the plot.
conf_colors	Length-2 character vector: colors to fill density curves. Default is c("black", "steelblue").
text_color	Character: color to write percent correct. Default is "yellow".
text_size	Numeric or logical: size of percent correct text. Defaults to 3, a readable size. Greater than 20 classes might need smaller text. Text can be turned off by setting to FALSE.
text_angle	Numeric or logical: angle to rotate x axis text. Defaults to 60 degrees. Setting to FALSE will turn text horizontal.
diag_color	Character: color to highlight main diagonal. These are correct predictions. Default is "red".

Details

Note that a ggplot object is returned, so you can do additional customization of the plot. See the third example.

Value

A ggplot object

Examples

```
# Some regression examples
models <- machine_learn(pima_diabetes[1:50, ], patient_id, outcome = plasma_glucose,
                        models = "rf", tune = FALSE)
predictions <- predict(models)
plot(predictions)
plot(predictions, caption = "Rsquared",
     title = "This model's predictions regress to the mean",
     point_size = 3, point_alpha = .7, font_size = 9)
p <- plot(predictions, print = FALSE)
p + theme_classic()
```

```
# A classification example with risk groups
class_models <- machine_learn(pima_diabetes, patient_id, outcome = diabetes,
                              models = "xgb", tune = FALSE)
predict(class_models,
        risk_groups = c("v low", "low", "medium", "high", "very high")) %>%
  plot()
```

`plot.thresholds_df` *Plot threshold performance metrics*

Description

Plot threshold performance metrics

Usage

```
## S3 method for class 'thresholds_df'
plot(
  x,
  title = NULL,
  caption = NULL,
  font_size = 11,
  line_size = 0.5,
  point_size = NA,
  ncol = 2,
  print = TRUE,
  ...
)
```

Arguments

`x`	A `threshold_df` object from `get_thresholds` or a data frame with columns "threshold" and other columns to be plotted against thresholds. If `optimize` was provided to `get_thresholds` a line is drawn in each facet corresponding to the optimal threshold.
`title`	Plot title. Default NULL produces no title
`caption`	Plot caption. Default NULL produces no caption unless `get_thresholds(optimize)` was provided, in which case information about the threshold and performance are provided in the caption.
`font_size`	Relative size of all fonts in plot, default = 11
`line_size`	Width of lines, default = 0.5
`point_size`	Point size. Default is `NA` which suppresses points. Set to a number to see where threholds are.
`ncol`	Number of columns of facets.
`print`	Print the plot? Default = TRUE
`...`	Unused

Value

A ggplot object, invisibly.

See Also

get_thresholds

Examples

```
m <- machine_learn(pima_diabetes[1:100, ], patient_id, outcome = diabetes,
                   models = "xgb", tune = FALSE, n_folds = 3)

get_thresholds(m) %>%
  plot()

get_thresholds(m, optimize = "cost", measures = c("acc", "cost"), cost_fn = 3) %>%
  plot(point_size = .5, ncol = 1)
```

---

plot.variable_importance

Plot variable importance

Description

Plot variable importance

Usage

```
## S3 method for class 'variable_importance'
plot(
  x,
  title = "model",
  max_char = 40,
  caption = NULL,
  font_size = 11,
  point_size = 3,
  print = TRUE,
  ...
)
```

Arguments

x	A data frame from get_variable_importance
title	Either "model", "none", or a string to be used as the plot caption. "model" puts the name of the best-performing model, on which variable importances are generated, in the title.

max_char	Maximum length of variable names to leave untruncated. Default = 40; use Inf to prevent truncation. Variable names longer than this will be truncated to leave the beginning and end of each variable name, bridged by " ... ".
caption	Plot title
font_size	Relative size for all fonts, default = 11
point_size	Size of dots, default = 3
print	Print the plot?
...	Unused

Value

A ggplot object, invisibly.

Examples

```
machine_learn(pima_diabetes[1:50, ], patient_id, outcome = diabetes, tune = FALSE) %>%
  get_variable_importance() %>%
  plot()
```

---

predict.model_list *Get predictions*

Description

Make predictions using the best-performing model. For classification models, predicted probabilities are always returned, and you can get either predicted outcome class by specifying outcome_groups or risk groups by specifying risk_groups.

Usage

```
## S3 method for class 'model_list'
predict(
  object,
  newdata,
  risk_groups = NULL,
  outcome_groups = NULL,
  prepdata,
  write_log = FALSE,
  ...
)
```

