Package 'healthcareai'

July 1, 2018

Type Package

Title Tools for Healthcare Machine Learning

Version 2.1.0

Date 2018-06-29

Description A machine learning toolbox tailored to healthcare data.

License MIT + file LICENSE

LazyData TRUE

Depends R (>= 3.3), methods

Imports broom, caret (>= 6.0.78), cowplot, data.table, dbplyr, dplyr, e1071, ggplot2, glmnet, lubridate, MLmetrics, purrr, ranger (>= 0.8.0), recipes (>= 0.1.3), rlang, stringr, tibble, tidyr, xgboost

RoxygenNote 6.0.1

Suggests DBI, odbc, testthat, lintr, covr

URL http://docs.healthcare.ai

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

NeedsCompilation no

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

Maintainer Michael Levy <michael.levy@healthcatalyst.com>

Repository CRAN

Date/Publication 2018-07-01 06:12:03 UTC

R topics documented:

`add_best_levels` *Build efficient features from 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::data_frame(
    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*

Description

This function is called in prep_data and so shouldn't usually need to be called directly. It tries to convert columns ending in "DTS" to type Date. It makes a best guess at the format and return a more standard one if possible. Times are be removed.

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

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_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))
```

---

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
outcome	Name of the column to predict
models	Names of models to try. See get_supported_models for available models. Default is all available models.
metric	What metric to use to assess model performance? Options for regression: "RMSE" (root-mean-squared error, default), "MAE" (mean-absolute error), or "Rsquared." For classification: "ROC" (area under the receiver operating characteristic curve), or "PR" (area under the precision-recall curve).
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	Logical, defaults to FALSE. If TRUE and a parallel backend is set up (e.g. with doMC) models with support for parallel training will be trained across cores.

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, diabetes)

# 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_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_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 <- flash_models(mtcars, outcome = mpg, models = "rf")
get_variable_importance(m)
```

`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"` or `"knnimpute"`.
`numeric_method`	Defaults to `"mean"`. 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_options` (bagimpute only), `knn_K`, (knnimpute only), `impute_with`, (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_options` (bagimpute only), `knn_K`, (knnimpute only), `impute_with`, (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, newdata = d)
missingness(data_modified)


# Specify methods:
my_recipe <- my_recipe %>%
  hcai_impute(numeric_method = "bagimpute",
    nominal_method = "new_category")
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_options (bagimpute only), knn_K, (knnimpute only), impute_with, (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_options (bagimpute only), knn_K, (knnimpute only), impute_with, (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)
```

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, 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.
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	Logical, defaults to FALSE. If TRUE and a parallel backend is set up (e.g. with doMC) models with support for parallel training will be trained across cores.

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)

## End(Not run)
```

`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()
```

---

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

---

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.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 <- tune_models(mtcars, 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, 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, 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
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.

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

```
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 = 14)
p <- plot(predictions, print = FALSE)
p + coord_fixed(ratio = 1) + theme_classic()
```

---

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` *Make predictions using the best-performing model*

Description

Make predictions using the best-performing model

Usage

```
## S3 method for class 'model_list'
predict(object, newdata, 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 overriden by setting ‘prepdata = FALSE‘, but this should rarely be needed.
`prepdata`	Logical, this should rarely be set by the user. By default, if ‘newdata‘ hasn’t been prepped, it will be prepped by ‘prep_data‘ before predictions are made. Set this to TRUE to force already-prepped data through ‘prep_data‘ again, or set to FALSE to prevent ‘newdata‘ from being sent through ‘prep_data‘.
`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`

Examples

```
# Tune models using only the first 40 rows to keep computation fast

models <- machine_learn(pima_diabetes[1:40, ], patient_id,
                        outcome = diabetes, tune = FALSE)

# Make prediction on the next 10 rows. This uses the best-performing model from
# tuning cross validation, and it also prepares the new data in the same way as
# the training data was prepared.

predictions <- predict(models, newdata = pima_diabetes[41:50, ])
predictions
evaluate(predictions)
plot(predictions)
```

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*

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, that recipe will be applied to the data. This allows you to transform data in model 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,
  center = FALSE, scale = FALSE, make_dummies = TRUE, add_levels = TRUE,
  factor_outcome = TRUE)
```

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 columns are identifed and used to generate day-of-week, month, and year columns, and the original date columns are removed. If FALSE, date columns are removed. If a character vector, it is passed to the 'features' argument of 'recipes::step_date'. E.g. if you want only quarter and year back: 'convert_dates = c("quarter", "year")'.

`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'.

`center` Logical. If TRUE, numeric columns will be centered to have a mean of 0. Default is FALSE.

`scale` Logical. If TRUE, numeric columns will be scaled to have a standard deviation of 1. Default is FALSE.

`make_dummies` Logical. If TRUE (default), dummy columns will be created for categorical variables.

`add_levels` Logical. If TRUE (defaults), "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.

`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.

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, convert_dates = "year",
          center = TRUE, scale = TRUE, make_dummies = FALSE,
          remove_near_zero_variance = .02)
```

---

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")

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.

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 data into training and test data frames

Usage

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

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

Details

This function wraps 'caret::createDataPartition'.

Value

A list of two data frames with names train and test

Examples

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

---

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)

## 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?
`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 Feature Generator*

Description

‘step_date_hcai‘ creates a a *specification* of a recipe step that will convert date data into one or more factor or numeric variables. It is a copy of ‘recipes::step_date‘ but will try to guess the date format of columns with the "_DTS" suffix.

Usage

```
step_date_hcai(recipe, ..., role = "predictor", trained = FALSE,
  features = c("dow", "month", "year"), abbr = TRUE, label = TRUE,
  ordinal = FALSE, columns = NULL, skip = FALSE)

## 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‘. 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.
`features`	A character string that includes at least one of the following values: ‘month‘, ‘dow‘ (day of week), ‘doy‘ (day of year), ‘week‘, ‘month‘, ‘decimal‘ (decimal date, e.g. 2002.197), ‘quarter‘, ‘semester‘, ‘year‘.
`abbr`	A logical. Only available for features ‘month‘ or ‘dow‘. ‘FALSE‘ will display the day of the week as an ordered factor of character strings, such as "Sunday". ‘TRUE‘ will display an abbreviated version of the label, such as "Sun". ‘abbr‘ is disregarded if ‘label = FALSE‘.
`label`	A logical. Only available for features ‘month‘ or ‘dow‘. ‘TRUE‘ will display the day of the week as an ordered factor of character strings, such as "Sunday." ‘FALSE‘ will display the day of the week as a number.
`ordinal`	A logical: should factors be ordered? Only available for features ‘month‘ or ‘dow‘.
`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?
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, newdata = examples)
date_values
```

---

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)
```

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?

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, newdata = d)
```

stop_prod_logs *Defunct*

Description

Defunct

Usage

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

Arguments

...	Defunct

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
outcome	Name of the column to predict
models	Names of models to try. See get_supported_models for available models. Default is all available models.
metric	What metric to use to assess model performance? Options for regression: "RMSE" (root-mean-squared error, default), "MAE" (mean-absolute error), or "Rsquared." For classification: "ROC" (area under the receiver operating characteristic curve), or "PR" (area under the precision-recall curve).
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	Logical, defaults to FALSE. If TRUE and a parallel backend is set up (e.g. with doMC) models with support for parallel training will be trained across cores.

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, outcome = diabetes)

# 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* R*hrefhttps://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