Arguments

`object`	model_list object, as from 'tune_models'
`newdata`	data on which to make predictions. If missing, out-of-fold predictions from training will be returned If you want new predictions on training data using the final model, pass the training data to this argument, but know that you're getting over-fit predictions that very likely overestimate model performance relative to what will be achieved on new data. Should have the same structure as the input to 'prep_data','tune_models' or 'train_models'. 'predict' will try to figure out if the data need to be sent through 'prep_data' before making predictions; this can be overridden by setting 'prepdata = FALSE', but this should rarely be needed.
`risk_groups`	Should predictions be grouped into risk groups and returned in column "predicted_group"? If this is NULL (default), they will not be. If this is a single number, that number of groups will be created with names "risk_group1", "risk_group2", etc. "risk_group1" is always the highest risk (highest predicted probability). The groups will have equal expected sizes, based on the distribution of out-of-fold predictions on the training data. If this is a character vector, its entries will be used as the names of the risk groups, in increasing order of risk, again with equal expected sizes of groups. If you want unequal-size groups, this can be a named numeric vector, where the names will be the names of the risk groups, in increasing order of risk, and the entries will be the relative proportion of observations in the group, again based on the distribution of out-of-fold predictions on the training data. For example, `risk_groups = c(low = 2,mid = 1,high = 1)` will put the bottom half of predicted probabilities in the "low" group, the next quarter in the "mid" group, and the highest quarter in the "high" group. You can get the cutoff values used to separate groups by passing the output of `predict` to `get_cutoffs`. Note that only one of `risk_groups` and `outcome_groups` can be specified.
`outcome_groups`	Should predictions be grouped into outcome classes and returned in column "predicted_group"? If this is NULL (default), they will not be. The threshold for splitting outcome classes is determined on the training data via `get_thresholds`. If this is TRUE, the threshold is chosen to maximize accuracy, i.e. false positives and false negatives are equally weighted. If this is a number it is the ratio of cost (badnesss) of false negatives (missed detections) to false positives (false alarms). For example, `outcome_groups = 5` indicates a preferred ratio of five false alarms to every missed detection, and `outcome_groups = .5` indicates that two missed detections is as bad as one false alarm. This value is passed to the `cost_fn` argument of `get_thresholds`. You can get the cutoff values used to separate groups by passing the output of `predict` to `get_cutoffs`. Note that only one of `risk_groups` and `outcome_groups` can be specified.
`prepdata`	Defunct. Data are always prepped in prediction.
`write_log`	Write prediction metadata to a file? Default is FALSE. If TRUE, will create or append a file called "prediction_log.txt" in the current directory with metadata about predictions. If a character, is the name of a file to create or append with prediction metadata. If you want a unique log file each time predictions are made, use something like `write_log = paste0(Sys.time()," predictions.txt")`. This param modifies error behavior and is best used in production. See details.
`...`	Unused.

Details

The model and hyperparameter values with the best out-of-fold performance in model training according to the selected metric is used to make predictions. Prepping data inside 'predict' has the advantage of returning your predictions with the newdata in its original format.

If `write_log` is TRUE and an error is encountered, `predict` will not stop. It will return the error message as: - A warning in the console - A field in the log file - A column in the "prediction_log" attribute - A zero-row data frame will be returned

Value

A tibble data frame: newdata with an additional column for the predictions in "predicted_TARGET" where TARGET is the name of the variable being predicted. If classification, the new column will contain predicted probabilities. The tibble will have child class "predicted_df" and attribute "model_info" that contains information about the model used to make predictions. You can call `plot` or `evaluate` on a predicted_df. If `write_log` is TRUE and this function errors, a zero-row dataframe will be returned.

Returned data will contain an attribute, "prediction_log" that contains a tibble of logging info for writing to database. If `write_log` is TRUE and predict errors, an empty dataframe with the "prediction_log" attribute will still be returned. Extract this attribute using `attr(pred,"prediction_log")`.

Data will also contain a "failed" attribute to easily filter for errors after prediction. Extract using `attr(pred,"failed")`.

See Also

`plot.predicted_df`, `evaluate.predicted_df`, `get_thresholds`, `get_cutoffs`

Examples

```
### Data prep and model training ###
####################################

set.seed(7510)
# Split the first 200 rows in pima_diabetes into a model-training dataset
# containing 3/4 of the data and a test dataset containing 1/4 of the data.
d <- split_train_test(pima_diabetes[1:200, ], diabetes, .75)

# Prep the training data for model training and train regularized regression
# and extreme gradient boosted models
models <-
  d$train %>%
  prep_data(patient_id, outcome = diabetes) %>%
  flash_models(outcome = diabetes, models = c("glm", "xgb"))

### Making predictions ###
##########################

# Make prediction on test data using the model that performed best in
# cross validation during model training. Before predictions are made, the test
# data is automatically prepared the same way the training data was.
```

```
predictions <- predict(models, newdata = d$test)
predictions
evaluate(predictions)
plot(predictions)

### Outcome class predictions ###
#################################

# If you want class predictions in addition to predicted probabilities for
# a classification model, specify outcome_groups. The number passed to
# outcome groups is the cost of a false negative relative to a false positive.
# This example specifies that one missed detection is as bad as ten false
# alarms, and the resulting confusion matrix reflects this preference.
class_preds <- predict(models, newdata = d$test, outcome_groups = 10)
table(actual = class_preds$diabetes, predicted = class_preds$predicted_group)

# You can extract the threshold used to separate predicted Y from predicted N
get_cutoffs(class_preds)

# And you can visualize that cutoff by simply plotting the predictions
plot(class_preds)

### Risk stratification ###
###########################

# Alternatively, you can stratify observations into risk groups by specifying
# the risk_groups parameter. For example, this creates five risk groups
# with custom names. Risk group assignment is based on the distribution of
# predicted probabilities in model training. This is useful because it preserves
# a consistent notion of risk; for example, if you make daily predictions and
# one day happens to contain only low-risk patients, those patients will all
# be classified as low risk. Over the long run, group sizes will be consistent,
# but in any given round of predictions they may differ. If you want fixed
# group sizes, see the following examples.
predict(models, d$test,
        risk_groups = c("very low", "low", "medium", "high", "very high")) %>%
  plot()

### Fixed size groups ###
#########################

# If you want groups of fixed sizes, e.g. say you have capacity to admit the three
# highest-risk patients, treat the next five, and have to discharge the remainder,
# you can use predicted probabilities to do that. One way to do that is to
# arrange the predictions data frame in descending order of risk, and then use the
# row numbers to stratify patients
library(dplyr)
predict(models, d$test) %>%
  arrange(desc(predicted_diabetes)) %>%
  mutate(action = case_when(
    row_number() <= 3 ~ "admit",
    row_number() <= 8 ~ "treat",
    TRUE ~ "discharge"
```

```
  )) %>%
  select(predicted_diabetes, action, everything())

# Finally, if you want a fixed group size that is further down on the risk
# scale, you can achieve that with a combination of risk groups and the
# stratifying approach in the last example. For example, say you have capacity
# to admit 5 patients, but you don't want to admit patients in the top 10% of
# risk scores.
predict(models, d$test,
        risk_groups = c("risk acceptable" = 90, "risk too high" = 10)) %>%
  filter(predicted_group == "risk acceptable") %>%
  top_n(n = 5, wt = predicted_diabetes)
```

`prep_data` *Prepare data for machine learning*

Description

`prep_data` will prepare your data for machine learning. Some steps enhance predictive power, some make sure that the data format is compatible with a wide array of machine learning algorithms, and others provide protection against common problems in model deployment. The following steps are available; those followed by * are applied by default. Many have customization options.

1. Convert columns with only 0/1 to factor*
2. Remove columns with near-zero variance*
3. Convert date columns to useful features*
4. Fill in missing values via imputation*
5. Collapse rare categories into "other"*
6. Center numeric columns
7. Standardize numeric columns
8. Create dummy variables from categorical variables*
9. Add protective levels to factors for rare and missing data*
10. Convert columns to principle components using PCA

While preparing your data, a recipe will be generated for identical transformation of future data and stored in the ‘recipe‘ attribute of the output data frame.If a recipe object is passed to ‘prep_data‘ via the ‘recipe‘ argument, thatrecipe will be applied to the data. This allows you to transform data inmodel training and apply exactly the same transformations in model testing and deployment. The new data must be identical in structure to the data that the recipe was prepared with.

Usage

```
prep_data(
  d,
  ...,
  outcome,
```

```
  recipe = NULL,
  remove_near_zero_variance = TRUE,
  convert_dates = TRUE,
  impute = TRUE,
  collapse_rare_factors = TRUE,
  PCA = FALSE,
  center = FALSE,
  scale = FALSE,
  make_dummies = TRUE,
  add_levels = TRUE,
  logical_to_numeric = TRUE,
  factor_outcome = TRUE,
  no_prep = FALSE
)
```

Arguments

d	A data frame
...	Optional. Columns to be ignored in preparation and model training, e.g. ID columns. Unquoted; any number of columns can be included here.
outcome	Optional. Unquoted column name that indicates the target variable. If provided, argument must be named. If this target is 0/1, it will be coerced to Y/N if factor_outcome is TRUE; other manipulation steps will not be applied to the outcome.
recipe	Optional. Recipe for how to prep d. In model deployment, pass the output from this function in training to this argument in deployment to prepare the deployment data identically to how the training data was prepared. If training data is big, pull the recipe from the "recipe" attribute of the prepped training data frame and pass that to this argument. If present, all following arguments will be ignored.
remove_near_zero_variance	Logical or numeric. If TRUE (default), columns with near-zero variance will be removed. These columns are either a single value, or the most common value is much more frequent than the second most common value. Example: In a column with 120 "Male" and 2 "Female", the frequency ratio is 0.0167. It would be excluded by default or if ‘remove_near_zero_variance‘ > 0.0166. Larger values will remove more columns and this value must lie between 0 and 1.
convert_dates	Logical or character. If TRUE (default), date and time columns are transformed to circular representation for hour, day, month, and year for machine learning optimization. If FALSE, date and time columns are removed. If character, use "continuous" (same as TRUE), "categories", or "none" (same as FALSE). "categories" makes hour, day, month, and year readable for interpretation. If make_dummies is TRUE, each unique value in these features will become a new dummy variable. This will create wide data, which is more challenging for some machine learning models. All features with the DTS suffix will be treated as a date.

impute Logical or list. If TRUE (default), columns will be imputed using mean (numeric), and new category (nominal). If FALSE, data will not be imputed. If this is a list, it must be named, with possible entries for 'numeric_method', 'nominal_method', 'numeric_params', 'nominal_params', which are passed to hcai_impute.

collapse_rare_factors
Logical or numeric. If TRUE (default), factor levels representing less than 3 percent of observations will be collapsed into a new category, 'other'. If numeric, must be in 0, 1, and is the proportion of observations below which levels will be grouped into other. See 'recipes::step_other'.

PCA Integer or Logical. PCA reduces training time, particularly for wide datasets, though it renders models less interpretable." If integer, represents the number of principal components to convert the numeric data into. If TRUE, will convert numeric data into 5 principal components. PCA requires that data is centered and scaled and will set those params to TRUE. Default is FALSE.

center Logical. If TRUE, numeric columns will be centered to have a mean of 0. Default is FALSE, unless PCA is performed, in which case it is TRUE.

scale Logical. If TRUE, numeric columns will be scaled to have a standard deviation of 1. Default is FALSE, unless PCA is performed, in which case it is TRUE.

make_dummies Logical or list. If TRUE (default), dummy columns will be created for categorical variables. When dummy columns are created, columns are not created for reference levels. By default, the levels are reassigned so the mode value is the reference level. If a named list is provided, those values will replace the reference levels. See the example for details.

add_levels Logical. If TRUE (default), "other" and "missing" will be added to all nominal columns. This is protective in deployment: new levels found in deployment will become "other" and missingness in deployment can become "missing" if the nominal imputation method is "new_category". If FALSE, these "other" will be added to all nominal variables if collapse_rare_factors is used, and "missingness" may be added depending on details of imputation.

logical_to_numeric
Logical. If TRUE (default), logical variables will be converted to 0/1 integer variables.

factor_outcome Logical. If TRUE (default) and if all entries in outcome are 0 or 1 they will be converted to factor with levels N and Y for classification. Note that which level is the positive class is set in training functions rather than here.

no_prep Logical. If TRUE, overrides all other arguments to FALSE so that d is returned unmodified, except that character variables may be coverted to factors and a tibble will be returned even if the input was a non-tibble data frame.

Value

Prepared data frame with reusable recipe object for future data preparation in attribute "recipe". Attribute recipe contains the names of ignored columns (those passed to ...) in attribute "ignored_columns".

See Also

To let data preparation happen automatically under the hood, see machine_learn

To take finer control of imputation, see impute, and for finer control of data prep in general check out the recipes package: https://topepo.github.io/recipes/

To train models on prepared data, see tune_models and flash_models

Examples

```
d_train <- pima_diabetes[1:700, ]
d_test <- pima_diabetes[701:768, ]

# Prep data. Ignore patient_id (identifier) and treat diabetes as outcome
d_train_prepped <- prep_data(d = d_train, patient_id, outcome = diabetes)

# Prep test data by reapplying the same transformations as to training data
d_test_prepped <- prep_data(d_test, recipe = d_train_prepped)

# View the transformations applied and the prepped data
d_test_prepped

# Customize preparations:
prep_data(d = d_train, patient_id, outcome = diabetes,
          impute = list(numeric_method = "bagimpute",
                        nominal_method = "bagimpute"),
          collapse_rare_factors = FALSE, center = TRUE, scale = TRUE,
          make_dummies = FALSE, remove_near_zero_variance = .02)

# Picking reference levels:
# Dummy variables are not created for reference levels. Mode levels are
# chosen as reference levels by default. The list given to `make_dummies`
# sets the reference level for `weight_class` to "normal". All other values
# in `weight_class` will create a new dummy column that is relative to normal.
prep_data(d = d_train, patient_id, outcome = diabetes,
          make_dummies = list(weight_class = "normal"))

# `prep_data` also handles date and time features by default:
d <-
  pima_diabetes %>%
  cbind(
    admitted_DTS = seq(as.POSIXct("2005-1-1 0:00"),
                       length.out = nrow(pima_diabetes), by = "hour")
  )
d_train = d[1:700, ]
prep_data(d = d_train)

# Customize how date and time features are handled:
# When `convert_dates` is set to "categories", the prepped data will be more
# readable, but will be wider.
prep_data(d = d_train, convert_dates = "categories")

# PCA to reduce training time:
```

```
## Not run:
start_time <- Sys.time()
pd <- prep_data(pima_diabetes, patient_id, outcome = diabetes, PCA = FALSE)
ncol(pd)
m <- machine_learn(pd, patient_id, outcome = diabetes)
end_time <- Sys.time()
end_time - start_time

start_time <- Sys.time()
pcapd <- prep_data(pima_diabetes, patient_id, outcome = diabetes, PCA = TRUE)
ncol(pcapd)
m <- machine_learn(pcapd, patient_id, outcome = diabetes)
Sys.time() - start_time

## End(Not run)
```

`rename_with_counts` *Adds the category count to each category name in a given variable column*

Description

'rename_with_counts' concatenates the count of each category to its category name given a specific variable. It can be useful in plots and tables to display the fequency of categories of a variable (see the example below).

Usage

```
rename_with_counts(d, variable_name)
```

Arguments

`d`	a tibble or dataframe
`variable_name`	the column with counts wanted

Value

a tibble with the counts appended to the 'variable_name' column

Examples

```
rename_with_counts(pima_diabetes, weight_class)

# Below is an example of how `rename_with_counts` can be helpful when
# creating plots and tables. This graph shows the outcomes of different
# weight classes in `pima_diabetes`. With the added information from
# `rename_with_counts`, we can see how common each category is.
library(ggplot2)
rename_with_counts(pima_diabetes, weight_class) %>%
```

```
ggplot(aes(x = reorder(weight_class, diabetes, function(x) mean(x == "Y")),
           fill = diabetes)) +
geom_bar(position = "fill") +
coord_flip()
```

`save_models` *Save models to disk and load models from disk*

Description

Note that model objects contain training data, except columns ignored (patient_id in the example below). Therefore, if there is PHI in the training data, the saved model object must be treated as PHI. `save_models` issues a message saying as much.

Usage

```
save_models(x, filename = "models.RDS", sanitize_phi = TRUE)

load_models(filename)
```

Arguments

`x`	model_list object
`filename`	File path to save model to or read model from, e.g. "models/my_models.RDS". Default for `save_models` is "models.RDS" in the working directory (`getwd()`). Default for `load_models` is to open a dialog box from which a file can be selected, in which case a message will issued with code to load the same file without interactivity.
`sanitize_phi`	Logical. If TRUE (default) training data is removed from the model object before being saved. Removing training data is important when sharing models that were trained with data that contain PHI. If removed, `explore` will not have data to process.

Value

`load_models` returns the model_list which can be assigned to any variable name

Examples

```
## Not run:
m <- machine_learn(pima_diabetes, patient_id, outcome = diabetes)
save_models(m, "diabetes_models.RDS")
# Restart R, move RDS file to another computer, etc.
m2 <- load_models("diabetes_models.RDS")
all.equal(m, m2)

## End(Not run)
```

selectData *Defunct. See* db_read

Description

Removed in v2.0.0

Usage

```
selectData(...)
```

Arguments

...	Garbage collector

separate_drgs *Convert MSDRGs into a "base DRG" and complication level*

Description

Convert MSDRGs into a "base DRG" and complication level

Usage

```
separate_drgs(drgs, remove_age = FALSE)
```

Arguments

drgs	character vector of MSDRG descriptions, e.g. MSDRGDSC
remove_age	logical; if TRUE will remove age descriptions

Details

This function is not robust to different codings of complication in DRG descriptions. If you have a coding other than "W CC" / "W MCC" / "W CC/MCC" / "W/O CC" / "W/O MCC", please file an issue on Github and we'll try to add support for your coding.

Value

a tibble with three columns: msdrg: the input vector, base_msdrg, and msdrg_complication

Examples

```
MSDRGs <- c("ACUTE LEUKEMIA W/O MAJOR O.R. PROCEDURE W CC",
            "ACUTE LEUKEMIA W/O MAJOR O.R. PROCEDURE W MCC",
            "ACUTE LEUKEMIA W/O MAJOR O.R. PROCEDURE W/O CC/MCC",
            "SIMPLE PNEUMONIA & PLEURISY",
            "SIMPLE PNEUMONIA & PLEURISY AGE 0-17")
separate_drgs(MSDRGs, remove_age = TRUE)
```

---

split_train_test *Split data into training and test data frames*

Description

'split_train_test' splits data into two data frames for validation of models. One data frame is meant for model training ("train") and the other is meant to assess model performance ("test"). The distribution of `outcome` will be preserved acrosss the train and test datasets. Additionally, if there are groups in the dataset, you can keep all observations within a in the same train/test dataset by passing the name of the group column to `grouping_col`; this is useful, for example, when there are multiple observations per patient, and you want to keep each patient within one dataset.

Usage

```
split_train_test(d, outcome, percent_train = 0.8, seed, grouping_col)
```

Arguments

`d`	Data frame
`outcome`	Target column, unquoted. Split will be stratified across this variable
`percent_train`	Proportion of rows in d to put into training. Default is 0.8
`seed`	Optional, if provided the function will return the same split each time it is called
`grouping_col`	column name that specifies grouping. Individuals in the same group are in the same training/test set.

Details

This function wraps 'caret::createDataPartition'. If outcome is a factor then the test/training porportions are stratified. Otherwise they are randomly selected.

If the grouping_col is given, then the groups are divided into the test/ training porportions.

Value

A list of two data frames with names train and test

Examples

```
split_train_test(mtcars, am, .9)


# Below is an additional example of grouping. Grouping is where individuals
# in the same group are in the same training/test set. Here we group on car
# owners. Owners will be in the same training/test set.
library(dplyr)

mtcars %>%
  mutate(owner = rep(letters[1:16], each = 2)) %>%
  split_train_test(., am, grouping_col = owner)
```

---

start_prod_logs *Defunct*

Description

Defunct

Usage

```
start_prod_logs(...)
```

Arguments

...	Defunct

---

step_add_levels *Add levels to nominal variables*

Description

Add levels to nominal variables

Usage

```
step_add_levels(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  cols = NULL,
  levels = c("other", "missing"),
  skip = FALSE,
```

```
  id = rand_id("bagimpute")
)

## S3 method for class 'step_add_levels'
tidy(x, ...)
```

Arguments

recipe	recipe object. This step will be added
...	One or more selector functions
role	Ought to be nominal
trained	Has the recipe been prepped?
cols	columns to be prepped
levels	Factor levels to add to variables. Default = c("other", "missing")
skip	A logical. Should the step be skipped when the recipe is baked?
id	a unique step id that will be used to unprep
x	A ‘step_add_levels‘ object.

Value

Recipe with the new step

Examples

```
library(recipes)
d <- data.frame(num = 1:30,
                has_missing = c(rep(NA, 10), rep('b', 20)),
                has_rare = c("rare", rep("common", 29)),
                has_both = c("rare", NA, rep("common", 28)),
                has_neither = c(rep("cat1", 15), rep("cat2", 15)))
rec <- recipe( ~ ., d) %>%
  step_add_levels(all_nominal()) %>%
  prep(training = d)
baked <- bake(rec, d)
lapply(d[, sapply(d, is.factor)], levels)
lapply(baked[, sapply(baked, is.factor)], levels)
```

---

step_date_hcai *Date and Time Feature Generator*

Description

‘step_date_hcai‘ creates a *specification* of a recipe step that will convert date data into factor or numeric variable(s). This step will guess the date format of columns with the "_DTS" suffix, and then create either ‘categories‘ or ‘continuous‘ columns. Various portions of this step are copied from ‘recipes::step_date‘.

Usage

```
step_date_hcai(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  feature_type = "continuous",
  columns = NULL,
  skip = FALSE,
  id = rand_id("bagimpute")
)

## S3 method for class 'step_date_hcai'
tidy(x, ...)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
...	One or more selector functions to choose which variables that will be used to create the new variables. The selected variables should have class 'Date' or 'POSIXct' or their name must end with 'DTS'. See [selections()] for more details. For the 'tidy' method, these are not currently used.
role	For model terms created by this step, what analysis role should they be assigned? By default, the function assumes that the new variable columns created by the original variables will be used as predictors in a model.
trained	A logical to indicate if the number of NA values have been counted in preprocessing.
feature_type	character, either 'continuous' (default) or 'categories'.
columns	A character string of variables that will be used as inputs. This field is a placeholder and will be populated once [prep.recipe()] is used.
skip	A logical. Should the step be skipped when the recipe is baked?
id	a unique step id that will be used to unprep
x	A 'step_date_hcai' object.

Details

Unlike other steps, 'step_date_hcai' does *not* remove the original date variables. [step_rm()] can be used for this purpose.

Value

For 'step_date_hcai', an updated version of recipe with the new step added to the sequence of existing steps (if any). For the 'tidy' method, a tibble with columns 'terms' (the selectors or variables selected), 'value' (the feature names), and 'ordinal' (a logical).

Examples

```
library(lubridate)
library(recipes)

examples <- data.frame(Dan = ymd("2002-03-04") + days(1:10),
                       Stefan = ymd("2006-01-13") + days(1:10))
date_rec <- recipe(~ Dan + Stefan, examples) %>%
  step_date_hcai(all_predictors())

date_rec <- prep(date_rec, training = examples)

date_values <- bake(date_rec, new_data = examples)
date_values

# changing `feature_type` to `categories`
date_rec <-
  recipe(~ Dan + Stefan, examples) %>%
  step_date_hcai(all_predictors(), feature_type = "categories")

date_rec <- prep(date_rec, training = examples)

date_values <- bake(date_rec, new_data = examples)
date_values
```

---

step_dummy_hcai *Dummy Variables Creation*

Description

step_dummy_hcai creates a *specification* of a recipe step that will convert nominal data (e.g. character or factors) into one or more numeric binary model terms for the levels of the original data. Various portions of this step are copied from recipes::step_dummy. Beyond original recipes::step_dummy implementation, this step sets reference levels to provided reference levels or mode.

Usage

```
step_dummy_hcai(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  naming = dummy_names,
  levels = NULL,
  skip = FALSE,
  id = rand_id("bagimpute")
)
```

```
## S3 method for class 'step_dummy_hcai'
tidy(x, ...)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
...	One or more selector functions to choose which variables will be used to create the dummy variables. See [selections()] for more details. The selected variables must be factors. For the tidy method, these are not currently used.
role	For model terms created by this step, what analysis role should they be assigned?. By default, the function assumes that the binary dummy variable columns created by the original variables will be used as predictors in a model.
trained	A logical to indicate if the quantities for preprocessing have been estimated.
naming	A function that defines the naming convention for new dummy columns. See Details below.
levels	A list that provides the ordered levels of nominal variables. If all the unique values in a nominal variable are not included, the remaining values will be added to the given levels. The first level will be listed as the ref_level attribute for the step object. If levels are not provided for a nominal variable, the mode value will be used as the reference level.
skip	A logical. Should the step be skipped when the recipe is baked by bake.recipe()? While all operations are baked when prep.recipe() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations
id	A character string that is unique to this step to identify it.
x	A ‘step_dummy_hcai‘ object.

Details

step_dummy_hcai will create a set of binary dummy variables from a factor variable. For example, if an unordered factor column in the data set has levels of "red", "green", "blue", the dummy variable bake will create two additional columns of 0/1 data for two of those three values (and remove the original column). For ordered factors, polynomial contrasts are used to encode the numeric values.

By default, the excluded dummy variable (i.e. the reference cell) will correspond to the first level of the unordered factor being converted.

The function allows for non-standard naming of the resulting variables. For an unordered factor named ‘x‘, with levels ‘"a"‘ and ‘"b"‘, the default naming convention would be to create a new variable called ‘x_b‘. Note that if the factor levels are not valid variable names (e.g. "some text with spaces"), it will be changed by [base::make.names()] to be valid (see the example below). The naming format can be changed using the ‘naming‘ argument and the function [dummy_names()] is the default. This function will also change the names of ordinal dummy variables. Instead of values such as "‘.L‘", "‘.Q‘", or "‘^4‘", ordinal dummy variables are given simple integer suffixes such as "‘_1‘", "‘_2‘", etc.

To change the type of contrast being used, change the global contrast option via ‘options‘.

When the factor being converted has a missing value, all of the corresponding dummy variables are also missing.

When data to be processed contains novel levels (i.e., not contained in the training set), a missing value is assigned to the results. See [step_other()] for an alternative.

The [package vignette for dummy variables](https://topepo.github.io/recipes/articles/Dummies.html) and interactions has more information.

Value

An updated version of `recipe` with the new step added to the sequence of existing steps (if any). For the `tidy` method, a tibble with columns `terms` (the selectors or variables selected).

See Also

[step_factor2string()], [step_string2factor()], [dummy_names()], [step_regex()], [step_count()], [step_ordinalscore()], [step_unorder()], [step_other()] [step_novel()]

Examples

```
rec <- recipes::recipe(head(pima_diabetes), ~.) %>%
  healthcareai:::step_dummy_hcai(weight_class)
d <- recipes::prep(rec, training = pima_diabetes)
d <- recipes::bake(d, new_data = pima_diabetes)

# Specify ref_levels
ref_levels <- list(weight_class = "normal")
rec <- recipes::recipe(head(pima_diabetes), ~.)
rec <- rec %>% healthcareai:::step_dummy_hcai(weight_class,
                                              levels = ref_levels)
```

---

step_locfimpute *Last Observation Carried Forward Imputation*

Description

`step_locfimpute` creates a *specification* of a recipe step that will substitute missing values with the most recent variable value. If the first variable value is missing, it is imputed with the first present value.

Usage

```
step_locfimpute(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  skip = FALSE,
```

```
  id = rand_id("bagimpute")
)

## S3 method for class 'step_locfimpute'
tidy(x, ...)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
...	One or more selector functions to choose which variables will be imputed. See [selections()] for more details. For the 'tidy' method, these are not currently used.
role	Not used by this step since no new variables are created.
trained	A logical to indicate if the number of NA values have been counted in preprocessing.
skip	A logical. Should the step be skipped when the recipe is baked?
id	a unique step id that will be used to unprep
x	A 'step_locfimpute' object.

Value

For step_locfimpute, an updated version of recipe with the new step added to the sequence of existing steps (if any). For the tidy method, a tibble with columns terms (the selectors or variables selected) and trained (a logical that states whether the recipe has been prepped).

Examples

```
library(recipes)

prepped <-
  recipe(formula = "~.", pima_diabetes) %>%
  step_locfimpute(weight_class, insulin, skinfold, diastolic_bp) %>%
  prep()

bake(prepped, new_data = pima_diabetes)
```

---

step_missing *Clean NA values from categorical/nominal variables*

Description

step_missing creates a specification of a recipe that will replace NA values with a new factor level, missing.

Usage

```
step_missing(
  recipe,
  ...,
  role = NA,
  trained = FALSE,
  na_percentage = NULL,
  skip = FALSE,
  id = rand_id("bagimpute")
)

## S3 method for class 'step_missing'
tidy(x, ...)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
...	One or more selector functions to choose which variables are affected by the step. See ?recipes::selections() for more details.
role	Not used by this step since no new variables are created.
trained	A logical to indicate if the number of NA values have been counted in preprocessing.
na_percentage	A named numeric vector of NA percentages. This is NULL until computed by prep.recipe().
skip	A logical. Should the step be skipped when the recipe is baked?
id	a unique step id that will be used to unprep
x	A ‘step_missing‘ object.

Details

NA values are counted when the recipe is trained using prep.recipe. bake.recipe then fills in the missing values for the new data.

Value

An updated version of recipe with the new step added to the sequence of existing steps (if any). For the tidy method, a tibble with columns terms (the selectors or variables selected) and value (the NA counts).

Examples

```
library(recipes)
n = 100
d <- tibble::tibble(encounter_id = 1:n,
                    patient_id = sample(1:20, size = n, replace = TRUE),
                    hemoglobin_count = rnorm(n, mean = 15, sd = 1),
```

```
                    hemoglobin_category = sample(c("Low", "Normal", "High", NA),
                                                 size = n, replace = TRUE),
                    disease = ifelse(hemoglobin_count < 15, "Yes", "No")
)

# Initialize
my_recipe <- recipe(disease ~ ., data = d)

# Create recipe
my_recipe <- my_recipe %>%
  step_missing(all_nominal())
my_recipe

# Train recipe
trained_recipe <- prep(my_recipe, training = d)

# Apply recipe
data_modified <- bake(trained_recipe, new_data = d)
```

`stop_prod_logs` *Defunct*

Description

Defunct

Usage

```
stop_prod_logs(...)
```

Arguments

`...`	Defunct

`summary.missingness` *Summarizes data given by* `missingness`

Description

Interpreting `missingness` results from wide datasets is difficult. This function helps interpret missingness output by summarizing this output by listing: the percent of variables that contain missingness, the variable name of the variable with the maximum amount of missingness along with its percent of observations containing missing values, and a tibble that lists the top 5 missingness levels with the count of the number of variables associated with each level (0 missingness level is ignored). If there are no variables with missingness, a message that reports no missingness is printed and NULL is returned instead.

Usage

```
## S3 method for class 'missingness'
summary(object, ...)
```

Arguments

object	Data frame from missingness
...	Unused

Value

a tibble of the top 5 missingness percentage levels with the count of the number of variables associated with each level. If no missingness is found, NULL is returned instead.

Examples

```
missingness(pima_diabetes) %>%
  summary()
```

---

tune_models *Tune multiple machine learning models using cross validation to optimize performance*

Description

Tune multiple machine learning models using cross validation to optimize performance

Usage

```
tune_models(
  d,
  outcome,
  models,
  metric,
  positive_class,
  n_folds = 5,
  tune_depth = 10,
  hyperparameters = NULL,
  model_class,
  model_name = NULL,
  allow_parallel = FALSE
)
```

Arguments

d	A data frame from prep_data. If you want to prepare your data on your own, use prep_data(...,no_prep = TRUE).
outcome	Optional. Name of the column to predict. When omitted the outcome from prep_data is used; otherwise it must match the outcome provided to prep_data.
models	Names of models to try. See get_supported_models for available models. Default is all available models.
metric	Which metric should be used to assess model performance? Options for classification: "ROC" (default) (area under the receiver operating characteristic curve) or "PR" (area under the precision-recall curve). Options for regression: "RMSE" (default) (root-mean-squared error, default), "MAE" (mean-absolute error), or "Rsquared." Options for multiclass: "Accuracy" (default) or "Kappa" (accuracy, adjusted for class imbalance).
positive_class	For classification only, which outcome level is the "yes" case, i.e. should be associated with high probabilities? Defaults to "Y" or "yes" if present, otherwise is the first level of the outcome variable (first alphabetically if the training data outcome was not already a factor).
n_folds	How many folds to use in cross-validation? Default = 5.
tune_depth	How many hyperparameter combinations to try? Default = 10. Value is multiplied by 5 for regularized regression. Increasing this value when tuning XGBoost models may be particularly useful for performance.
hyperparameters	Optional, a list of data frames containing hyperparameter values to tune over. If NULL (default) a random, tune_depth-deep search of the hyperparameter space will be performed. If provided, this overrides tune_depth. Should be a named list of data frames where the names of the list correspond to models (e.g. "rf") and each column in the data frame contains hyperparameter values. See hyperparameters for a template. If only one model is specified to the models argument, the data frame can be provided bare to this argument.
model_class	"regression" or "classification". If not provided, this will be determined by the class of 'outcome' with the determination displayed in a message.
model_name	Quoted, name of the model. Defaults to the name of the outcome variable.
allow_parallel	Depreciated. Instead, control the number of cores though your parallel back end (e.g. with doMC).

Details

Note that this function is training a lot of models (100 by default) and so can take a while to execute. In general a model is trained for each hyperparameter combination in each fold for each model, so run time is a function of length(models) x n_folds x tune_depth. At the default settings, a 1000 row, 10 column data frame should complete in about 30 seconds on a good laptop.

Value

A model_list object. You can call plot, summary, evaluate, or predict on a model_list.

See Also

For setting up model training: prep_data, supported_models, hyperparameters

For evaluating models: plot.model_list, evaluate.model_list

For making predictions: predict.model_list

For faster, but not-optimized model training: flash_models

To prepare data and tune models in a single step: machine_learn

Examples

```
## Not run:
### Examples take about 30 seconds to run
# Prepare data for tuning
d <- prep_data(pima_diabetes, patient_id, outcome = diabetes)

# Tune random forest, xgboost, and regularized regression classification models
m <- tune_models(d)

# Get some info about the tuned models
m

# Get more detailed info
summary(m)

# Plot performance over hyperparameter values for each algorithm
plot(m)

# To specify hyperparameter values to tune over, pass a data frame
# of hyperparameter values to the hyperparameters argument:
rf_hyperparameters <-
  expand.grid(
    mtry = 1:5,
    splitrule = c("gini", "extratrees"),
    min.node.size = 1
  )
grid_search_models <-
  tune_models(d = d,
              outcome = diabetes,
              models = "rf",
              hyperparameters = list(rf = rf_hyperparameters)
  )
plot(grid_search_models)

## End(Not run)
```

---

writeData *Defunct. See* Rhrefhttps://docs.healthcare.ai/articles/site_only/db_connections.htmlthis *vignette for help writing to databases.*

Description

Removed in v2.0.0

Usage

```
writeData(...)
```

Arguments

...	Garbage collector

Index